1997
6.270 Autonomous LEGO Robot Design Competition
Course Notes

To obtain additional copies of this document, write to:

6.270 Organizers
The EECS Department
The Massachusetts Institute of Technology
50 Vassar Street Room
Cambridge, MA 02139

or by sending electronic mail to `6.270-organizers@mit.edu`

Chapter 2 was revised for 1997 by Yonah Schmeidler.

Chapter 3 was written by Fred Martin and Pankaj Oberoi, and has been revised by Pankaj Oberoi and Yonah Schmeidler from previous 6.270 notes.

Chapter 4 was written by Fred Martin and Pankaj Oberoi and has been revised by Pankaj Oberoi and Anne Wright to include parts locations, functionality of parts, and to reflect changes in the hardware. The section has been modified by Matt Domsch, Karsten Ulland, and Pankaj Oberoi to reflect the modifications to the board.

Chapter 5 was written by Matt Domsch, Pankaj Oberoi, and Karsten Ulland.

Chapter 6 was revised by Anne Wright from Fred Martin's *Robot Builder's Guide* to reflect changes in the compiler and libraries.

Chapter 7 was written by Randy Sargent.

Chapter 8 was written by Sanjay Vakil and edited by Pankaj Oberoi.

Appendix A was written by Fred Martin and has been painstakingly revised by Anne Wright, Karsten Ulland, Matt Domsch, and Pankaj Oberoi to reflect changes in the hardware.

Appendix B was written by Fred Martin and revised by Matt Domsch and Pankaj Oberoi to reflect changes in the hardware.

Appendix C was written by Fred Martin, *Robot Builder's Guide.*

All other work was co-authored by the organizers of the 1992 6.270 course.

The format for the course notes is based on previous 6.270 course notes.

Preface

The origins of this course begin with Woody Flowers in MIT's Mechanical Engineering department. Woody Flowers had the idea that teaching should be interactive and not just lecturing. He developed the famous "Introduction to Design" class (course number 2.70). In 2.70, undergraduates use scrap parts—metal, plastic, and wood—to build machines that go on to compete in a head-to-head contest at the end of the course.

Michael B. Parker, an undergraduate in MIT's Electrical Engineering and Computer Science (Course Six) department, had just taken 2.70. Mike liked the course so much that he was jealous: "Why should there be a course like this for Mechanical Engineering students, but not for the students in his department?" he thought.

So in 1987, Mike organized the first 6.270 contest as "Course Six's answer" to the 2.70 course. The contest was a programming competition in which students wrote programs to control computer-simulated robots. In the first two years of the contest, the goal was to design a simulated robot that tried to find and destroy other robots. Unlike the machines that are built in the 2.70 course, there was no human control of the simulated robots (in 2.70 the students control the machines through a joystick and some switches). This was what separated the 2.70 course and the 6.270 contest.

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

Mike recruited Fred Martin and Randy Sargent to be the technical consultants to the upcoming 6.270 contest (which was starting in just a few weeks). It sounded like a fun way to spend IAP.[1]

Randy and Fred spent most of their holiday break designing an interface board that connected to a PC or Macintosh computer, controlling motors and providing input from a few simple sensors. The budget was tight and time was short as they

[1]MIT's "Independent Activities Period" is a one-month break between the fall and spring semesters.

scurried about the local Radio Shack stores, buying electronic parts for the twenty teams of students who had preregistered for the course.

Everything went wrong that month: the LEGO parts showed up late; Athena – the campus computer network – would not give approval for 6.270 to use its workstation and mix-ups in room scheduling forced groups of students wielding soldering irons to wander about campus looking for available classrooms.

The robots were powered and controlled through a tether connected to a personal computer. A lounge was transformed into a workshop, and only the students' excitement carried the contest through the month and into a competition at the end of IAP. Students who were building robots contributed their time to making the contest a reality. The contest lasted about four hours—it was a long, drawn-out affair—but the students enjoyed it.

The unique feature of 6.270 was that the students were running the course themselves. The learning was through mutual help. No professors or faculty members were involved with the organization or teaching of the course. Students learned from interaction with other students, and this formed the basis for future 6.270 contests.

After the contest was over, Pankaj ("P.K.") Oberoi, who had been a student in the contest, called a meeting of interested past participants. P.K. thought the class had great potential as a learning project and wanted to help organize it for the upcoming year. He felt that given some organization and structure, the contest could be transformed into a course in which the students could learn more than just how to "hack" something together.

P.K. had already worked on key administrative aspects, like recruiting corporate sponsors. Microsoft had donated some money to the course previously and was willing to up the ante for another year. P.K. also got the support of Motorola, which agreed to donate valuable semiconductor parts. He also gained the support of the Course 6 faculty to allow the contest for another year, with a more structured environment.

Randy and Fred were recruited once again by P.K. to help develop technology for the course. Hesitant at first, they laid out artwork for a custom printed circuit board that used a new Motorola microprocessor with more control features than the tethered machines. The original board was based on a microprocessor board designed at the MIT AI Laboratory by Henry Minsky.

P.K. and Fred wrote handouts for the students telling them how to build sensors, a battery charger, and other robotics components. The contest was transformed from a group of students wandering around looking for a place to work into a course with lectures and labs in where the students could learn more technical aspects in addition to the hands-on exposure they were getting.

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

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

By providing the tools and materials for students to work with complex electronic, mechanical, and software ideas, 6.270 gives the students a place to explore and learn about key ideas in technology, engineering, and design. Teamwork, learning-by-doing, and learning from one's peers is primary. 6.270 provides a hands-on style of learning experience for MIT students who are used to the theoretical lecturing style presented in most of the core classes.

The 1990 class was a big success but the organizers wanted to make the class even better. The class was hampered by a controller board which had to be programmed in assembly language and only had a small amount of memory. After the contest, work began on technology that would be more powerful and more useful to 6.270 students, allowing them to get even deeper into robot design and other technological issues.

By the start of the 1991 class, Fred and Randy had developed a robot-building kit with the high degree of power and flexibility we had wanted. Students were able to develop software for their robot using a subset of the C programming language called IC. The new embedded controller board they developed had a number of new features, including a small display screen that could be used to print debugging messages. Students were able to use powerful Unix workstations all across campus to develop programs for their robots.

The 1992 contest was organized by Fred, Randy, and P.K. joined by Karsten Ulland and Matt Domsch, two sophomores who had taken the course the previous year. New features continued to be added. Fred changed the layout of the board to include more channels of analog input for sensors; Randy improved IC to allow multiple C and binary modules to be loaded to the board; and PK, Karsten, and Matt worked staffing the labs during the day and building the contest tables at night.

The enthusiasm in the course was increasing by enormous proportions. This was evident in the increase in enrollment. Over 300 students signed up to take the course for 150 spots. Several new sponsors were recruited to participate in the course. As in previous years, many participants in the course also contributed towards helping prepare for the contest. The rapid change in technology did not allow much time for debugging or completion of the software. Anne Wright, a student in the course helped take up the slack by reporting bugs and writing software updates.

1993 organizers consisted of P.K., Matt, and Karsten joined by Anne Wright and Sanjay Vakil, two participants from the 1992 contest. They worked to update the technology and notes for the course and to organize the course for a new group of approximately 170 students divided into 55 teams. With Fred and Randy no longer active organizers, they set about to establish a new division of labor which would allow the standards of progress set by the previous contests to continue.

Matt took responsibility of class registration; P.K., Karsten, and Sanjay contacted greater support from the sponsors and ordered the kits; Karsten and Anne made changes to the controller board to fix hardware "bugs" from the previous year and some additional features (a servo); and Anne rewrote portions of the IC compiler to give it more of the features of real C. These course notes have come about from a combination of past experiences and each organizer took on the section of the notes with which they felt most comfortable.

In 1993, over 125 teams signed up to take the course. Because of the popularity and demand, the lab was kept open most evenings. To handle the demands of the course, eight TAs were hired from 6.270 alumni. Tripling the staff helped to offload the organizers. In addition, Motorola sent a film crew to the contest to document the contest from the distribution of the kits to the final round on contest night.

1994's contest expanded once more: 10 TAs and five organizers helped to create the contest. Of the 265 registrants came fifty teams. A new weighting system was also used to try and give past registrants who were not accepted a better chance at getting into the course. Given that the boards changed minimally, the extra time was spent updating and clarifying the course notes.

Unfortunately, due to budget and staff constraints, participation was back to fourty teams (about 110 students) in 1996. If you're taking the class now and would like to see it grow again, speak to an organizer about helping organize next year's contest.

Over the years, the organizers of the 6.270 contest have put a great deal of thought into how to organize the class to maximize the students learning potential and enjoyment of the course. We have tried at all times to provide the best educational and technical environment we could. We have tried to schedule events such that teams have a maximum amount of time to experiment with design and programming issues with a functioning robot. But the thing which really sets 6.270 apart from all the other technical and lab courses at MIT is that despite the enormity, this project remains to be one of the largest student-run activities due to the enthusiasm of the students who have taken the course. We believe that this is 6.270's greatest strength, and we hope that this enthusiasm will continue to attract generations of student organizers to keep 6.270 alive.

The 6.270 Organizers
January 2, 1997

MIT Departments

Two departments have played a major role in the development and running of the 6.270 project. We are grateful to both departments who have helped us, and for the encouragement given to us by many individuals in those departments.

Electrical Engineering and Computer Science (EECS) Department

The EECS department is the main supporter for the course and provides the lecture and laboratory space to run the course for 120 students. The EECS department provides for lab technicians to ensure that the labs are open during late hours. The department also funds some of the organizers. In addition, the EECS department provides funding in excess of $16,000 so that students are able to keep their kits when the course is over. The department offers this course for 6 units of credit to students wishing to get credit for the course.

MIT Media Laboratory

The Media Lab has been a past supporter by assisting in providing development resources for the 6.270 technology from 1990 to 1992. The Media Lab has sponsored Fred Martin's work on the development of the course and his research on the learning process of the students in the course.

Sponsors

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

Microsoft Corp.

Motorola Semiconductor Inc.

LEGO Systems Inc.

Hawker Energy Products Inc.

Polaroid Inc.

MIT EECS Department

Introduction to this Guide

These course notes contain a lot of information that should assist you in developing a working robot by the end of the course. The notes have been arranged in the order in which you should proceed. Many of the topics can be done simultaneously by different members of the team so as to multitask. We suggest that you follow a strict approach through these notes and that you read before you make a mistake.

- Chapter 1 is an introduction to the course. It contains all the administrative information you need.
- Chapter 2 contains the rules and guidelines for this year's contest.
- Chapter 3 explains good assembly technique and how to determine different component types.
- Chapter 4 is the assembly manual for nearly all of the components used to construct a 6.270 robot—including the printed circuit boards, electronic sensors, motors, and battery packs.
- Chapter 5 delves into various robotic sensors, explaining the principles of operation and applications of various sensors in the 6.270 kit. This section also include assembly instructions for the sensors.
- Chapter 6 is a reference manual for the C language software that has been developed for the 6.270 contest.
- Chapter 7 is a chapter which will help you get started with designing your LEGO robot. It goes into to intricacies of building strong LEGO machines.
- Chapter 8 investigates how to program a mobile robot to face up to the uncertainties and challenges of practical operation.

Three appendix sections provide additional material:

- Appendix A explains the workings of the 6.270 hardware, including the microprocessor board, the expansion board, and the infrared circuitry. This section is written with the assumption of some prior background in electronics.
- Appendix B contains copies of the printed circuit board artwork patterns for the 6.270 Rev. 2.21 boards.
- Appendix C discusses battery technology and battery charger operation.

Contents

List of Figures

Chapter 1

Introduction to 6.270

6.270 is a hands-on, learn-by-doing class in which participants design and build a robot that will play in a competition at the end of IAP. The goal for the students is to design a machine that will be able to navigate its way around the playing surface, recognize other opponents, and manipulate game objects. Unlike the machines in Introduction to Design (2.70), 6.270 robots are totally autonomous, so once a round begins, there is no human intervention (in 2.70 the machines are controlled by joystick).

The goal of 6.270 is to teach students about robotic design by giving them the hardware, software, and information they need to design, build, and debug their own robot. The subject includes concepts and applications that are related to various MIT classes (e.g. 6.001, 6.002, 6.004, and 2.70). *However,* there are no formal prerequisites for 6.270. We've found that people can learn everything they need to know by working with each other, being introduced to some material in class, and mostly, by hacking on their robots. All undergraduate students, from freshmen to seniors, are encouraged to register and take the class.

One caveat: 6.270 does require that you be psyched to put forth a real effort! We expect most students to spend about eighty hours over the month of IAP building their robots. Other commitments during the month of IAP are not recommended. We've also noticed that people who make a real commitment to the class are more confident, feel more involved, and have a lot more fun. So, if you are going to take 6.270, be ready for a month-long immersion into robotics!

1.1 Registration Policy

Registration in the class is limited to forty (40) teams. We would accept more students if resources permitted, but they do not.

All entrants will be organized into teams. There are a couple of reasons for this. First, we find that people learn a lot in the close and intense relationship of a small team. Second, we think the class would be too much work for one person to handle alone.

This class will take up enough of your time that you will not be able to work on other projects such as another course, UROP, or Thesis. Past students that have tried to do several time consuming projects have usually dropped out of the subject, or have not been able to produce a working robot. This year the duration of the subject will be shorter than past years because of the shortening of the IAP period so it is especially important you estimate the amount of time you will have before registering.

You are encouraged to form a team of two to three people and register together. You may also register alone, in which case we will find you a team with two other people.

1.2 Kit Fee and Toolkit Fee

Your 6.270 kit, which is yours to keep at the end of the contest, is valued at about \$750. The class is mostly financed by our commercial sponsors (namely Microsoft, Motorola, LEGO, Hawker, and Polaroid) and Course Six, but part of the budget is derived from the entry fee.

The team will be required to forfeit the kit back to the EECS department if it fails to present something to the organizers by the preliminary round of the contest (Monday, January 27th). Teams that do not return their kit once it is forfeited or lose their kit will be charged the full \$750 for the kit through the Bursar's office.

Separate from the 6.270 kit, a complete set of electronic hand tools will be reserved for purchase by your team. This kit will include a soldering iron, diagonal cutters, long nose pliers, wire strippers, a multimeter, and several other useful implements.

The 6.270 tool kit will have a retail value between \$75 and \$100; we expect to sell the kits for between \$50 and \$60 (we can give you these prices due to the quantity discounts we get in purchasing for the class). *You will be expected to either provide your own electronic assembly tools or purchase the standard tool kit.* It is very important that you have a good set of tools to work with. You will save many hours of debugging and frustration if you use good tools and assemble the material carefully. A sharp tipped soldering iron is essential to assembling your microprocessor board.

A final word about contest costs: if it is difficult for you to afford the contest costs, both the 6.270 kit and the toolkit are returnable (if in good condition) for a refund. If you would like to take the class, but you cannot afford to put up the money to register for the class and buy the toolkit, come talk to the organizers. We can probably work something out. Cost should not be a factor in determining whether you are able to take the course.

1.3 Credit Guidelines

6.270 is offered as MIT subject 6.190 for six units of Pass/Fail credit. Taking the class for credit is optional. You will be doing a lot of work in the class regardless; if you sign up for credit you will get official recognition for taking the class. If you sign up for credit but then do not complete the requirement, your registration will be dropped; it will be as if you never signed up in the first place.

Our job as instructors is to ensure that credit is properly awarded to students deserving of it. Our basic assumption is that anyone who is in the class is going to be doing a lot of work; the guidelines should add only a little bit of overhead to you in reporting your work to us. Hopefully, you may even learn a little more by going through the process of reporting on your progress.

1.3.1 Credit Guidelines

The following requirements for credit have been established:

- **Individual Journal Reports.**

  Each individual desiring credit must turn in a journal report that will be due on Thursday, January 30th. The journals are meant to help you with your thought processes. You should try to make an entry every day or every other day. The journals should include:

  - ideas that you have contributed to the development of the robot;
  - what management techniques your team is using;
  - strategies you have thought of;
  - problems you have encountered;
  - actual construction work, programming, or other tangible results.

  These ideas are examples of thoughts you might include. You are free to include anything else you think is appropriate. Pictures are a good way to try to convey your ideas and for reference.

  After the contest is over, you can pick up your journals to think about your ideas.

  The purpose of the individual journal is to get a sense of what each person on a team is contributing to the design, so it's important to make sure we know what you've done.

- **Team Video Reports.**

  In addition to the individual reports, a team video report will be made once per week.

  A video station will be set up in the 6.270 lab area. To make your report, you and your team can simply go to the camera and make a brief presentation on the status of your robot. This presentation should focus on issues that the team has worked on together, such as the current state of the robot, the strategy of the robot, and how the team arrived at a consensus (or not!) on particular issues.

  Hopefully, the video station concept will make the design reporting a fun and painless process. Any ideas presented to the camera *will remain confidential* for the duration of the contest.

- **Recitation Attendance.** You must attend at least three of the four meetings for your recitation section.

- **Completed Robot.** Your team must "show" a robot the day of Round 1. Its functionality (or lack thereof) has no affect on your receiving credit for the work you have done; the combination of the individual journals, the video reports, and class participation will be the main indicators of your involvement.

- **Program Listing.** You must turn in a copy of the program that your robot uses in the contest.

These subject requirements are meant to be useful to both you, the class participant, and the instructors, who will be authorizing credit. You should have no problem at all receiving credit if all of the requirements are satisfied. If you have any questions about your standing in the subject at any time, feel free to ask any of the instructors for feedback.

Please note that there is no leeway on any of the due dates, due to the scheduling constraints of the Registrar and the sanity of the organizers. Please do not ask for extensions.

1.3.2 Design Units

Since design is an important factor in 6.270, the EECS department will be offering 6 design units for EECS students that take 6.270. There are some guidelines and requirements for getting the design units.

First you must complete all the requirements to get credit for the course. It will come on your transcript. At the end of the contest, you must do an evaluation of your robot. In all design processes there should be some type of evaluation and redesign.

You will need to submit a 5-10 page paper. The paper should include, but need not be limited to, the following:

- An overall summary of your robot. This could include pictures or drawings.
- An evaluation of your robot's performance.
- Your individual redesign of the robot. If you were given an opportunity to retake the course with the same goals how would you make your robot different?
- Possible design flaws in the goals of the contest.

This paper should be submitted by Monday, February 24th to the EECS undergraduate office. The papers should be your individual evaluations, and not a general group evaluation.

1.4 Schedule

The schedule of activities between the start of IAP and the eve of the contest is very tight. You will have to work steadily and with determination to produce a working machine by the end of the course. In no fashion do we, the contest organizers, say that this course is not time consuming! In fact, we believe that you should be spending somewhere between 30 and 40 hours a week on average. However, since it is IAP, we can assume it is the main timesink you've signed up for.

There will be about 120 students taking the 1997 6.270 course, making it one of the largest courses taught during IAP. Since much of the learning, we believe, occurs with hands-on instruction, the class will be too large as a whole to teach on this basis. Therefore we have several class meeting formats, including lectures, recitations, lab demos, and lab sessions.

We recommend that you attend all of the lectures and recitations (for the section you are in) and **be on time**. We will deal with administrative and "bug fix" matters at the beginning of each meeting.

To make the course more personal, each organizer and TA will be the primary advisor for about 8 teams. The TA and organizer pair will be similar to the recitation instructor and TA pair you have in your normal classes. While these people are your primary advisors, you can approach anyone with questions you may have.

It is imperative that you check your e-mail often. Most notices will be posted through electronic mail. In addition, we will mention these notices in labs and lectures. You should check your mail at least once a day, if not more. This is the best way we can get in touch with the whole class on short notice.

- **General Lectures** The objective of the general lectures is to introduce you to the basics of the course. These sessions will try to give you an overview of the course and what you will be doing. The lectures will take place during the first week of the course. Since the students in this class typically have widely varying experience with the material, we will try to keep the lectures as general as possible.

  The lectures will also show you where to find advanced topics and more detailed answers for ambitious teams. There will be five basic lectures, from two to three hours each, to be held in 34-101. Check the schedule below for times and dates. It is important that you attend these lectures because they will give you the essential starting blocks.

- **Catch-Up IC Session** This is a general lecture for students who have had no C programming experience. We will go over the basics of the C language in particular how it applies to the IC language which will be used in the course. The main purpose of this lecture is to introduce basic concepts like variables, functions, and syntax. The lecture will be held in 34-101 on Thursday, January 9th at 12:00 PM.

- **Recitations** Detailed material will be presented in recitations rather than in lectures, to encourage a more interactive format. There will be several recitation sections, led by someone who has already taken 6.270 so they can tell you about their experiences and how to avoid the 6.270 pitfalls. The recitation leader will usually be one of your TA's or an organizer.

  The size of the recitation will be between 5 and 6 teams. The recitations are meant for group discussions, thinking about problems, sharing ideas, and experimenting. Many of the recitations will have hands-on experiments and will require you to have built sensors and motors.

  Recitations will be held during the second and third weeks of the course. There will be two recitations per week. The schedule for the recitations will be discussed during the first lecture.

- **Laboratory Sessions** This is supervised time for building your robot. Lab time will be critical when working on your circuit boards. After that, building motors and sensors will be important. During the final week, testing machines on the table will be the focus of lab activity.

  There will be smaller lab discussions where the TA's will give ideas on mounting sensors, soldering, programming, and general construction. It is also a good idea to use the lab facilities because there will be people there who can help you with your ideas. One of the goals of 6.270 is to teach interactively, and by working in the labs you will be able to share ideas with other people and experiment with ideas you may not have thought of.

Labs will be held on the 6th floor of building 38. They will be open from 8:45 AM to 11:45 PM during the weekdays, and noon to 10:00 PM on Saturday and Sunday. The final few days of the course, the lab will be open 24 hours.

1.4.1 Important Dates

Before reading the listing of the full month of meetings, please note the following *very important* meetings:

Parts-Sorting Session. Attendance at this session is mandatory: each team must provide one person-hour of manual labor helping to sort out the kit parts. Usually this session is a lot of fun as you get to meet other people in the class and see all of the electronic goodies.

Date, Time, and Place: Sunday, January 5th, 1:00 pm, Room 38-201 (The Chu Lounge).

Official Orientation Meeting. Attendance at this session is mandatory: each team must have at least 50% of its members in attendance. In this session, we will go over the contest rules and organization of the class, and hand out the kits.

Date, Time, and Place: Monday, January 6th, 10:00 am to 1:00 pm, Room 34-101.

The Contest, First Round. Your machine must compete in the first round to qualify for the second round.

Date, Time, and Place: Monday, January 27th, 6:00 pm, Room 26-100.

Robot Impounding. All work on robots will cease one day after the first round. All robots, including the ones that haven't made it past the second round will be impounded in 38-600.

Date, Time, and Place: Tuesday, January 28th, 6:00 pm, Room 38-600 (the lab).

The Contest, Second Round The second round of the double elimination contest will take place. There should be TV cameras to cover the event for local TV.

Date, Time, and Place: Wednesday, January 29th, 11:00 am, Room 26-100, **if necessary**.

The Contest, Final Round. Robots will be released from impoundment at 10:00 am, on Wednesday, January 29th. You must check your robot into 26-100 by 10:30 am. Good luck!

Date, Time, and Place: Wednesday, January 29th, 6:00 pm, Room 26-100.

1.4.2 Progress Schedule

The time allocated for the 6.270 course is short. There are only 21 days between the day you get your kit and the preliminary contest. It is therefore imperative that you set a personal schedule with goals before you begin the course. You may want to distribute the work among the team members in order to optimize team productivity.

Here is a checklist of important tasks you will need to do in order to make a working robot with the completion dates to prevent end of IAP stress:

- ○ **Course Notes** Read the Course notes as soon as possible. All of the details covered in class will be in the course notes. They contain the administrative material as well. You should read this by the end of the first week, **Friday, January 10th.** If you come and ask us a question without reading the notes, we will be more hesitant to answer your question.

- ○ **Microprocessor Board** The assembly of the microprocessor board should take between 10-15 hours for someone who has not soldered before. You should complete soldering by the morning of **Wednesday, January 8th.**

- ○ **Sensor Assembly** You should assemble your sensors early so that you can play around with them. This will take you about one day. Soldering the sensors together and testing their properties should be done by **Friday, January 10th.**

- ○ **Motor Assembly** You should "LEGOize" at least two motors so you can build a simple bot. By building a simple gearing mechanism early, you can test out the properties of the motor such as the torque and speed. We expect you to have simple gear assemblies being controlled by the microprocessor board by the evening of **Friday, January 10th.**

- ○ **Strategy** By the beginning to the middle of the second week your team should formulate a strategy for your robot. In the past teams have spent many days pondering over strategies. The indecisiveness usually leads to panic during the last week. You should get a strategy and stick with it rather than trying to restructure your strategy every day. To be at a reasonable pace, without too much stress at the end, you should have a defined strategy by **Monday, January 13rd.**

- **Simple Tasks** While you are formulating your strategy, your robot will need to do some simple tasks depending on the contest. You should use your simple bot and sensors to program these tasks. The tasks will be discussed in recitation. You should formulate your strategy, depending on how easy these task are. The tasks should be tested by **Wednesday, January 15th.**

- **Structure of Robot** During the first week, your team should "fool around" with the LEGO to get familiar with the structural properties. Once you have decided upon a strategy, you should complete the actual robot, with motor attachments and sensors by **Friday, January 17th.**

- **Programming** This is where you will have to tie everything together. You will need to combine your strategy, sensors, and robot to make the robot do what you want it to do. Do not underestimate the amount of time needed for this activity. Hopefully the simple tasks that you had a simple robot do during the first week will fit into your strategy. Complete your basic program by **Tuesday, January 21st.**

- **Debugging and Testing.** Your code probably won't work perfectly the first time you try it out. You should spend a few days testing out the machine and fixing any quirks it may have. This will be the long and tedious process of fine tuning your machine. By **Friday, January 24th,** you should have a pretty robust machine.

- **Mock Contest** We will hold a mock contest on the afternoon of **Saturday, January 25th** so that you can see how your machine performs against other machines. It is advisable to try your machine against other machines before this day.

- **Final Revisions** The final fine tuning of the machines can be done on **Monday, January 27th.**

Many of the teams that have done well in the past have been teams that have completed a final design and strategy early, and have left time to debug the machine. When the course is four weeks long, teams have the tendency to take the second week off because they feel they are ahead, and that programming is a cinch. Be aware, though, that you will want as much time as possible for debugging.

6.270 1997 Schedule: Week 1

Time	Monday 1/6	Tuesday 1/7	Wednesday 1/18	Thursday 1/9	Friday 1/10
10:00	**Opening Lecture** **34-101** General Information Distribute Kits Video Presentation	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36)	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36)	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36)	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36)
11:00					
12:00				**Beginners' C** **34-101**	
1:00					
2:00	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36) ***Soldering Demos Offered Periodically***				
3:00		**Lecture #2** **34-101** Team Organization The Board, Demos Brainstorming	**Lecture #3** **34-101** Sensors, Batteries, Motors, and LEGO	**Lecture #4** **34-101** Software Welcome to IC	**Lecture #5** **34-101** Integrating Systems Control Theory
4:00					
5:00					
6:00					
7:00	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36) ***Soldering Demos Offered Periodically***	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36)	**LEGO Lab** Group 1 34-301	**LEGO Lab** Group 1 34-301	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36)
8:00			**LEGO Lab** Group 2 34-302	**LEGO Lab** Group 2 34-302	
9:00			**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36)	**Lab Hours** **6.111 Lab** 38-6th floor (go up in 36)	
10:00					
11:00					
	• Solder boards • Experiment with LEGOs	• Finish soldering boards • Begin wiring motors and sensors	• Experiment with sensors and the processor board	• Begin programming simple tasks • Build simple robots	• Begin geartrain and chasis design • *Do video report #1 over the weekend!*

6.270 1997 Schedule: Week 2

Time	Monday 1/13	Tuesday 1/14	Wednesday 1/15	Thursday 1/16	Friday 1/17
10:00	**Lab Hours**	**Lab Hours**	**Lab Hours**	**Lab Hours**	**Lab Hours**
	6.111 Lab	**6.111 Lab**	**6.111 Lab**	**6.111 Lab**	**6.111 Lab**
11:00	38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)
12:00		**Recitation #1**	**Recitation #1**	**Recitation #2**	**Recitation #2**
		Group 1 34-301	Group 5 34-301	Group 1 34-301	Group 5 34-301
1:00		**Recitation #1**	**Recitation #1**	**Recitation #2**	**Recitation #2**
		Group 2 34-302	Group 6 34-302	Group 2 34-302	Group 6 34-302
2:00		**Recitation #1**	**Recitation #1**	**Recitation #2**	**Recitation #2**
		Group 3 34-301	Group 7 34-301	Group 3 34-301	Group 7 34-301
3:00					
4:00					
5:00					
6:00					
7:00	**Lab Hours**	**Recitation #1**	**Recitation #1**	**Recitation #2**	**Recitation #2**
	6.111 Lab	Group 4 34-302	Group 8 34-302	Group 4 34-302	Group 8 34-302
8:00	38-6th floor (go up in 36)	**Lab Hours**	**Lab Hours**	**Lab Hours**	**Lab Hours**
		6.111 Lab	**6.111 Lab**	**6.111 Lab**	**6.111 Lab**
9:00		38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)
10:00					
11:00					

- Finalize strategy
- Finish testing simple programs
- Start programming for final strategy

- Finish construction of robot

- Begin integration of software, mechanics, and sensors.
- *Do video report #2 over the weekend!*

6.270 1997 Schedule: Week 3

Time	Monday 1/20	Tuesday 1/21	Wednesday 1/22	Thursday 1/23	Friday 1/24
10:00		**Lab Hours** **6.111 Lab**	**Lab Hours** **6.111 Lab**	**Lab Hours** **6.111 Lab**	**Lab Hours** **6.111 Lab**
11:00		38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)
12:00	Holiday - Lab Closed	**Recitation #3** Group 1 34-301	**Recitation #3** Group 5 34-301	**Recitation #4** Group 1 34-301	**Recitation #4** Group 5 34-301
1:00		**Recitation #3** Group 2 34-302	**Recitation #3** Group 6 34-302	**Recitation #4** Group 2 34-302	**Recitation #4** Group 6 34-302
2:00		**Recitation #3** Group 3 34-301	**Recitation #3** Group 7 34-301	**Recitation #4** Group 3 34-301	**Recitation #4** Group 7 34-301
3:00					
4:00					
5:00					
6:00					
7:00		**Recitation #3** Group 4 34-302	**Recitation #3** Group 8 34-302	**Recitation #4** Group 4 34-302	**Recitation #4** Group 8 34-302
8:00		**Lab Hours** **6.111 Lab**	**Lab Hours** **6.111 Lab**	**Lab Hours** **6.111 Lab**	**Lab Hours** **6.111 Lab**
9:00		38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)	38-6th floor (go up in 36)
10:00					
11:00					

	• Finish your robot	• Test your robot against others on the table in lab		• *Do video report #3 over the weekend!* • *Mock contest Saturday or Sunday!*

6.270 1997 Schedule: Week 4

Time	Monday 1/27	Tuesday 1/28	Wednesday 1/29	Thursday 1/30	Friday 1/31
10:00	**Lab Hours**	**Lab Hours**	**Pick up Robots**		
	6.111 Lab	**6.111 Lab**	**Robot Check-in**		
11:00	38-6th floor (go up in 36)	38-6th floor (go up in 36)	**Round Two**		
			26-100		
12:00				**Cleanup**	
				6.111 Lab	
1:00				1 person-hour per team required	
2:00					
3:00					
4:00					
5:00					
6:00	**Round 1**	**Impounding**	**Final Contest**		
	All Robots Must	***Get some sleep!***	**26-100**		
7:00	**Be Here!**				
	26-100				
8:00					
9:00					
10:00	***Get some sleep!***				
			PARTY!		
11:00					

- Final bug fixes
- Last chance lab!
- Turn in journals and videos

1.5 Computer Facilities

In this course you will have access to several types of computer facilities. Specifics will be covered in lecture. There is one main facility: the 6th floor lab in building 38. This facility will have playing fields so that you can debug your machine while you edit your code. You can download code to your robots at other Athena machines, but you may need a special connector. We recommend that you do most of your debugging at this facility.

1.5.1 6th Floor Laboratory

This area has some Athena machines, which can be used for your purposes. The machines are located near workbenches so you can fix any hardware problems. This is the *only computer location* where you may solder, build, glue, or cut hardware. All hardware work must be done at the benches and not at the Athena terminals. The terminals will already have their own cables, and you will not need to remove them. *There is to be no eating or drinking in the lab; you may do so in the hallways outside, but food will not be tolerated in the lab itself.*

1.5.2 Athena Clusters

In all Athena locations, there is to be *NO soldering, cutting, or gluing in the cluster. If anyone is caught doing any of these tasks, not only will you be asked to leave the cluster, but you will also be required to return your 6.270 kit and you will be thrown out of the course. There are no exceptions to the rule.* Debris from cutting wire, soldering, or gluing can get lodged inside the keyboards and short something.

1.5.3 Athena Etiquette

If you use other Athena clusters please follow the following rules so that 6.270 is not looked down upon.

- *Noise* Your machines will be quite noisy. If there are lots of people working in the cluster who are trying to get work done, please minimize the machine usage, or move to another cluster.
- *Tidiness* Don't leave your stuff lying around all over the place. Other people have to work and move around.
- *Hardware* Don't solder, glue, or cut any hardware in the clusters. If things go wrong because of this, 6.270 as a whole may suffer, and we may be denied access to Athena machines in the future.

- *Locked Screens* Don't leave your screen locked for long periods of time. Towards the end of the month, we will need every available machine. If you lock your screen for more than 20 minutes, we will log you out.
- *Multiple Machines* Don't log on at multiple machines. Also try to minimize the number of people in your team that are logged on. If everyone logs on, then we will need three times as many machines to download to the robots.

If there are any complaints about 6.270 people working in any of the clusters, we will have to make external Athena clusters off limits. Violation of the rules will not be tolerated and we will be enforcing them strictly.

Chapter 2

RoboRats

Night falls on Vassar Street. You awake from your nest under the train tracks, stretch out your long body, yawn (displaying your four long teeth), and groom yourself for the coming night. Scraping your forepaws' nails through your fur you groom your left rear leg, right rear leg, back (twisting all the way around), right fore arm, left fore arm, behind the ears, over your face. Standing up on your hind legs, your whiskers twitch as you get your bearings and survey the environment. That's right, you're a rat.

You notice that there's some food scattered around your nest. There's a mound of garbage in a straight shot to a neighboring family's nest. There's food near to their nest too, but perhaps you can beat your neighbors to the food near their nest, then gather the ones around your nest later. But rats aren't that stupid, and you quickly realize that your neighbor could do the same. However, the mound of garbage catches your eye. If it were less stable to travel on, the neighbors couldn't get to your food so easily. You ponder this for a moment and then realize:

Rats eat *garbage!*

The contest for the 1997 6.270 contest is **RoboRats**. The scenario is this: you are a rat. You want food. You want lots of food. In fact, you want to have more food than any other rat. Of course, the goal is simple, but there are a few details you need to examine to fully appreciate the simplicity and complexity of the contest.

2.1 The Table

The layout of the contest table is shown in Figure 2.1. The table, overall, is 4 feet by 10 feet 8 inches, and the base color of the table is white. As you can see, the table

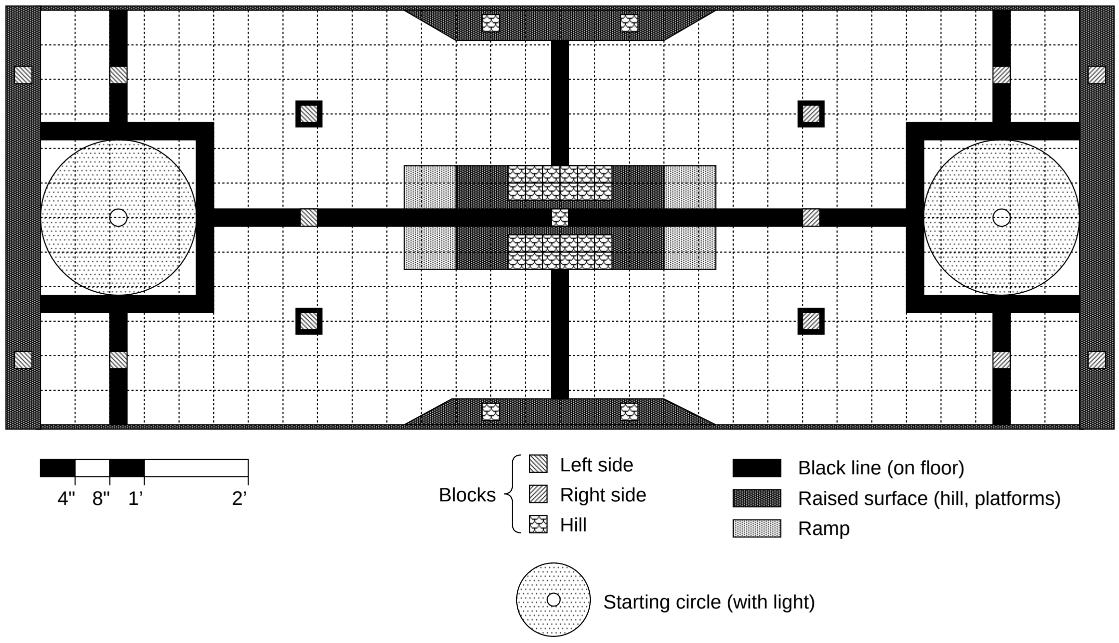


Figure 2.1: The contest table for RoboRats

is symmetric as viewed from each side's starting circle, thus we shall concentrate the description on only one half of the table.

The starting circle is positioned in the center of the table, 9 inches from the back edge of the board. It is 18 inches in diameter with the circle just tangent to the center of the back edge of the table. There is a starting light array, which is about 3 inches in diameter, at the center of the circle.

Forming a box (with the back edge), in which the starting circle is inscribed, are three black lines, each 2 inches wide. A black line down the center of the table connects this box with the matching box on the other end. A food cube will be placed along this line, halfway to the hill (see below). An additional food cube will be placed between this one and each side wall, centered on a 3 inch by 3 inch black square. In addition, a line extends from each of the two other sides to the corresponding edge of the table. Again, food cubes will be placed at the middle of these lines.

Along the back edge of the table, is a raised scoring platform 4 inches deep and 3 inches tall. Two food cubes will initially be placed on the platform.

In the center of the table there is a hill, which is 2 inches tall, 12 inches wide (towards the sides of the table), and 24 inches long. At each end there is a 6 inch long ramp down to the table surface. A food cube will be placed on the center of the hill. A 12 inch by 4 inch section of each side of the hill will be filled with a 6 by 2 group of food cubes.

Along the edges of the table nearest the hill will be additional raised platforms,

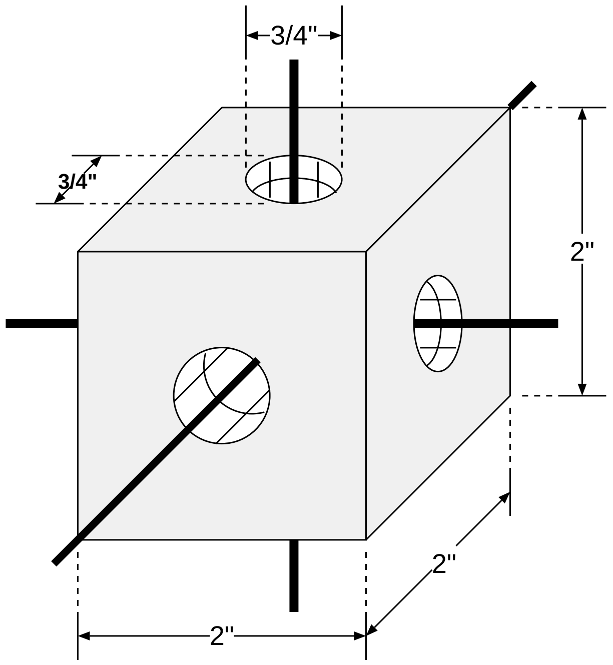


Figure 2.2: A food cube (holes not to scale)

here 3 inches deep (except at the ends, where they slope to join the edge of the table) and half an inch tall. Each of these platforms will be 36 inches long, including 6 inches at each end where they are sloping to the table edge. Two food cubes will be placed on each platform, one towards each end.

Each side has seven food cubes that are considered to be "owned" by the side that they begin on (and they will be colored as such). The cubes in the center area of the board are considered "unowned" and will have a third color.

The food cubes (see Figure 2.2) are made of a foam rubber block 2 inches in each dimension. The cubes also have a three-quarter inch diameter hole drilled through them along each major axis in the center of the appropriate cube faces.

2.2 Scoring

The score of each individual rat is determined by the end state of contest board. Each rat receives points for collecting and storing food cubes. The scoring is summarized in Table 2.1.

A food cube is considered to be in the possession of your robot if, if your robot were moved in the plane of the table, the cube would move with the robot. A food cube is considered to be on the scoring platform if it is *directly* in contact with the platform and can stay on the platform without support. A cube which is in the

Cube Type	**In Your Robot**	**On Your Platform**
Yours	2	4
Center	3	6
Opponent's	4	8

Table 2.1: Scoring summary

possession of your robot as well as being on a platform scores points for being on the platform but not for being in your possession.

Each food cube from the center area which is in your possession at the end of the round is worth three points. Each cube from your side of the table in your possession is worth two points, and each cube from the other side of the table in your possession is worth four points. Cubes on your platform at the end of the round are worth twice as much as if they were in your possession.

Note that the initial table configuration scores 4 points for each side, so ending up with this configuration (without scoring other points) is not sufficient to pass the qualifying round.

2.3 Period of Play

- The contestants will have 30 seconds to place their machines on the field from the time they are called to the playing field.

- The contestants will place the machines on the playing field within the designated starting circles. The starting orientation for the round will be randomly selected by the judges from 4 discrete directions. Both machines will have the same orientation.

- Each machine must have a clearly marked "forward" direction which must point in the direction indicated by the judges at the start of the round.

- Each machine must have a clearly marked "center" point which must be above the starting light at the start of each round.

- The contestants must stand a given distance away from the playing field. Any contestant who touches his or her machine or otherwise affects its performance during the round of play will automatically disqualify his or her robot from the round. All robots must be solely controlled by their onboard computers.

- The beginning of a round is signalled by the judges turning on the starting lights. The lights are located underneath the table in the center of the robots' starting circles and will remain on for the duration of the round.
- The machines must have their own internal clock (software will be provided to do this) that cuts off power to the motors at the end of 60 seconds. Any machine that continues to supply actuator power after 60 seconds will be disqualified.
- The round ends when all machines and other objects on the table come to rest.

2.4 The Competition

- The contest will be an double elimination competition held over two days (three sets of rounds). Machines must qualify for the final night of competition, as follows:
  - *Round 1.* All machines will play in a qualifying round. If a machine demonstrates the ability to score points, it will proceed to the next rounds of the contest, regardless of a win or loss. If a machine fails to do this, modifications may be made, and it will have two chances to run against an inert placebo. If it cannot win against the inert placebo after two tries, it will not qualify for the rest of the contest. Losses from round 1 *do* count and carry through the rest of the contest.
  - *Round 2.* Only qualifying robots will play in the second round. Robots which lost in round 1 will be eliminated from competition should they lose in this round as well. Round 2 may be skipped this year if few enough robots make it past round 1.
  - *Final Contest.* The main competition. Machines will play until they accumulate two losses. Losses against opponents from rounds 1 and 2 still count. Robots with interesting behavior which have not qualified or have been eliminated in rounds 1 and 2 may have an opportunity to perform between actual contest rounds.

    The final round of competition may be conducted in round-robin format, ignoring previous losses, at the discretion of the organizers.
- All rounds will have two robot players. If necessary, a placebo will be used for one player in some rounds.
- In rounds involving a placebo, the contestant's robot must win by at least one point in order to be declared the winner of that round.

- If there is a tie score at the end of a round, the judges may award either a double win or a double loss.

2.5 Infrared Beacon and Light Sources

All robots are required to carry an infrared transmitter. This transmitter acts as a beacon so that robots can locate each other on the playing field. The following rules describe the functionality of the infrared beacon.

- All entries must carry an infrared beacon that is capable of broadcasting infrared (IR) light modulated at either 100 Hertz or 125 Hertz with a 40,000 Hertz carrier (hardware and software is provided to do this).

- Machines failing to meet the infrared transmission specification, or in any way modifying or jamming their transmission frequency during the round of play will be disqualified.

- Judges will assign frequencies for IR emitters to the machines in the beginning of each round. Contestants should set this using the robot's DIP switch 1. If the switch is one, the robot should broadcast 100 Hertz infrared light. If the switch is zero, the robot should broadcast 125 Hertz infrared light. Software will be provided to do this.

- The IR broadcasting beacon must be located at between 17 and 18 inches above the surface of the playing field when mounted on the robot.

- The beacon must be located so that its center is never more than four inches (measured horizontally) from the geometric center of the microprocessor board.

- The beacon may not be deliberately obstructed, or be designed in such a way that "accidental" obstructions are probable. Because of this, robots may not extend farther than 16.5 inches vertically (and should avoid lifting objects above 16.5 inches off above table).

- A polarized light lamp will be placed behind each end of the table. The lamp near the robot transmitting 100 Hertz IR will have a +45 degree (with respect to the vertical) polarization, while the lamp near the robot transmitting 125 Hertz IR will have a −45 degree polarization.

2.6 Structure

- All kits contain exactly the same components, with the exception of some LEGO parts that may be colored differently in different kits.
- Only LEGO parts and connectors may be used as robot structure.
- LEGO pieces may not be joined by adhesive.
- LEGO pieces may not be altered in any way, with the following exceptions:
  1. The grey or green LEGO baseplate may be altered freely.
  2. LEGO pieces may be modified to facilitate the mounting of sensors and actuators.
  3. LEGO pieces may be modified to perform a function directly related to the operation of a sensor. For example, holes may be drilled into a LEGO wheel to help make an optical shaft encoder.
- String may not be used for structural purposes.
- The wooden dowel may be used only as a tower to mount the infrared transmitters and any receivers.
- Any non-LEGO part may be attached to at most five LEGO parts.
- A reasonable amount of cardboard, other paper products, and tape may be used for the purpose of creating optical shields for light sensors. The shield may not obstruct IR transmission. Please ask TAs or organizers if you would like ruling on your particular shield.
- Wire may only be used for electrical, and not structural, purposes.
- Rubber bands may be glued to LEGO wheels or gears to increase the coefficient of friction.
- Only the thin rubber bands may be used to provide stored energy.
- Contestants may not alter the structure of their entry once the contest has begun, but may repair broken components between rounds if time permits.
- Contestants may not alter the program being used by their entry once the contest has begun, except by setting DIP switch 1 as described above.

- At the start of each round, each robot must fit within a one foot cube. The broadcast and detection beacon may extend above one foot. Wires may be compressed, if necessary in order to fit. Entries may, however, expand once the round has begun.
- Entries may not drag wires between two or more structurally separate parts of their robot.
- No lubricants may be used.
- Cable ties may not be used for structural purposes.
- Some parts in the 6.270 kit are considered tools and may not be used on the robot. If there is any question about whether an object is a "kit part" or a "tool part," ask the organizers.
- No parts or substances may be deliberately dumped, deposited, or otherwise left to remain on the playing surface. A machine that appears to have been designed to perform such a function will be disqualified.
- Any machine that appears to be a safety hazard will be disqualified from the competition.
- Machines are not allowed to destroy, or attempt to destroy, their opponent's microprocessor board or infrared beacon.

2.6.1 The $10 Electronics Rule

To encourage creativity, contestants may spend up to $10 of their own funds for the purchase of additional electronic components used in their design. Other than this rule, robots must be designed completely from standard kit parts. The following conditions apply to all non-kit-standard electronic additions:

- The following components, categories of components, or varieties of circuitry are *disallowed:*
  - Batteries of any variety.
  - Motor driver circuitry, including relays, power transistors, or any other replacements or modifications to the standard motor driver circuitry.
  - Microprocessors of any kind.
- Resistors rated less than 1 watt and capacitors valued less than 100 μF may be used freely, without accounting toward the $10 total.

- Contestants who add *any* non-kit parts to their project must turn in a design report that includes: description of the modification, schematic of all added circuitry, and store receipts for parts purchases. This design report must be turned in to the organizers with the robot at impounding time. *Any machines found with added circuitry that has not been documented in this fashion will be disqualified.*

- If a contestant wishes to use an electronic part which has been obtained through other means than retail purchase, an equivalent cost value to the part will be assigned by the organizers. Contestants must obtain this cost estimate *in writing* from the organizers and include it in the design report mentioned above.

  The main reason for this rule is to allow contestants to explore new ways for sensing, and create new sensors.

2.7 Organizers

Contestants may approach the organizers in privacy to consult about possible designs that may be questionable under the rules listed above. These designs will not be divulged to any of the other contestants. You can send e-mail to `6.270-organizers` for any rule clarifications.

Chapter 3

Electronic Assembly Technique

3.1 Electronic Assembly Technique

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

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

By following the instructions and guidelines presented here, you will make your life more enjoyable when debugging time rolls around. A rule of thumb is that a job may take one hour to solder, but if there is a mistake, it takes 3 hours to undo. A little extra care will save you time in the long run.

3.1.1 Soldering Technique

Figure 3.1 shows proper soldering technique. The diagram shows the tip of the soldering iron being inserted into the joint such that it touches both the lead being soldered *and* the surface of the PC board.

Then, solder is applied into the joint, *not* to the iron directly. This way, the solder is melted *by the joint*, and both metal surfaces of the joint (the lead and the PC pad) are heated to the necessary temperature to bond chemically with the solder. The solder will melt into the hole and should fill the hole entirely. Air or gaps in the hole can cause static discharges which may damage some components.

Figure 3.2 shows the typical result of a bad solder joint. This figure shows what happens if the solder is "painted" onto the joint after being applied to the iron directly. The solder has "balled up," refusing to bond with the pad (which did not receive enough heat from the iron).

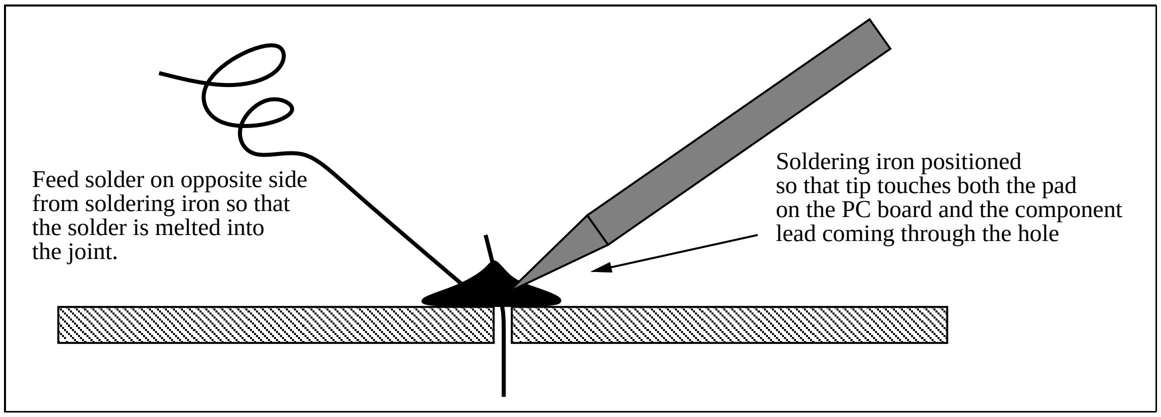

Feed solder on opposite side from soldering iron so that the solder is melted into the joint.
Soldering iron positioned so that tip touches both the pad on the PC board and the component lead coming through the hole

Figure 3.1: Proper Soldering Technique

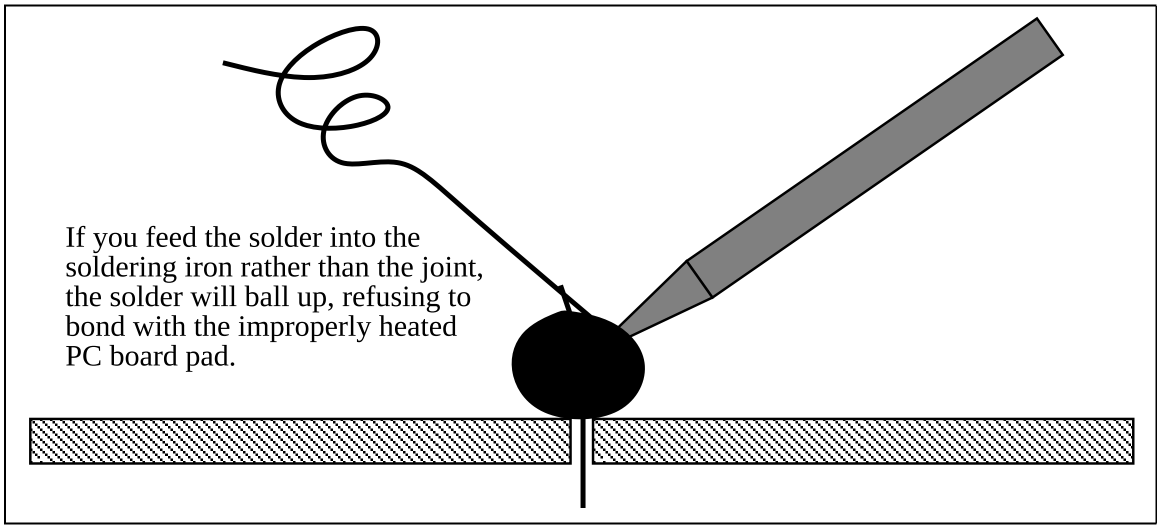

If you feed the solder into the soldering iron rather than the joint, the solder will ball up, refusing to bond with the improperly heated PC board pad.

Figure 3.2: Improper Soldering Technique

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

1. Keep the soldering iron tips away from everything except the point to be soldered. The iron is *hot* and can easily damage parts, cause burns, or even start a fire. Keep the soldering iron in its holder when it is not being held.

2. Make sure that there is a damp sponge available used for cleaning off and tinning the tip. Soldering is basically a chemical process and even a small amount of contaminants can prevent a good joint from being made.

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

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

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

4. The tips of the irons are nickel-plated, so do not file them or the protective plating on the tips will be removed.

5. A *cold solder joint* is a joint where an air bubble or other impurity has entered the joint during cooling. Cold solder joints can be identified by their dull and mottled finish. The solder does not flow and wrap around the terminal like it should.

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

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

   Diodes, ICs, and transistors are particularly sensitive to heat damage.

7. It is good practice to tin stranded wire before soldering to other components. To tin the wire, first strip the insulation and twist the strands. Apply heat with the soldering iron and let the solder flow between the strands.

8. After a component has been soldered, clip the component's leads (wires coming out of the component) away from the printed circuit board. Leave about a $\frac{1}{8}''$ of the lead sticking out of the board. When clipping the leads, *face the board and the lead down into a garbage bag or into your hand. Leads tend to shoot off at high speeds, and can fly into someone's eye.*

3.1.2 Desoldering Technique

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

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

Two methods of desoldering are most common: desoldering pumps and desoldering wick. Both of these are available from the 6.270 staff.

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

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

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

3.1.3 Component Mounting

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

Components come in two standard packaging types: axial and radial. Axial mounts, shown in Figure 3.3 and Figure 3.4, generally fit right into the holes in the PC board. The capacitors and LEDs in the 6.270 kit are all radial components. The leads of axial components must be bent or modified to mount the component. The resistors, diodes and inductor are all axial components.

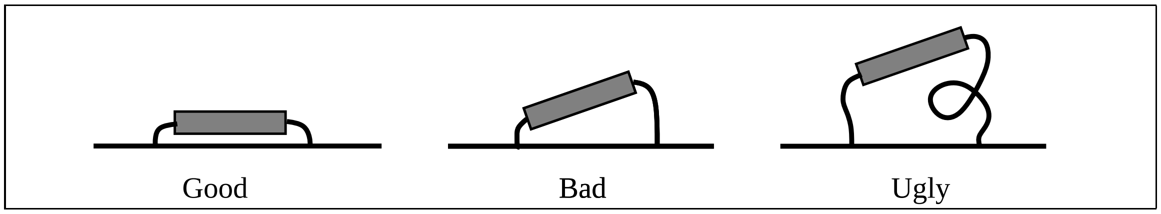


Figure 3.3: Flat Component Mounting

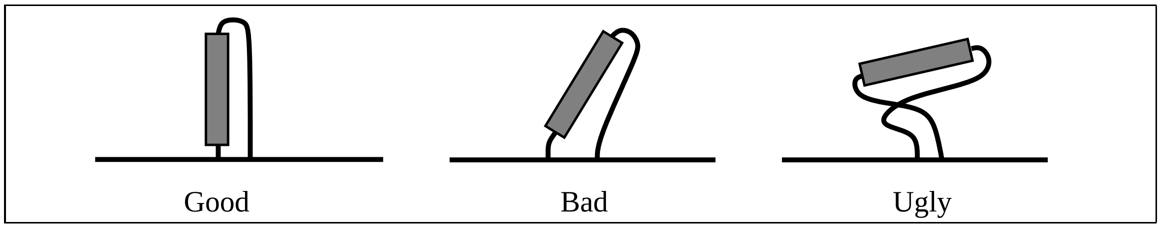


Figure 3.4: Upright Component Mounting

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

See Figures 3.3 and 3.4 for clarification.

3.1.4 Component Types, Polarity, and Markings

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

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

The following components are always polarized:

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

Capacitors are an interesting case, because some are polarized while others are not. Fortunately, there is a rule: large capacitors (values 1 μF and greater) are generally polarized, while smaller ones are not.

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

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

The sections on resistors, resistor packs, and capacitors explain how to read their markings. Other devices, such as diodes, transistors and integrated circuits, are referenced by their part numbers, which are printed on the device packages.

Resistors

Most resistors are small cylindrical devices with color-coded bands indicating their value. Almost all of the resistors in the 6.270 kit are rated for $\frac{1}{8}$ watt, which is a very low power rating. Hence they are quite tiny devices.

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

The largest resistors—in terms of wattage, not resistive value—simply have their value printed on them. For example, the two large, 5 watt, 7.5Ω resistors in the 6.270 kit are marked in this fashion.

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

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

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

Figure 3.5 shows the meaning of the colors used in reading resistors.

A few examples should make this clear.

- *brown, black, red*: 1,000Ω, or 1kΩ.
- *yellow, violet, orange*: 47,000Ω, or 47kΩ.
- *brown, black, orange*: 10,000Ω, or 10kΩ.

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

Figure 3.5: Resistor Color Code Table

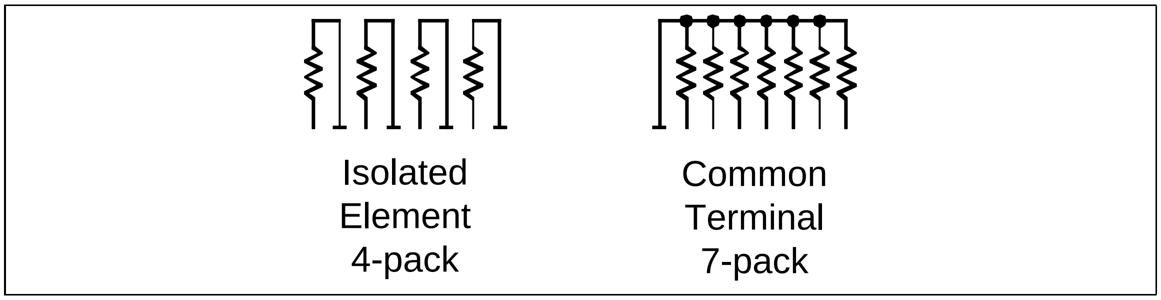


Figure 3.6: Resistor Pack Internal Wiring

Resistor Packs

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

- **Isolated Element.** Discrete resistors; usually three, four, or five per package. These are not polarized.
- **Common Terminal.** Resistors with one pin tied together and the other pin free. Any number from three to nine resistors per package. These are polarized components. The common ground should be marked by a dot or bar by one end of the package.

Figure 3.6 illustrates the internal wiring of an 8-pin resistor pack of each style.

Common terminal (polarized) resistor packs are usually marked with an “E” before their value, i.e. “E47KΩ” designates a 47KΩcommon terminal resistor pack. Isolated element resistor packs are marked with a “V,” as in “V1KΩ.”

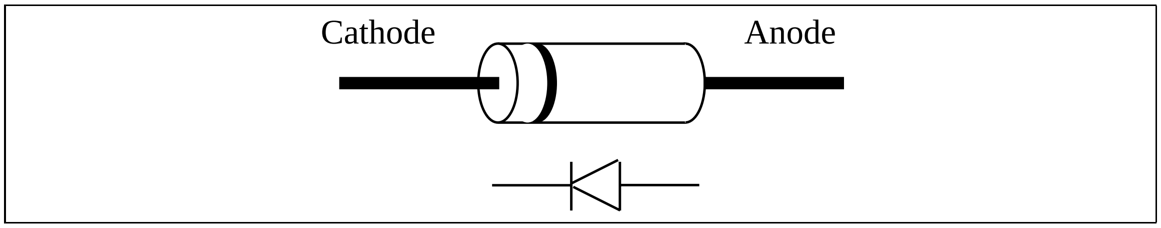


Figure 3.7: Typical Diode Package

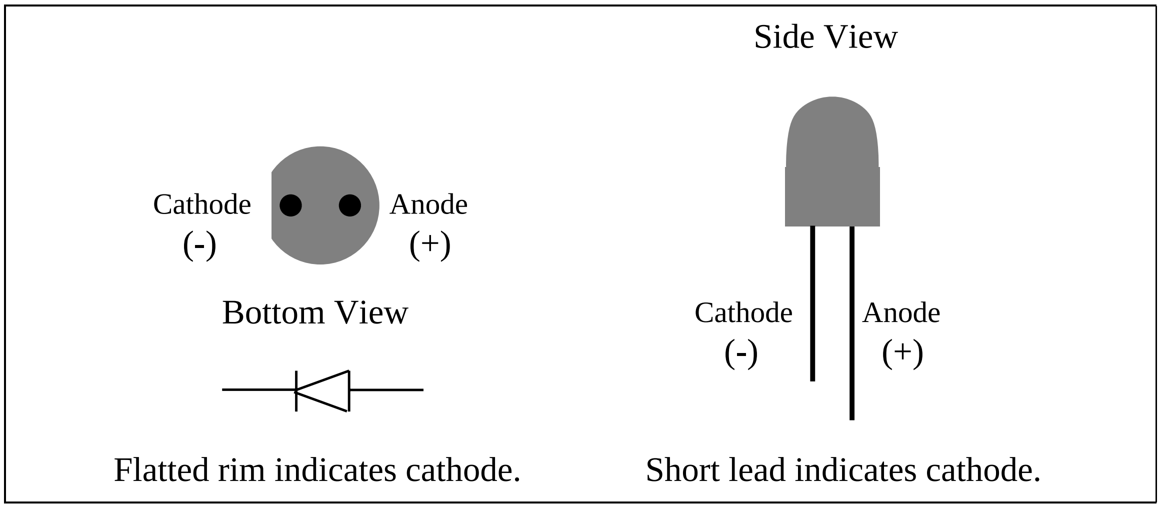


Figure 3.8: Identifying LED Leads

Diodes

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

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

Figure 3.7 shows a typical diode package.

LEDs

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

Figure 3.8 shows a typical LED package.

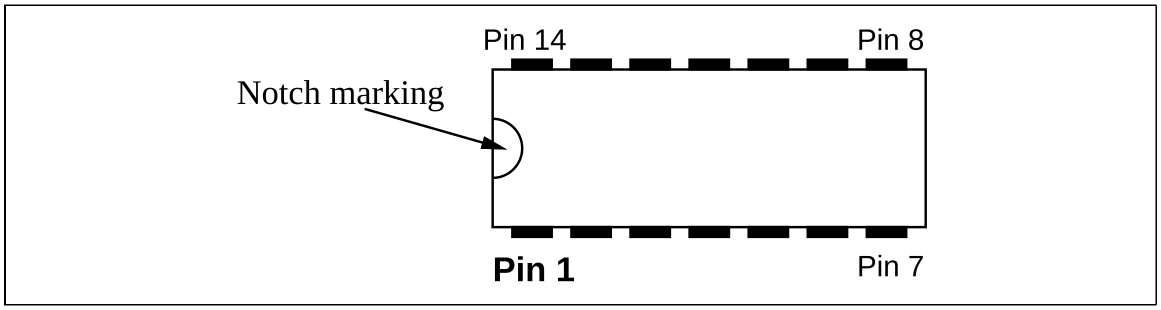


Figure 3.9: Top View of 14-pin DIP

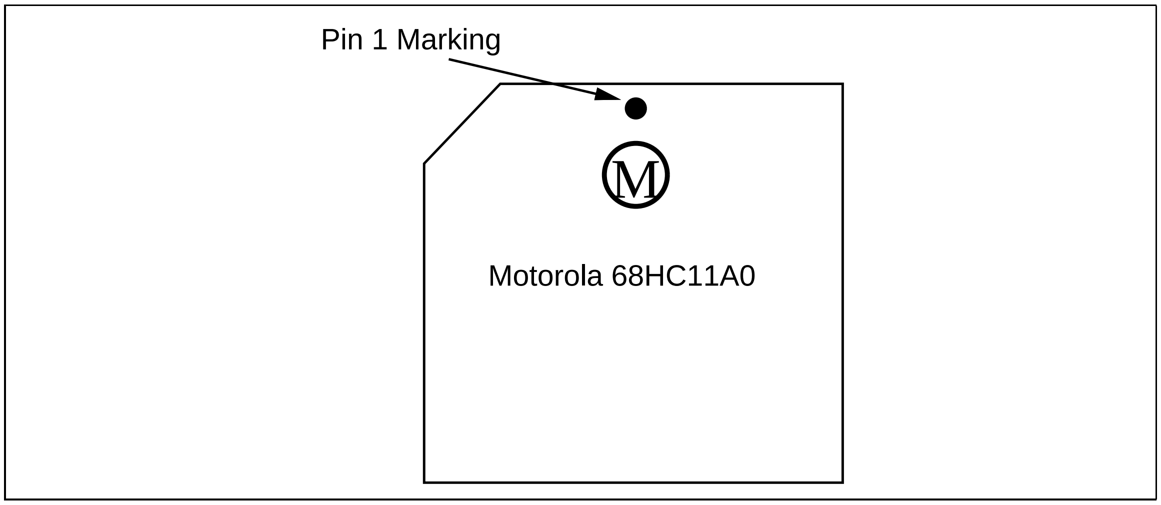


Figure 3.10: Top View of 52-pin PLCC

Integrated Circuits

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

In both types, a marking on the component package signifies "pin 1" of the component's circuit. This marking may be a small dot, notch, or ridge in the package. After pin 1 is identified, pin numbering proceeds sequentially in a counter-clockwise fashion around the chip package.

Figure 3.9 shows the typical marking on a DIP package. Figure 3.10 is a drawing of the PLCC package.

DIP Sockets

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

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

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

PLCC Sockets

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

When assembling the 6.270 board, a marking printed onto the board indicates the correct orientation of the PLCC socket. There are smaller corner holes that will help you orient the socket. Place the socket on the board and double check the polarity before soldering.

Capacitors

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

- **Monolithic.** These are very small-sized capacitors that are about the size and shape of the head of a match from a matchbook. They are excellent for use when small values are needed (0.1 μF and less). They are inexpensive and a fairly new capacitor technology. Monolithic capacitors are always non-polarized.

- **Electrolytic.** These capacitors look like miniature tin cans with a plastic wrapper. They are good for large values (1.0 μF or greater). They become bulky as the values increase, but they are the most inexpensive for large capacitances.

  Electrolytics can have extremely large values (1000 μF and up). They are usually polarized except for special cases; all the electrolytics in the 6.270 kit are polarized.

- **Tantalum.** These capacitors are compact, bulb-shaped units. They are excellent for larger values (1.0 μF or greater), as they are smaller and more reliable than electrolytic. Unfortunately they are decidedly more expensive. Tantalum capacitors are always polarized.

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

On the 6.270 boards, all polarized capacitor placements are marked with a plus symbol (+) and a minus symbol (−). The pads on the boards are also marked differently. The negative lead (−) goes through a square hole and the positive lead (+) goes through the round hole.

The capacitors themselves are sometimes are obviously marked and sometimes are not. One or both of the positive or negative leads may be marked, using (+) and (−) symbols. In this case, install the lead marked (+) in the hole marked (+).

Some capacitors may not be marked with (+) and (−) symbols. In this case, one lead will be marked with a dot or with a vertical bar. This lead will be the positive (+) lead. This should not be confused with the stripe with several minus signs which runs down one side of many electrolytics.

Polarized capacitors that are mounted backwards won't work. In fact, they often overheat and explode. Please take care to mount them correctly. If you are not sure about the polarity of a capacitor, please ask a TA.

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

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

Capacitors smaller than 1μF have their values printed in picofarads (pF). There are 1,000,000 pF in one μF.

Capacitor values are similar to resistor values in that there are two digits of mantissa followed by one digit of exponent. Hence the value "472" indicates 47×10^2 picofarads, which is 4700 picofarads or 0.0047 μF.

Inductors

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

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

Inductors are not polarized.

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

Figure 3.11: Summary of Polarization Effects

Transistors

There are two types of transistors used in the 6.270 kit. Both are three-wire devices. The larger the transistor is used for larger currents.

Transistors are polarized devices.

The table shown in Figure 3.11 summarizes this discussion of polarity issues.

Chapter 4

Assembly Manual

This chapter presents an introduction to electronic assembly followed by step-by-step instructions for assembling the 6.270 hardware. The instructions assume no prior background in electronics. The order of the assembly should help you get into the soldering mode, and will give you practice at soldering some of the bulky items before soldering the delicate devices.

This chapter was revised by Matt Domsch '94 in his "Advanced Undergraduate Project" to correct inaccuracies and simplify some assembly steps.

Instructions are provided for the following boards and devices:[1]

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

If your team has more than one soldering iron, you can assemble some of the boards in parallel.

The reasons for having the teams build the boards are two-fold. First it gives you an opportunity to learn about the components and how to solder. The second reason

[1]Note that this year the Microprocessor Board and Expansion Board are pre-assembled, and the Motor Switching Board is no longer provided in the 6.270 kit.

is to get you familiar with the boards and how they operate. As you read through the assembly sections, a brief description of the functionality will be given so that you become familiar with the system.

4.1 The Battery System

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

These batteries should power the microprocessor board for about thirty hours of operation before needing to be replaced. The board should not be left on inadvertently because the batteries will be drained. When the board is off, you should not remove the AA batteries, or else the RAM will be erased. These batteries will last longer if the motor batteries are also plugged into the board.

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

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

These lead-acid cells are extremely powerful devices. Car batteries are constructed of similar lead-acid technology. These batteries can be used to start a motorcycle, or maybe even a car. *When handling the batteries, be extremely careful not to short the (+) and (−) terminals of the battery together.* A huge surge of current will flow, melting the wire and causing burns. In extreme cases, batteries can explode and cause serious injury. These batteries however, have been reinforced very well and should not explode, but will burn you if they are shorted.

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

The following instructions explain how to build the battery recharger and how to wire the Hawker cells into power-packs. Note that contest rules prohibit using the Hawker cells in any configuration other than what is presented here. This means that you cannot alter the electrical configuration, but you can modify the physical configuration. You can also use the battery casings that were donated by Hawker Batteries to hold your batteries. It is important that you take care in assembling the battery packs because the batteries are charged in their initial state. You should assemble the batteries first so that there are no inadvertent shorts that can cause the batteries to overheat.

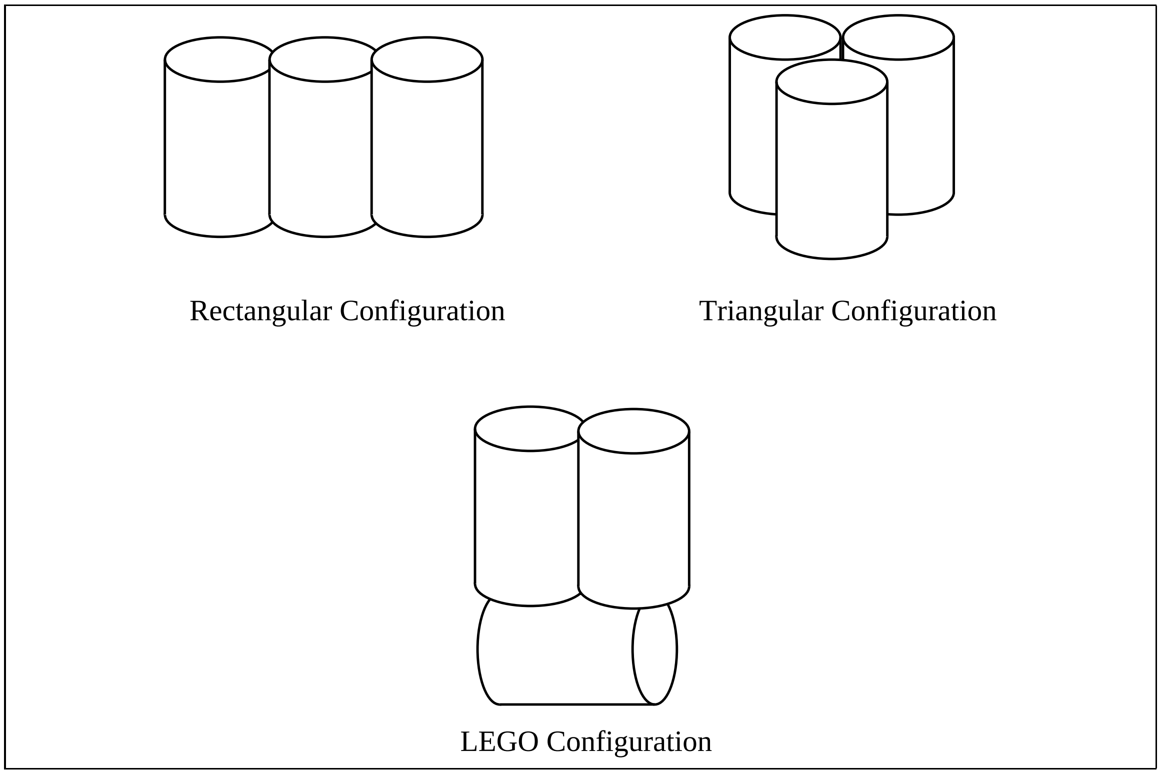


Figure 4.1: Three Battery Pack Configurations

4.1.1 Battery Pack Construction

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

The 6.270 kit includes 6 Hawker cells, enough to make two battery packs. It is recommended that contest robots be designed in a fashion that facilitates battery pack swapping. One battery pack can be used to operate the robot while the other is being charged (charging takes about 10 hours). There are reasons for which you may want your design to have batteries in different locations.

Two obvious alternatives for battery pack construction are depicted in Figure 4.1: a rectangular configuration and a triangular one. Another possibility is a LEGO configuration shown in Figure 4.1 which can be mounted on a $6x8$ LEGO plate. Other configurations may be explored.

Wiring the Battery Cable

Figure 4.2 illustrates how to wire the battery plug and cable assembly.

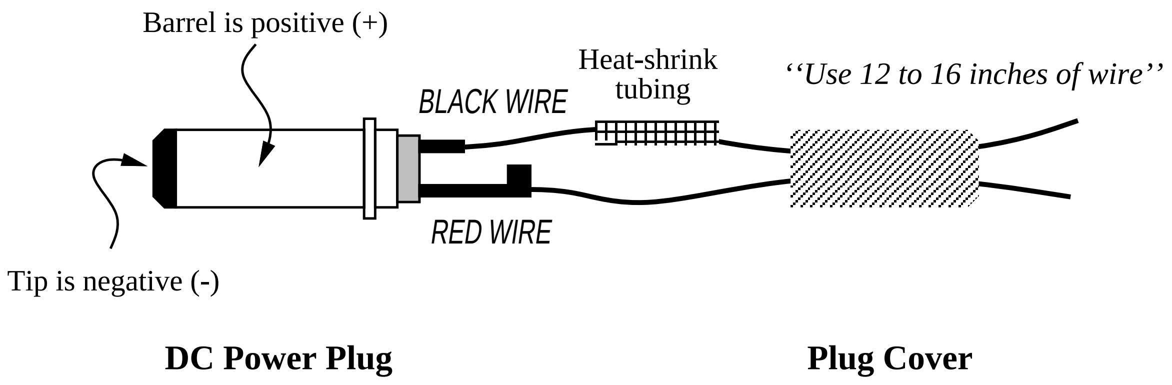


Figure 4.2: Battery Plug and Cable Wiring Diagram

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

○ $\frac{1}{16}''$ Heat shrink tubing is used on the shorter terminal of the DC power plug. The tubing acts as an insulator to minimize the likelihood of an electrical short at the plug terminals. *It is essential that this wiring be performed carefully because a short in the power plug will short out the battery terminals and create a serious hazard.*

○ Proper polarity is important. The use of red wire to signify the (+) terminal and black wire to signify the (−) terminal is an international standard. Mount the black wire to the short terminal and the red wire to the long terminal.

○ After soldering, slide the heat shrink tubing down over the short terminal and shrink it. Also, crimp the prongs of long terminal onto the red wire as a stress relief.

○ Screw the plug cover onto the plug.

○ *Before installing the cable onto a battery pack, use an ohmmeter to make absolutely sure that the cable is not shorted.* The cable should measure open circuit or infinite resistance. If a short is placed across the terminals of lead-acid batteries (like the Hawker cells), a huge surge of current will flow, melting the wire causing the short and possibly causing the battery to explode.

Constructing the Battery Pack

Wire the 3-cell pack to the battery cable as indicated in Figure 4.3. Use the red and black wire to make the two jumpers between the cells (color of these jumpers does

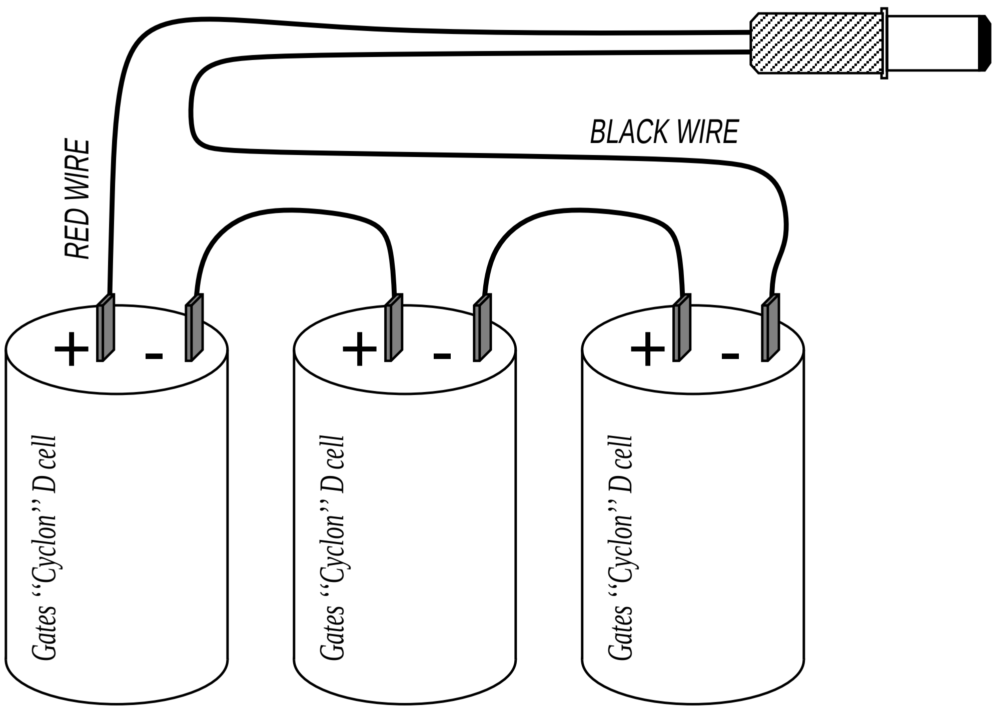


Figure 4.3: Battery Pack Wiring Diagram

not matter). *Make sure to get polarities correct. Incorrect wiring will cause the wire to get hot and catch on fire.*

After the battery pack is wired, an overall configuration (as suggested in Figure 4.1 can be selected. The battery pack may be held in the desired configuration using a variety of materials, including rubber bands, cable ties, hot glue, and/or electrical tape. The terminal of the batteries can be bent and hot glued so they do not inadvertently get shorted. Do not put too much hot glue, or else the battery will not be able to breathe. There must be a pathway for the gases in the battery to escape or else too much pressure builds up in the casing and may cause an explosion.

4.1.2 The Battery Charger

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

- **Normal charge.** Marked SLOW on the charger board, this is the normal charge position. A battery pack will recharge completely in about ten to fourteen hours. When the batteries become slightly warm they are fully charged. In this configuration, the power is supplied to the battery through the larger valued resistor (15Ω). The batteries are charged with a constant current of about 400 milliamps.

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

- **Fast charge.** Marked FAST on the charger board, this position will recharge a battery pack in five to seven hours. The batteries are being charged at a constant current of about 800 milliamps.

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

  Is is better to charge the batteries at the slower rate if possible. When high currents are being passed through the batteries, they tend to heat up. The batteries do not accept charge very well at higher temperatures.

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

Assembly Instructions

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

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

Figure 4.4 shows component placement on the battery charger board.

1–☐ **Get the battery charger board and determine which is the component side.** The component side is marked with the placement guidelines in white. You should always solder on the solder side of the board, which does not have white writing.

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

2–☐ **Resistor Pack.**

Install **RP7**, 1.2kΩ×4, 8 pins, *Blue single in-line pin.* *The board is labelled for 1kΩ; this marking is incorrect.* This resistor pack consists of four isolated resistors so orientation is not significant.

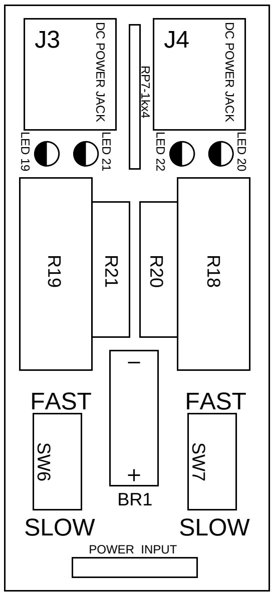


Figure 4.4: Battery Charger Component Placement

3–☐ LEDs.

These LEDs are the LEDs which are in the bag with the battery charger board. There are additional red LEDs in the kit – do not mix these two up. The other LEDs are not low power LEDs. You need to use the low powered LEDs in this section. Mount LEDs so that the short lead is inserted in the shaded half of the placement marking. Make sure to push the LEDs all the way through and mount them as close to the board as possible.

○ **LED19**–red

○ **LED20**–red

○ **LED21**–green

○ **LED22**–green

4–☐ DC Power Jacks.

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

5–☐ Power Resistors.

○ **R18**–7.5Ω, 5 watts. This resistor is marked by a big white square outline on the component side.

◯ **R19**–7.5Ω, 5 watts. This resistor is marked by a big white square outline on the component side.

◯ **R20**–15Ω, 2 watts, *brown, green, black*

◯ **R21**–15Ω, 2 watts, *brown, green, black*

6–☐ Slide Switches.

◯ **SW6**–miniature SPDT slide switch

◯ **SW7**–miniature SPDT slide switch

7–☐ Bridge Rectifier.

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

8–☐ Power Cord.

Get the large DC power adapter. Clip off any connectors on the DC side of the adapter. Measure the voltage across the wires and make sure that the voltage is above 12VDC when the adapter is plugged into the wall. Strip $\frac{1}{4}$" of insulation from power wires. Insert stripped wires into holes marked POWER INPUT from component side of board; solder from solder side.

The polarity of the power connection is not significant.

9–☐ Check Out

You can test the battery charger in the lab using a voltmeter. Plug the charger into the wall socket and measure the voltage between positive and negative of the bridge rectifier. Make sure that the polarity is correct. Plug the battery pack into each of the jacks, and make sure the red LED is on when the switch is on FAST and the green LED is on when the switch is on SLOW. You should check it out with a TA if you are unsure.

10–☐ After Checkout.

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

4.2 The Motor Switch Board

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

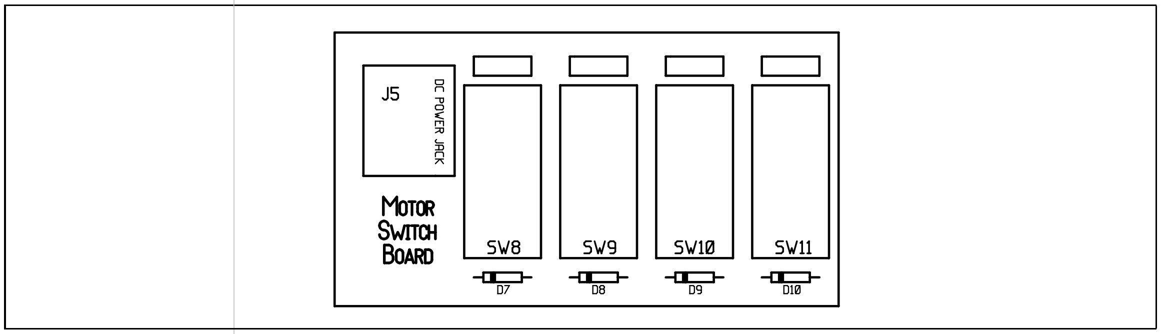


Figure 4.5: Motor Switch Board Component Placement

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

Motors driven from the Expansion Board will operate at even less power than those driven by the Microprocessor Board. The motors are driven through a diode which provides a .6 volt drop, but the motor drivers may drop up to 1.2 volts and reduce your motor output.

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

The simplicity of the Motor Switch Board will give the inexperienced solderer an opportunity to get some practice before committing to the bigger boards. A second member in the group should begin to assemble a motor to test the Motor Switch Board.

4.2.1 Assembly Instructions

Figure 4.5 provides a reference to parts mounting on the Motor Switch Board.

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

2–☐ Diodes.

These diodes have black epoxy bodies. Polarity matters: Install the diodes with the banded end as marked on the circuit board. These diodes can be mounted in the horizontal position.

○ **D7**–1N4001

- ◯ **D8**–1N4001
- ◯ **D9**–1N4001
- ◯ **D10**–1N4001

3–☐ DC Power Jack

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

4–☐ Switches.

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

5–☐ Female Socket Headers.

To cut socket headers to length, repeatedly score between two pins using an exacto knife. Score on both sides of one division and then snap or cut the strip carefully with the diagonal cutters in two. *Do not try to snap header pieces before they have been sufficiently scored,* or they will break, destroying one or both of the end pieces in question. These are often difficult to cut without some past experience, so don't hesitate to ask a TA if you have any trouble. You only have a limited number of the female header so do not waste them because they are very expensive and we do not have very many extra.

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

4.3 The Expansion Board

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

- analog multiplexers to provide up to eight times more analog inputs;
- four DIP configuration switches;
- a user-adjustable "frob knob" for analog input;
- drivers for two additional motors;

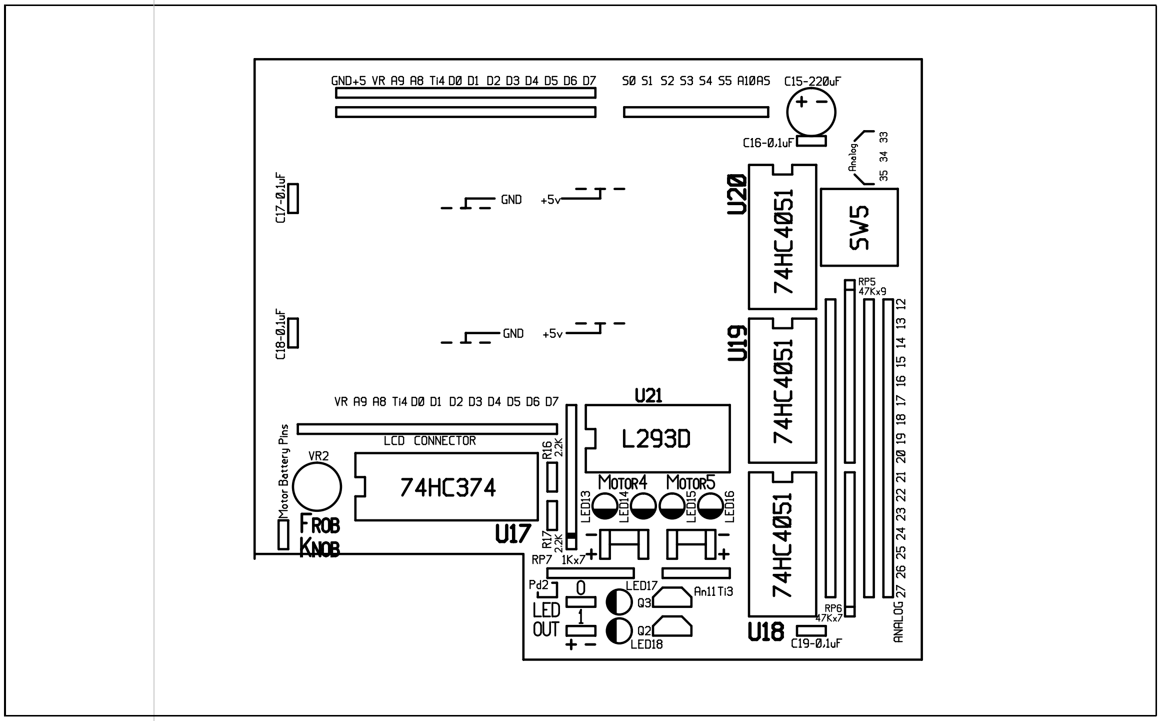


Figure 4.6: Expansion Board Component Placement

- drivers for two LED/lamp circuits;
- a general purpose prototyping construction area.

Figure 4.6 is a component placement guide for the Expansion Board.

Assembling the expansion board is the next step before taking on the task of soldering the microprocessor board. It is important that good soldering technique be developed before moving on to the microprocessor board.

4.3.1 Assembling the Expansion Board

1–☐ Get the 6.270 expansion board, and determine which is the component side.

The side that has white component markings is the component side. The reverse is the solder side.

2–☐ Check for Power-Ground shorts. Before placing any components, take a multimeter and test for a short between power and ground. Power and ground can be found on the prototype area as marked on the white silkscreen. The resistance between power and ground should be infinite. If a resistance of 0 ohms is found, replace the board.

3–☐ Resistor Pack.

These resistor pack are all polarized resistor packs where the common terminal end is marked with a band. On the 6.270 board, find a *square metal pad* at one end of the area that each resistor pack will mount. *Insert the resistor pack such that the marked end mounts in the shaded hole.*

The "catty-cornering" technique of soldering the two end terminals first is helpful here. Solder one end of the terminals before soldering the remaining pins. Adjust the component such that it is straight and the pins are oriented properly, and then solder the other end of the resistor pack. After the resistor pack is straight and aligned, solder the middle pins. This will allow you to align the resistor pack and make it straight before all the pins are fastened.

○ **RP5**, 47kΩ×9, 10 pins, polarized, *marked "E47KΩ."* The marked end of the resistor pack goes through the shaded square hole.

○ **RP6**, 47kΩ×7, 8 pins, polarized, *marked "E47KΩ."* Mount on the component side so that the marked end of the resistor pack goes through the shaded square hole. (**Note**: do not install this part if you wish to use the phototransistors since they require different values.)

RP5 and RP6 are pull up resistors for the analog inputs. These are used so that the inputs to the analog ports don't float. They are also used as part of a resistor divider in some of the sensors used.

○ **RP7**, 1kΩ×7, 8 pins, polarized, *marked "E1KΩ."* Mount so that marked end of resistor pack goes in square hole on board.

4–☐ IC Sockets.

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

The catty-cornering technique should help here too. After inserting a DIP into the board, solder its two opposite-corner pins first. This will hold the chip in place. Make sure it is pressed down as far as it can go; then solder the other pins. You may need to apply heat to the corner pins to press the socket down if it is not flush with the board.

○ **DIP21**–16 pins. The socket used for this DIP is different from the other sockets. The pins are more rounded, and the sockets are circular holes.

The other sockets have more rectangular socket holes. These are gold plated sockets and are used for a better conduction pathway to the motor outputs.

- ◯ **DIP17**–20 pins
- ◯ **DIP18**–16 pins
- ◯ **DIP19**–16 pins
- ◯ **DIP20**–16 pins

5–☐ LEDs.

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

- ◯ **LED13**–red, indicates motor 4 in reverse direction.
- ◯ **LED14**–green, indicates motor 4 in forward direction.
- ◯ **LED15**–red, indicates motor 5 in reverse direction.
- ◯ **LED16**–green, indicates motor 5 in forward direction.
- ◯ **LED17**–red, indicates LED out 1 is on.
- ◯ **LED18**–red, indicates LED out 0 is on.

6–☐ Trimpot.

Install **VR2**, 50kΩ, this is the FROB KNOB. The three pins of the potentiometer are polarized. They should look like they form a triangle. The triangle should correspond to the triangle of pins on the board.

7–☐ Resistors.

These resistors must be mounted in the upright position due to the tight spacing.

- ◯ **R16**, 2.2kΩ, *red, red, red.*
- ◯ **R17**, 2.2kΩ, *red, red, red.*

8–☐ Capacitors.

- ◯ **C16**–0.1μF, non-polarized
- ◯ **C17**–0.1μF, non-polarized
- ◯ **C18**–0.1μF, non-polarized
- ◯ **C19**–0.1μF, non-polarized

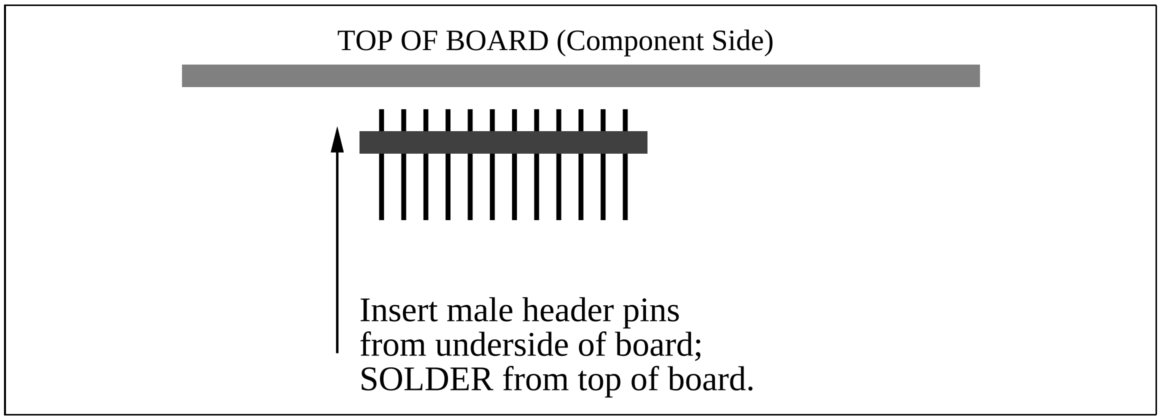

TOP OF BOARD (Component Side)
Insert male header pins from underside of board; SOLDER from top of board.

Figure 4.7: Mounting Method for Male Header Pins

9–☐ Male Header Pins.

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

When mounting these pins, insert upward from the underside of the board so that the maximal pin lengths protrude downward (see Figure 4.7). These pins are then soldered from the top, component side of the board.

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

For the following instructions, refer to Figure 4.8 for pin placement.

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

10–☐ Transistors.

Install transistors **Q2** and **Q3** (type MPS2222A) where indicated on the Expansion Board. The transistors mount so that their flat edge is above the flat edge of the placement marking.

These transistors look identical to the DS1233 Econo Reset chip. Be sure you have the MPS2222A transistors and not the DS1233, or your board will not work.

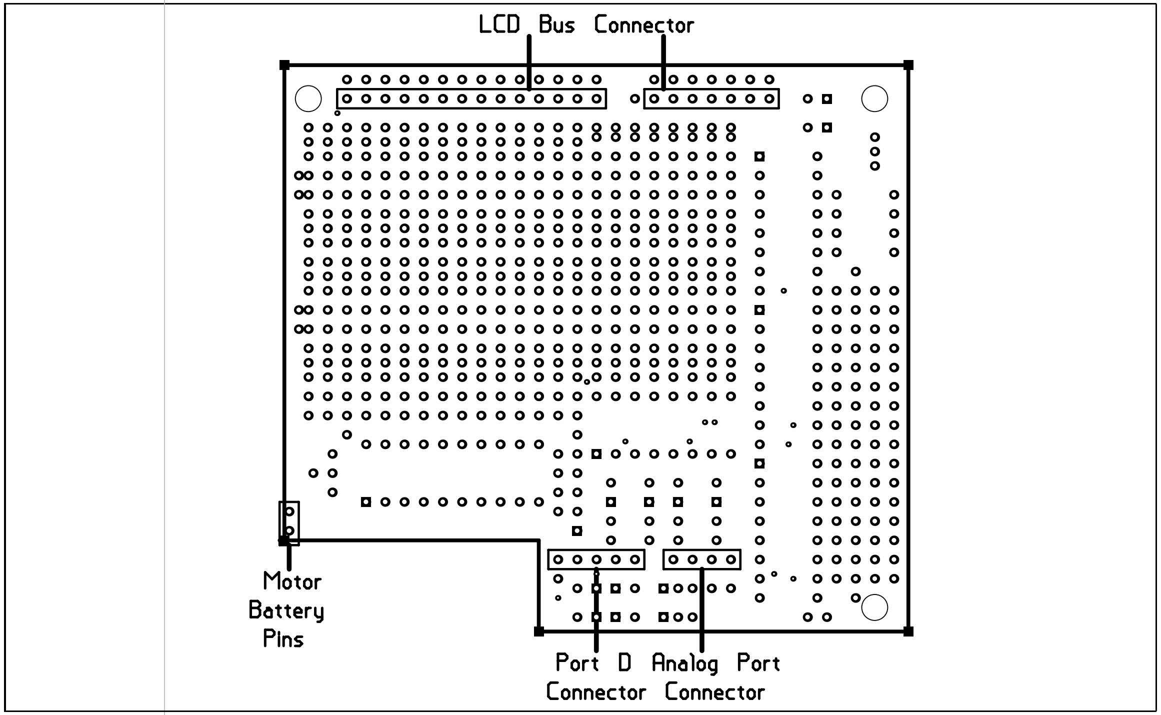


Figure 4.8: Expansion Board Male Header Pin Placement

11–☐ C15–220μF, *polarized.* Be sure to mount with correct polarity.

Leave some spacing between the board and the capacitor so the capacitor can be bent over. Before soldering the capacitor bend it sideways so it points to the left side of the board, and then solder.

12–☐ Female socket headers.

Refer to Figure 4.9 to be sure of placement of these parts.

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

◯ Cut three 16-long strips. Before installing the Female header, make sure that the resistor packs RP5 and RP6 are correctly installed. Once the female header is in place, it is nearly impossible to replace the resistor packs. Install the **Analog Input Port**. A male header strip can be used

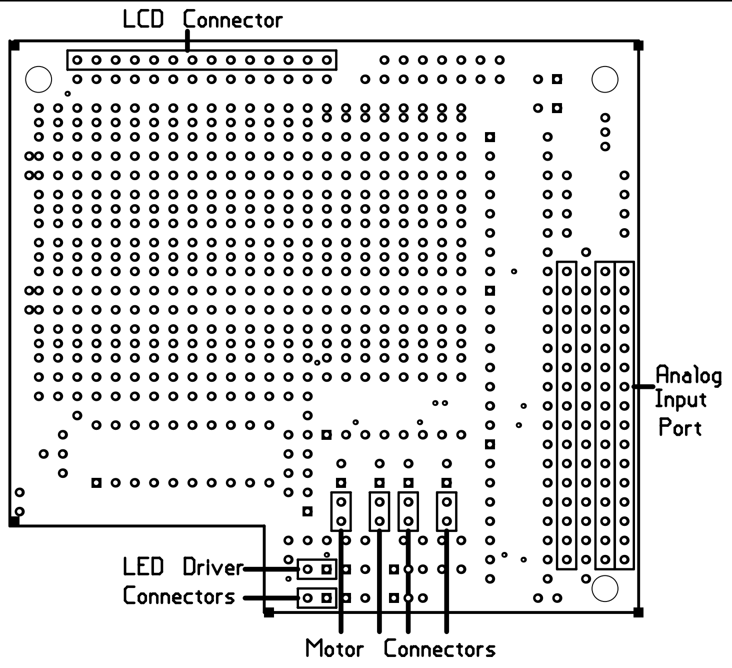


Figure 4.9: Expansion Board Female Header Mounting

at each end to align the vertical and horizontal placement of the female header. Solder.

○ Cut one 14-long strip. Install the **LCD Connector**. *Note*: The correct position for this header is *not* the location marked *LCD CONNECTOR* on the board. The correct position is indicated properly in Figure 4.9, at the top edge of the board.

○ Cut six 2-long strips. Install **Motor Connectors** and **LED Driver Connectors**.

13–☐ DIP Switches.

Install **SW5**, 4-position DIP switch. Install so that numbers are on the outside edge of the board.

4.3.2 Testing the Expansion Board

As with the Microprocessor Board, run through the following checklist before mounting the chips into the Expansion Board.

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

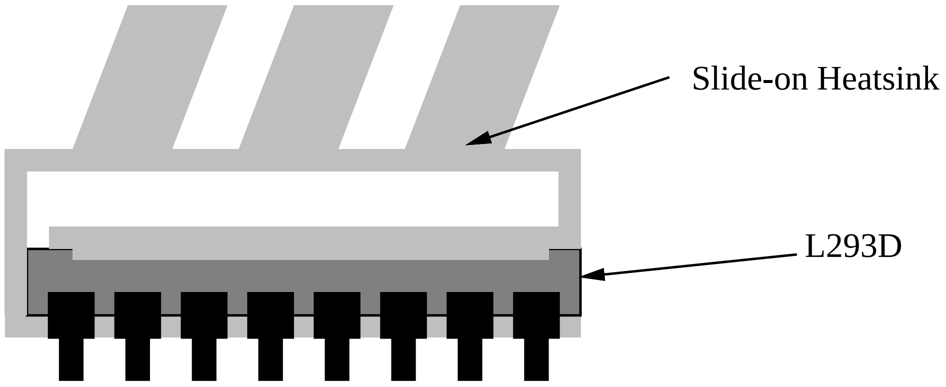


Figure 4.10: Heat Sink on Motor Chip

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

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

3–☐ Install ICs in the board, observing correct polarity:

- ○ **U17**–74HC374. This is the output latch used for controlling motor 4 and 5, and the two LED outputs. There are two additional outputs that can be jumpered to.
- ○ **U18**–74HC4051. This is the analog multiplexer that controls analog outputs 20-27. It feeds the signal to analog 9.
- ○ **U19**–74HC4051. This is the analog multiplexer that controls analog outputs 12-19. It feed the signal to analog 8.
- ○ **U20**–74HC4051. This is the analog multiplexer that controls the Frob Knob, the DIP switches, and analog 33-35. It feeds the signal to analog 10.
- ○ **U21**–L293D. Slide gold heat sink onto chip before installing in socket. The diagram for the mounting is shown in Figure 4.10. The gold heat sink slides right onto the chip.

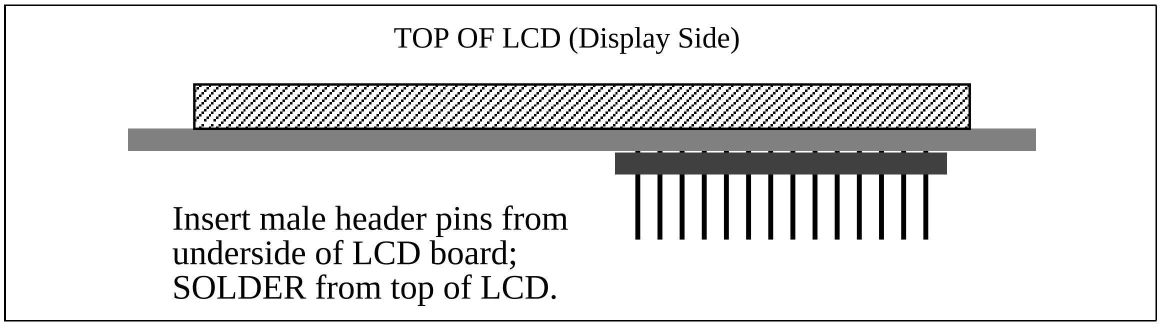

TOP OF LCD (Display Side)
Insert male header pins from underside of LCD board; SOLDER from top of LCD.

Figure 4.11: LCD Connector Mounting

4.4 The LCD Display

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

The display needs to have a 14-pin male header soldered to its interface. Figure 4.11 shows how these pins should be installed, in a similar fashion to the pins protruding from the Expansion Board.

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

4.5 The Microprocessor Board

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

4.5.1 Assembling the Microprocessor Board

Figure 4.12 illustrates the component placement on the microprocessor board.

In addition to checking off the boxes and circles after completion of a component, it may be helpful to fill in the component location in Figure 4.12.

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

1–☐ Get the 6.270 Microprocessor Board, and determine which is the

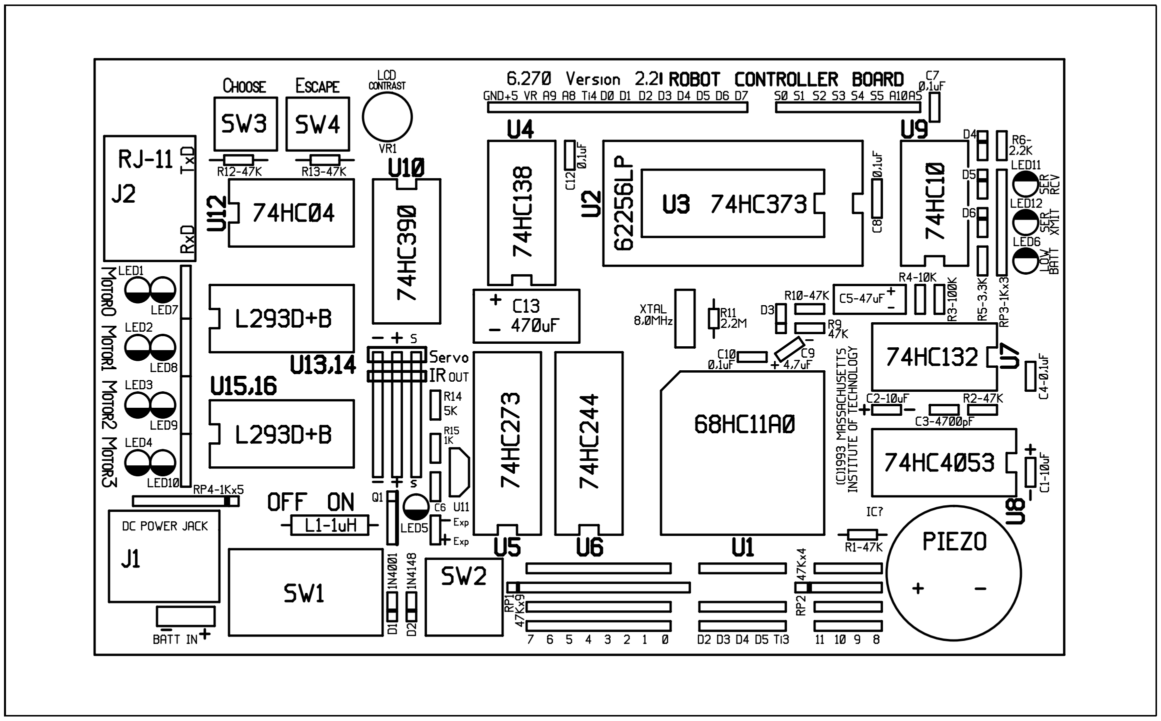


Figure 4.12: 6.270 Microprocessor Board Component Placement

"component side." The Microprocessor Board is the largest of the 6.270 boards.

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

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

2–☐ **Check for Power-Ground shorts.** Before placing any components, take a multimeter and test for a short between power and ground. Power and ground can be found on the LCD connector ports at the top of the board as marked on the white silkscreen. The resistance between power and ground should be infinite. If a resistance of 0 ohms is found, replace the board.

3–☐ **Flat Resistors**

Begin by installing the resistors that lie flat along the board. Try to get the body of the resistor very close to the board.

4–☐ **R1**–47kΩ, *yellow, violet, orange,* flat mounting, lower right corner.

5–☐ **R11**–2.2MΩ, *red, red, green,* flat mounting, next to oscillator.

6–☐ **R12**–47kΩ, *yellow, violet, orange,* flat mounting, top left corner under SW3.

7–☐ **R13**–47kΩ, *yellow, violet, orange,* flat mounting, top left corner under SW4.

8–☐ **Non-polarized Capacitors.**

Next install the non-polarized capacitors. These are the smallest components on the board. After installing, solder and clip leads close to the board.

○ **C3**–4700 pF, *marked "472."*, above U8.

○ **C4**–0.1 µF, *marked "104."*, right edge next to U7.

○ **C6**–0.1 µF, *marked "104."*, above SW2.

○ **C7**–0.1 µF, *marked "104."*, top right corner above U9.

○ **C8**–0.1 µF, *marked "104."*, between U9 and U2.

○ **C10**–0.1 µF, *marked "104."*, just above 6811.

○ **C12**–0.1 µF, *marked "104."*, left of U2.

9–☐ **Resistor Packs.**

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

The "catty-cornering" technique of soldering the two end terminals first is helpful here. Solder the two ends of the terminals before soldering the middle pins. This will allow you to align the resistor pack and make it straight before all the pins are fastened.

○ **RP1**–47kΩ×9, 10 pins, polarized, *marked "E47KΩ."*, located at the bottom of the board.

○ **RP2**–47kΩ×5, 6 pins, polarized, *marked "E47KΩ."*, located at the bottom of the board. You must cut off one pin which is the farthest from the marked end before mounting the component.

RP1 and RP2 are pull-up resistors for the digital inputs and the analog inputs. Only the analog 10 and 11 pull-up resistors are connected. Since analog 8 and 9 are multiplexed inputs, a pull-up resistor is connected to each of the multiplexed inputs. If you use the analogs without the expansion board, you must jumper the pull-ups to analog 8 and 9.

- ◯ **RP3**–1k$\Omega\times$3, 6 pins, non-polarized, *marked "V1KΩ,"* top right corner.
- ◯ **RP4**–1k$\Omega\times$5, 6 pins, polarized, *marked "E1KΩ,"* bottom left corner.

10–☐ IC Sockets.

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

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

The socket used for the first two DIPs are different from the other sockets. The pins are more rounded, and the sockets are circular holes. The other sockets have more rectangular socket holes. These are gold plated sockets and are used for a better conduction pathway to the motor outputs.

- ◯ **DIP13/14**–16 pins. There is only one socket for these two ICs. (Use the gold plated sockets)
- ◯ **DIP15/16**–16 pins. There is only one socket for these two ICs. (Use the gold plated sockets)
- ◯ **DIP4**–16 pins
- ◯ **DIP5**–20 pins
- ◯ **DIP6**–20 pins
- ◯ **DIP7**–14 pins
- ◯ **DIP8**–16 pins
- ◯ **DIP9**–14 pins
- ◯ **DIP10**–16 pins
- ◯ **DIP12**–14 pins

11–☐ Direct Mount Chip.

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

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

U3–74HC373.

12–☐ Ceramic Resonator.

Install **XTAL1**, 8 Mhz. ceramic resonator. This is a heat sensitive device and the soldering iron should not be in contact with any given pin for more than about eight seconds.

13–☐ Inductor.

Install **L1**, 1 μH, located below the OFF and ON markings on the board. The inductor looks like a miniature coil of wire wound about a thin plastic core. It is about the size of a resistor.

14–☐ 28-pin Socket.

You must first cut out the bar across the middle of the socket. Do this carefully by scoring the bar and then cutting it with wire cutters. Install on top of **U3**, with the notch marking as indicated. Solder.

15–☐ LEDs.

These LEDs draw less current than other LEDs in you kit. If you put the wrong LEDs in, your batteries will die out much faster than you expect.

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

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

- ○ **LED1**–red, indicates motor 0 in reverse direction.
- ○ **LED2**–red, indicates motor 1 in reverse direction.
- ○ **LED3**–red, indicates motor 2 in reverse direction.
- ○ **LED4**–red, indicates motor 3 in reverse direction.
- ○ **LED5**–red, indicates IR emitters are on.
- ○ **LED6**–red, indicates Low Battery.
- ○ **LED7**–green, Indicates motor 0 in forward direction.
- ○ **LED8**–green, Indicates motor 1 in forward direction.
- ○ **LED9**–green, Indicates motor 2 in forward direction.
- ○ **LED10**–green, Indicates motor 3 in forward direction.
- ○ **LED11**–green, Indicates serial receive.
- ○ **LED12**–yellow, Indicates serial transmit, and is off when the board is in download mode.

16–☐ Resistors.

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

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

- ○ **R2**–47kΩ, *yellow, violet, orange,* upright mounting, above U8.
- ○ **R3**–100kΩ, *brown, black, yellow,* upright mounting, right side above U7.
- ○ **R4**–10kΩ, *brown, black, orange,* upright mounting, right side above U7.
- ○ **R5**–3.3kΩ, *orange, orange, red,* upright mounting, right side of U9.
- ○ **R6**–2.2kΩ, *red, red, red,* upright mounting, top right corner.
- ○ **R9**–47kΩ, *yellow, violet, orange,* upright mounting, above 6811.
- ○ **R10**–47kΩ, *yellow, violet, orange,* upright mounting, above 6811.
- ○ **R14**–4.7kΩ, *yellow, violet, red,* upright mounting, center underneath IR out. Mislabeled "5k" on silkscreen.
- ○ **R15**–1kΩ, *brown, black, red,* upright mounting, under R14.

17–☐ Polarized Capacitors.

All of these capacitors are polarized. Make sure that the lead marked (+) on the capacitor goes into the hole that is marked (+). Some of the tantalum capacitors are not marked. If the capacitor leads are not marked (+) or (−), the lead marked with a dot or bar is the (+) lead. Be careful. The electrolytic capacitors have a bar with a minus sign in them, and these are the negative terminals.

- ○ **C1**–10 μF Tantalum, right side of U8.
- ○ **C2**–10 μF Tantalum, above U8.
- ○ **C5**–47 μF Electrolytic, above U7. Fold capacitor flat to the board before soldering.
- ○ **C9**–4.7 μF Tantalum, above 6811.
- ○ **C13**–470 μF Electrolytic. Fold capacitor flat to the board before soldering. You will need to extend the capacitor into the space next to the oscillator. This is a tight squeeze between the oscillator and the IC sockets.

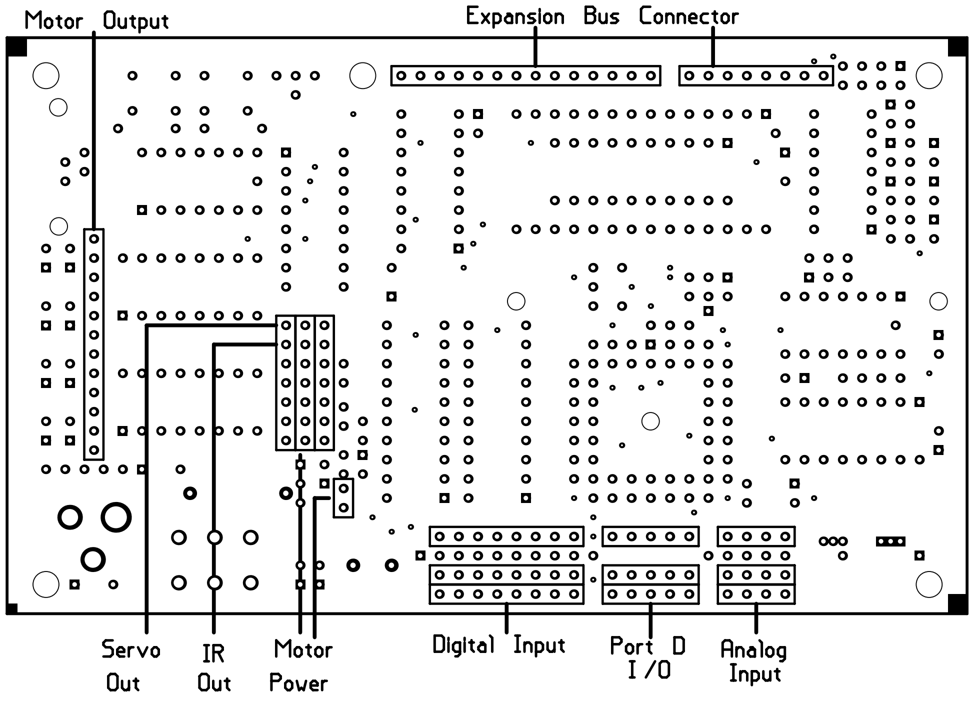


Figure 4.13: 6.270 Microprocessor Board Header Placement

18–☐ Diodes.

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

- ◯ **D1**–1N4001, right of SW1. This diode has a black epoxy body and fairly thick leads.
- ◯ **D2**–1N4148, left of SW2. This is a glass-body diode.
- ◯ **D3**–1N4148, under U2.
- ◯ **D4**–1N4148, next to U9.
- ◯ **D5**–1N4148, next to U9.
- ◯ **D6**–1N4148, next to U9.

19–☐ Female Socket Headers.

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

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

Refer to Figure 4.13 for placement of these parts.

- ◯ Cut three 8-long strips, Install the **Digital Input** connector block. Before placing these header, make sure that RP1 is properly aligned. Once the female header is in place, removal of RP1 is nearly impossible, You may wish to solder all three strips simultaneously. The male pins can be put across the three strips at each end to make sure the female strips are aligned properly. Solder.
- ◯ Cut three 5-long strips. Install the **Port D I/O** connector block. You may wish to solder all three strips simultaneously. The male pins can be put across the three strips at each end to make sure the female strips are aligned properly. Solder.
- ◯ Use three 4-long strips. Install the **Analog Input** connector block. Before placing these header, make sure that RP2 is properly aligned. Once the female header is in place, removal of RP2 is nearly impossible, You may wish to solder all three strips simultaneously. The male pins can be put across the three strips at each end to make sure the female strips are aligned properly. Solder.
- ◯ Cut one 12-long strip. Install the **Motor Output** connectors. Solder.
- ◯ Cut one 8-long and one 14-long strip. Install the **Expansion Bus** connector. Solder.
- ◯ Cut three 7-long strips. Install the **Motor Power** connector. Solder.
- ◯ Cut one 2-long strip. Install the **Expansion power** connectors. Solder.

20–☐ PLCC Socket

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

21–☐ Switches.

- ◯ **SW1**–DPDT slide switch
- ◯ **SW2**–large red pushbutton switch
- ◯ **SW3**–miniature pushbutton switch. The switch is polarized and will fit snugly in one direction and not the other. Do not bend the leads too much, or force the switch in.
- ◯ **SW4**–miniature pushbutton switch. The switch is polarized and will fit snugly in one direction and not the other. Do not bend the leads too much, or force the switch in.

22–☐ Trimpot.

Install **VR1**, 50kΩ.

23–☐ U11. Reset Power Regulator – DS1233

This component looks like a transistor and is located with the expansion board ICs in your parts bin. DS1233 goes here with the rounded part towards the PLCC socket, as shown on the silk screen on the board.

This part looks a lot like the MPS2222A transistors. Be sure you install the DS1233 here, or your board will not work.

24–☐ Transistor

Install **TIP120** such that the metal backing is facing the expansion port power connector and the plastic is facing the DC power jack. This is a tight squeeze. Fold the transistor to the left so it lies flat above the inductor (L1). It is important that the heat sink on the transistor does not touch the edge of the Expansion Board.

25–☐ Piggy-Backing the L293 Chips.

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

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

Begin by sliding the gold-colored heat sink over an L293B chip. Then, press this assembly onto an L293D chip, as indicated in Figure 4.14. Make sure that the two chips have their notches lined up. Also, be sure to remember where which way the notches face, as they may be obscured.

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

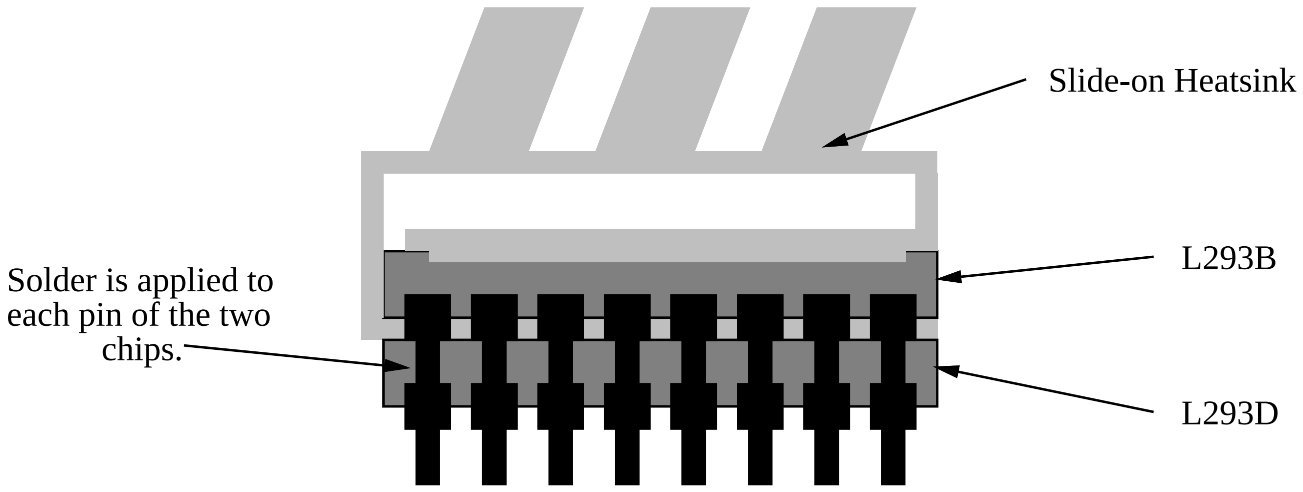


Figure 4.14: Motor Chip Stacking Technique

Be careful not to apply too much heat to the IC. Soldering the opposite corners will help secure the ICs in place and will make soldering the remaining pins easier.

Repeat for the other pair of motor driver chips.

By piggy-backing the two chips, there is a parallel circuit for the motor current to flow through, so the amount of current that can be delivered to the motor is almost doubled to about 1.2 Amps.

26–☐ Power Jack.

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

27–☐ Phone Jack.

Install **J2**, modular phone jack. The phone jack is polarized, and should pace outward.

28–☐ Piezo Beeper.

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

29–☐ Battery pack.

◯ Clip connector on the battery pack and about 1/2" of length off battery pack leads.

◯ *From bottom of board*, insert leads for battery pack. Note polarization: black lead goes in hole marked (−), red lead in hole marked (+). Solder from top of board and clip leads.

4.5.2 Testing the Microprocessor Board

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

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

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

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

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

Make sure that joints do not have too much solder.

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

Resistance should increase as the board capacitor charges. The board resistance should measure greater than 0. *If a reading of zero ohms is observed, the board probably has a power to ground short.* Do not proceed with testing until this is corrected.

3–☐ Insert 4 AA batteries into battery holder.

4–☐ Turn on board power switch.

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

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

7–☐ Install ICs in the board. *Be careful not to damage the component leads when installing the chips into their sockets!* Make sure to get the orientation correct—refer to Figure 4.12 if necessary. Remove the 4AA batteries before installing the ICs.

- ◯ **U1**–68HC11A0 microprocessor. The brains of the board. A 68HC11A1 microprocessor may be substituted.
- ◯ **U2**–62256LP 32K static RAM where the memory is stored.
- ◯ **U3**–74HC373 (already soldered to board). This is the latch that is used to access the memory locations.
- ◯ **U4**–74HC138. This is the address decoder for memory mapping input and output latches
- ◯ **U5**–74HC273. This is the output latch to the motors 1-4. Upon power up, the latch is cleared so all the motors are turned off when the board is turned on.
- ◯ **U6**–74HC244. This is the tristate input latch which drives the bus for reading the digital inputs.
- ◯ **U7**–74HC132. Schmitt trigger used for the serial communications with the downloading machine.
- ◯ **U8**–74HC4053. Used to switch the RS232 TxD.
- ◯ **U9**–74HC10. 3-input NAND used in the low-battery indicator.
- ◯ **U10**–74HC390. Dual decade counter that divides the 2MHz clock to a 40kHz signal used in the IR emitters.
- ◯ **U12**–74HC04. An inverter through which the motor outputs to the L293 are inverted. Because the L293s draws less processor current when all the inputs are high than when all the inputs are low the outputs are inverted so that when all the motors are off, all the inputs to the 293 are high. Another inverter is used to invert the signals in the IR circuitry.
- ◯ **U13,14**–L293D + L293B motor driver assembly with heatsink
- ◯ **U15,16**–L293D + L293B motor driver assembly with heatsink

4.5.3 Board Checkoff

You now have the components in place to check your board. Follow the instructions to check off your board. If there are any problems along the way, check the debug chapter or find someone to help you.

1–☐ Turn the board off, and plug in the AA batteries. You should also plug the motor batteries into the board. When the motor batteries are not plugged in, the motor chips draw power from the processor batteries, and therefore reduce the lifetime of your AA cells. To extend the lifetime of your AA cells, always have your motor batteries plugged in.

2–☐ Attach the expansion board on top of the microprocessor board. Be careful not to bend any pins, and make sure that all the pins go in the correct sockets. Then put the LCD on the expansion board.

3–☐ When you turn on the board, the yellow light should be on. To get the board into download mode, you must turn the board off, and while holding down the escape button, turn the board on. The yellow light should flicker, and then be off. If this does not happen, check the debug section under startup. When the yellow light is off, the board is in download mode.

4–☐ When you hit the big red reset button, the yellow light should come on permanently. If the yellow light does not come on, then check the debug section under startup.

5–☐ In order for your board to work, the pcode must be loaded into the board. You need to do this only when there are new revisions of the pcode and when the RAM memory has been corrupted.

There are two types of downloading. The first download mode is the mode where the microprocessor load the pcode. This is done using the **init_bd** command. When the yellow LED is off, the processor is ready to accept new assembly code.

To download the pcode to the board you must:

- Plug the serial cable into the board. The green LED should be on whenever the board is connected to the host computer. If the green LED is not on, check the debug section on serial problems.
- Turn on the board. (SW1)
- Hold down the “CHOOSE” button (SW3) while pressing the red reset button (SW2). Release the reset button. Watch for the yellow serial transmit light to extinguish. Release the “CHOOSE” button. Your board is now in pcode download mode.
- You need to download the pcode to the board. The command to download will be different, depending on the machine you are using. On the Athena workstations you must **add 6.270** and then type **init_bd** at the prompt. The yellow LED must be off for downloading to occur.
- You should get the following response, or something similar, when downloading the pcode:

```
6811 .s19 file downloader.  Version 6.1  16-Nov-91
 Downloading 256 byte bootstrap (229 data)
```

and a bunch of dots should appear on the monitor and the yellow and green LEDs should begin to flicker. This flickering lets you know that there is communication between the board and the host computer. If the response is a different, check the debug section, on downloading or serial problems.

- Once the pcode is loaded into the board, press the reset button, and the board should beep and on the LCD should be the message:

  ```
  Interactive C
  V 2.71 1/4/93
  ```

  or something similar. You may need to adjust your LCD contrast to see the messages on the LCD. Do this by turning the variable resistor VR1.

The second download mode is through IC when you download your code or the IC libraries. The yellow light must be on for this download to occur. When you press reset, the yellow light should come on.

6–☐ When the pcode is loaded into the board, you can use the IC program. This can be done by simply typing **ic** at the prompt when the board is connected to the host computer. The IC program will download several libraries to the board.

7–☐ The next step is to use the test program to make sure that all the outputs and inputs are working. Before you do this, you must build a simple digital sensor for testing the input ports.

8–☐ After you are in the IC program, you will need to load the test code into the board. To do this, at the **C>** prompt type **load testboard.c** to load in the test code. This program is designed to help you become familiar with the board, and where things are located.

9–☐ The following tests will be performed:

- Check Motor Outputs.
- Check Digital Inputs.
- Check Analog Inputs.
- Check Dip Switches.
- Check Frob Knob.
- Check LED outputs.

You can advance through the menu by using the ESC button and the CHOOSE BUTTON. To advance to the next test push the ESC button. When checking the analog and digital ports push the CHOOSE button to advance the port number.

10–☐ You should test each of the digital and analog ports using a digital switch. The analog readings for an open switch should be around 255 since the inputs are connected to a pull up resistor. When the switch is closed, the analog reading should yield a low number below 50. The digital ports should show a 1 when the switch is closed and a zero when it is opened.

4.5.4 After Board Checkout

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

1–☐ Use 2 strips of double sticky tape to attach the AA battery pack to board. Make sure that the wire in the AA holder does not come in contact with any of the protruding leads on the underside of the board. Many problems can occur is the wire in the battery pack shorts adjacent leads on the board.

You should then bring your boards to one of the organizers or a TA to get it checked off. The checkoff procedure will require that you have some knowledge of the location of the ports and we expect you to try out the test code before checkoff.

4.6 The Infrared Transmitter

The infrared (IR) transmitter board emits modulated infrared light that can be detected by the Sharp IR sensors (of type GP1U52). The board has infrared transmitting LEDs that are driven by a divide by 50 counter (the 74HC390 chip) and a power transistor (TIP120) on the Microprocessor Board.

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

4.6.1 Assembly Instructions

Figure 4.15 illustrates component placement on the infrared transmitter board. *Note that the LED numbering that was printed on the actual boards is incorrect.* The numbering shown in the figure is correct.

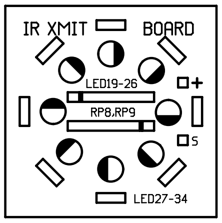


Figure 4.15: Infrared Transmitter Component Placement

1–☐ Resistor Packs.

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

- ◯ **RP8**–33Ω×4
- ◯ **RP9**–33Ω×4

2–☐ Visible LEDs.

The visible LEDs used on the infrared transmitter board have *red lenses.* They should look similar to the low powered LEDs. These will be in the bag with the IR beacon board. *Be sure to use this variety of LED here.* These LEDs can handle more current than the LEDs that have been used in other circuitry. The LEDs will glow red when powered.

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

- ◯ **LED23**–red lens, red element
- ◯ **LED24**–red lens, red element
- ◯ **LED25**–red lens, red element
- ◯ **LED26**–red lens, red element
- ◯ **LED27**–red lens, red element
- ◯ **LED28**–red lens, red element
- ◯ **LED29**–red lens, red element
- ◯ **LED30**–red lens, red element

3–☐ Infrared LEDs.

The infrared LEDs come in a small rectangular package.

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

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

○ **LED31**–MLED71 IR LED

○ **LED32**–MLED71 IR LED

○ **LED33**–MLED71 IR LED

○ **LED34**–MLED71 IR LED

○ **LED35**–MLED71 IR LED

○ **LED36**–MLED71 IR LED

○ **LED37**–MLED71 IR LED

○ **LED38**–MLED71 IR LED

4–☐ **Cable and Connector.**

○ Cut a 12" length of the twisted-pair red/black cable. Strip $\frac{1}{4}$" of insulation from the wire on both ends.

○ From *underside* of IR board, insert red wire into hole marked (+) and black wire into hole marked (−). Solder from *top* of board.

○ Mount other end of red wire to the middle pins of a three-pin male connector and the black wire to one of the outside pins. Use guideline shown in Section 4.7.

The infrared transmitter plugs into the connector labelled IR OUT on the Microprocessor Board (see Figure 4.13), with the red lead inserted into middle (power) strip and the black lead plugged into the right hand (signal) strip. The transistor acts as a switch between the signal lead of the IR emitter and ground so no current may flow when the signal to the base of the transistor is off. If the connector is plugged in backwards, the IR LED will always be on, and the transmitter will get very hot.

4.7 Cable and Connector Wiring

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

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

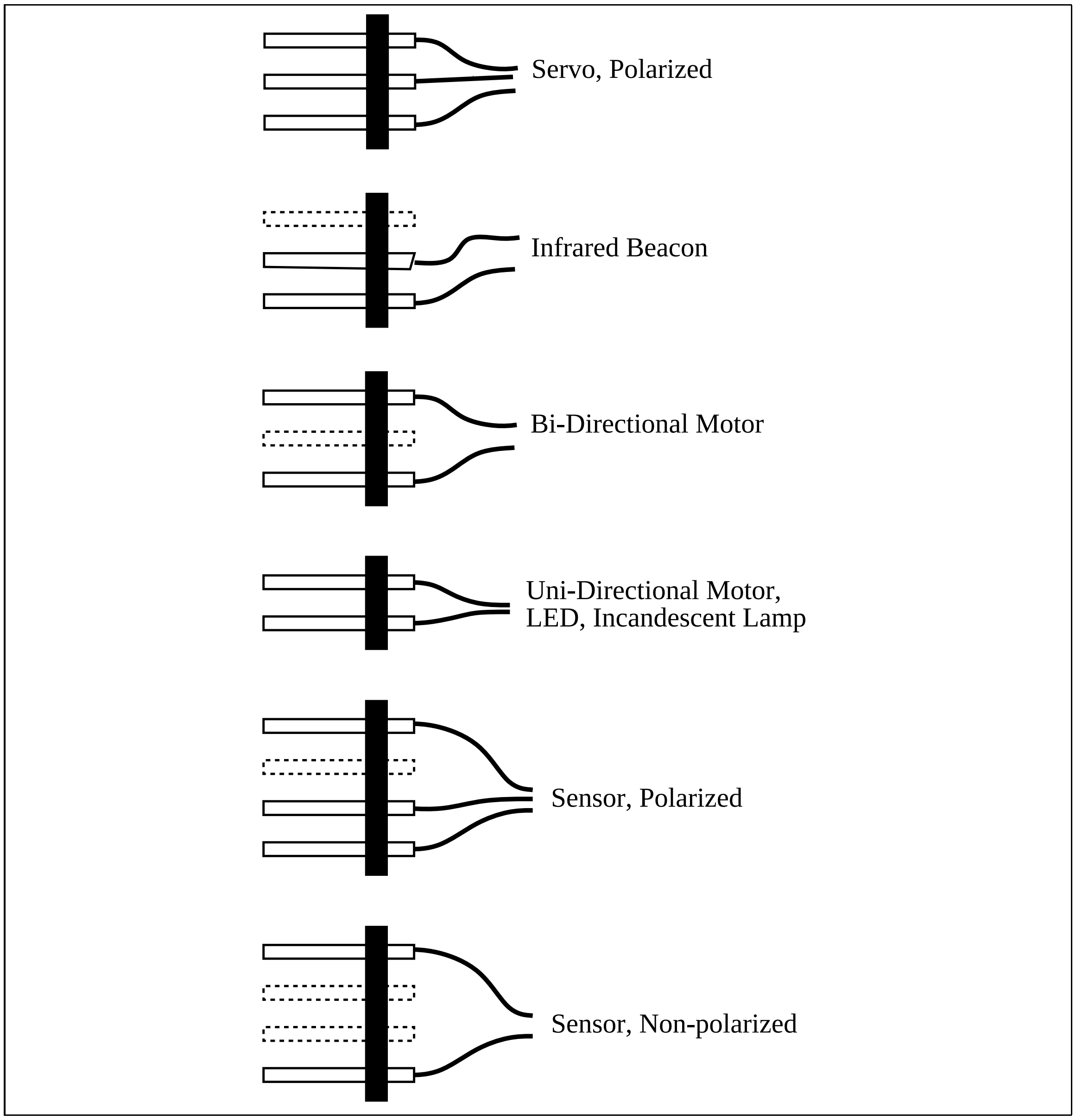
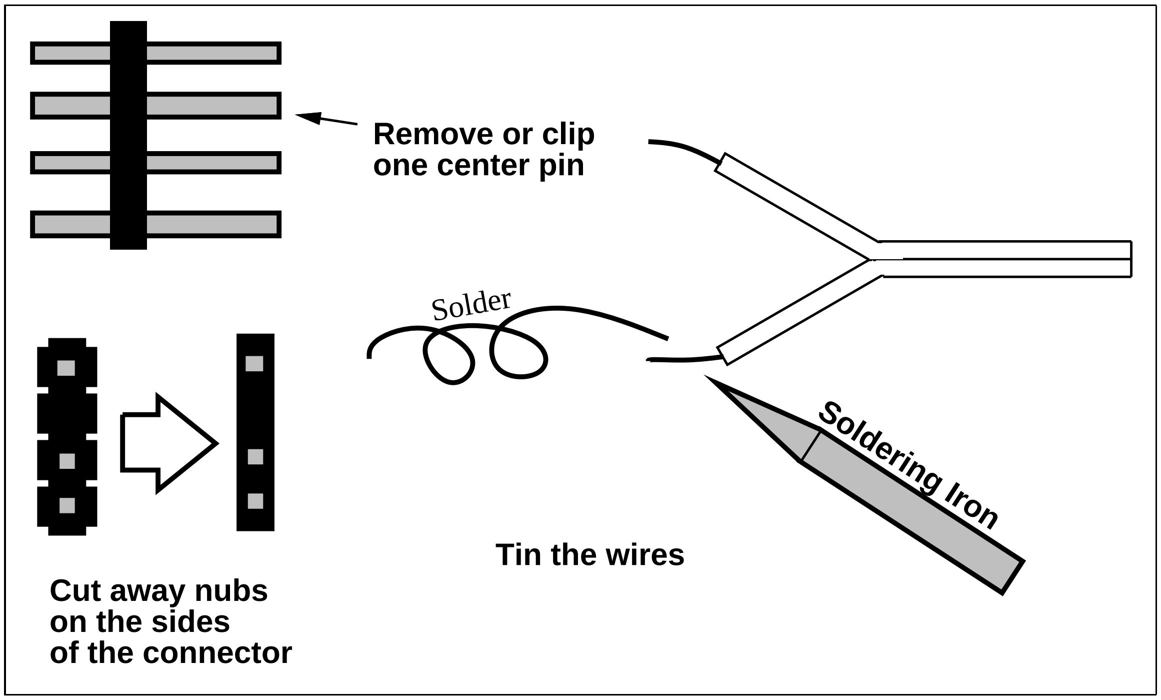
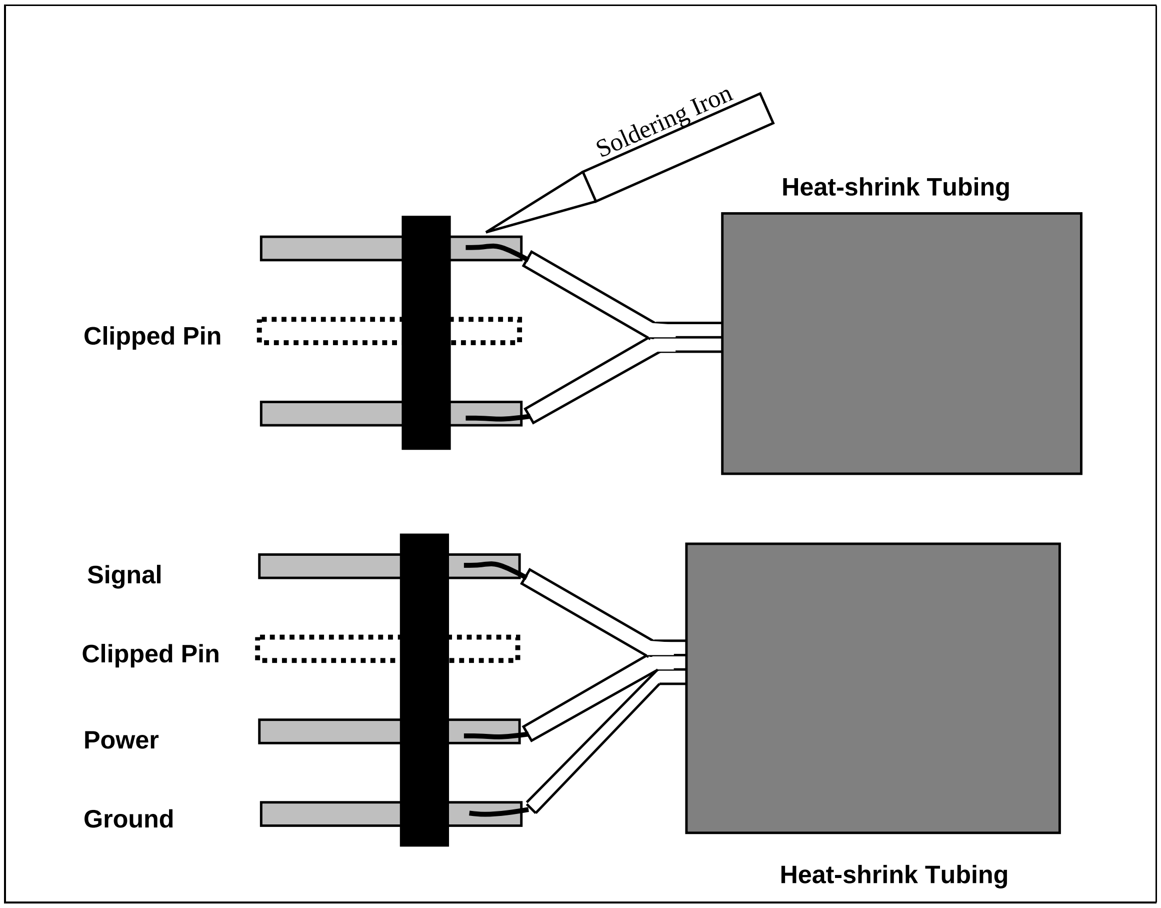


Figure 4.16: Standard Connector Plug Configurations



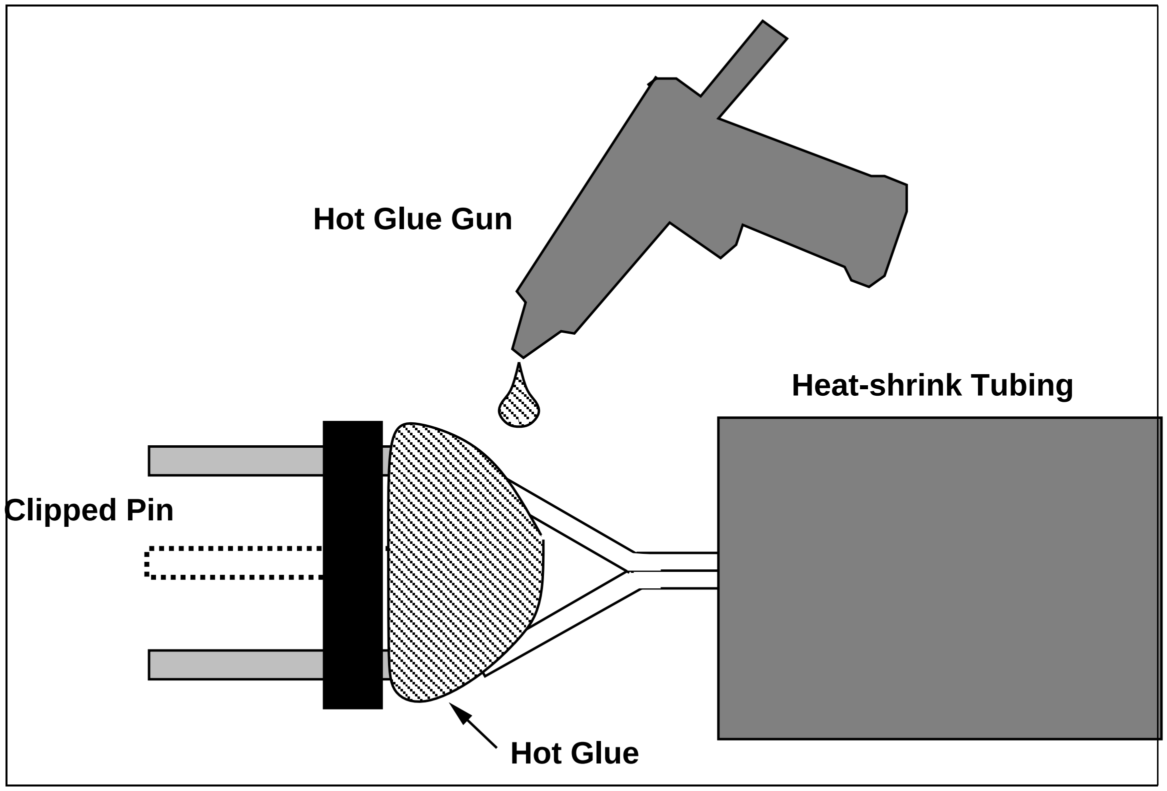
Strip a small amount of insulation off the wire ends. Tin the wire ends by applying a thin coat of solder to them.

Figure 4.17: Step One of Connector Wiring



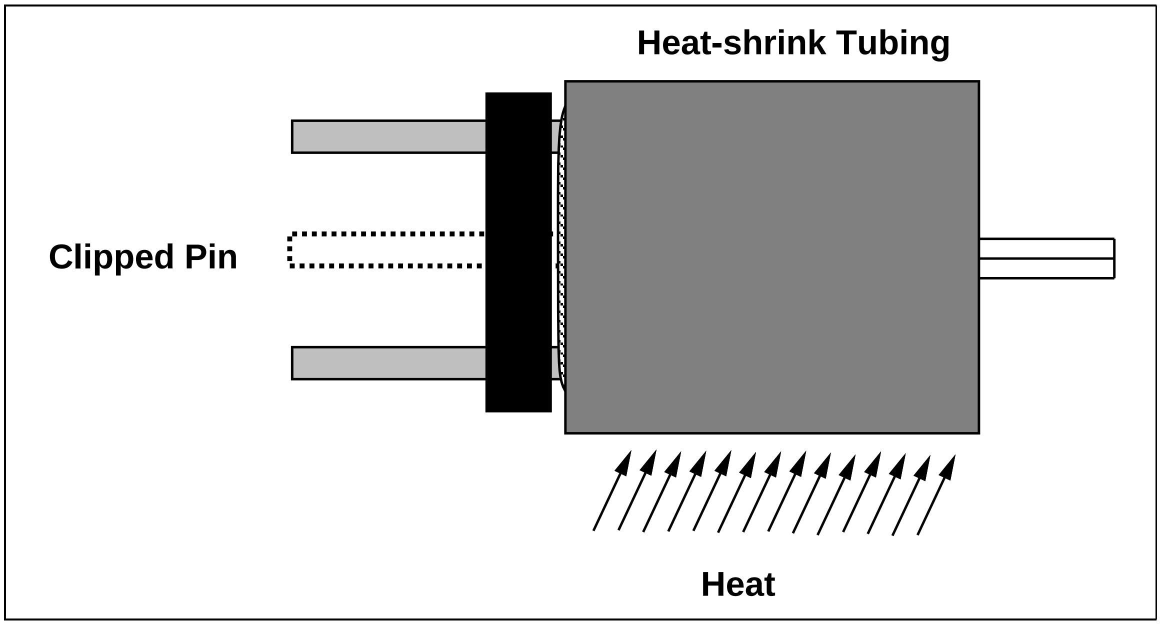
Cut the male connector to size.This example shows a plug that can be used to wire a motor and the bottom a polarized sensor. Cut $\frac{1}{2}$ inch length pieces of 1/4" heat-shrink tubing. Solder the wires to the connector, being careful not to let the heat from the soldering iron shrink the tubing.

Figure 4.18: Step Two of Connector Wiring



Use hot glue to strengthen and insulate the connection. Be careful not to use too much glue, or else the connector will be too fat. While the glue is cooling off, you should slide the heat shrink tubing over the glue and flatten the glue while it is till soft.

Figure 4.19: Step Three of Connector Wiring



Slide a piece of heat-shrink tubing over connections. Shrink using heat gun, flame from a match or lighter, or the side of a soldering iron. A heat gun provides by far the best results, and you may want to come to lab to use one.

Figure 4.20: Step Four of Connector Wiring

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

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

Several different connector styles are used depending on the device which is being connected to. Figure 4.16 shows the connector configurations used for bidirectional motors, unidirectional motors, sensors, and the infrared beacon.

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

Figures 4.17 through 4.20 illustrate the recommended method for wiring to a connector plug. When assembled properly, this method will provide for a sturdy, well-insulated connector that will be reliable over a long period of use. Too much glue or heat shrink tubing will make the connector fat, and later you will not be able to connect several connectors side by side. You will want to keep the connectors small and sturdy.

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

4.8 Motor Wiring

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

4.8.1 The Polaroid Motor

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

These instructions describe how to attach an eight-tooth LEGO gear to a Polaroid motor. Other configurations are possible, but require considerably more work. Speak to a TA or organizer if you really want to use a different setup.

Attaching a LEGO Gear to the Polaroid Motor

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

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

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

○ Get some small ($\frac{1}{16}$" and $\frac{1}{8}$") heat-shrink tubing. Shrink a couple of layers of tubing onto the motor shaft. An eight-tooth LEGO gear should now fit snugly over the tubing.

○ Cut off any excess tubing which sticks out from the LEGO gear. Place a drop of super glue around the outer area of the motor shaft farthest away from the motor housing. Using a paper napkin, pat off any of the excess super glue.

Attaching the Polaroid Motor to a LEGO Base

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

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

This platform or jig is shown in Figure 4.21. It is constructed from two 2×8 beams, one 6×8 flat plate, one 2×4 plate, two 24-tooth gears, and two axles.

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

○ Assemble the jig as shown in Figure 4.21. A second 2×4 plate will be mounted to the motor.

○ Cut off the LEGO nubs from the second 2×4 plate that will be connected to the motor. Place a piece of double-sided sticky tape on the plate.

○ Position the motor on its plate so that the 8-tooth gear is meshed between the two 24-tooth gears, and the center line of the motor shaft is parallel with the axles of the 24-tooth gears. Remove the paper from the tape and secure the motor onto the tape.

○ With a second 2×4 plate, cut off the bottom ridges so that it is a flat piece with just the nubs, or use a 2×4 piece of grey plate. Attach a piece of double-sided sticky tape to the bottom of the piece.

○ Make a second jig as shown in Figure 4.22 out of four 2×4 bricks, two 2×4 plates, and two 6×8 plates that sandwich the motor in place.

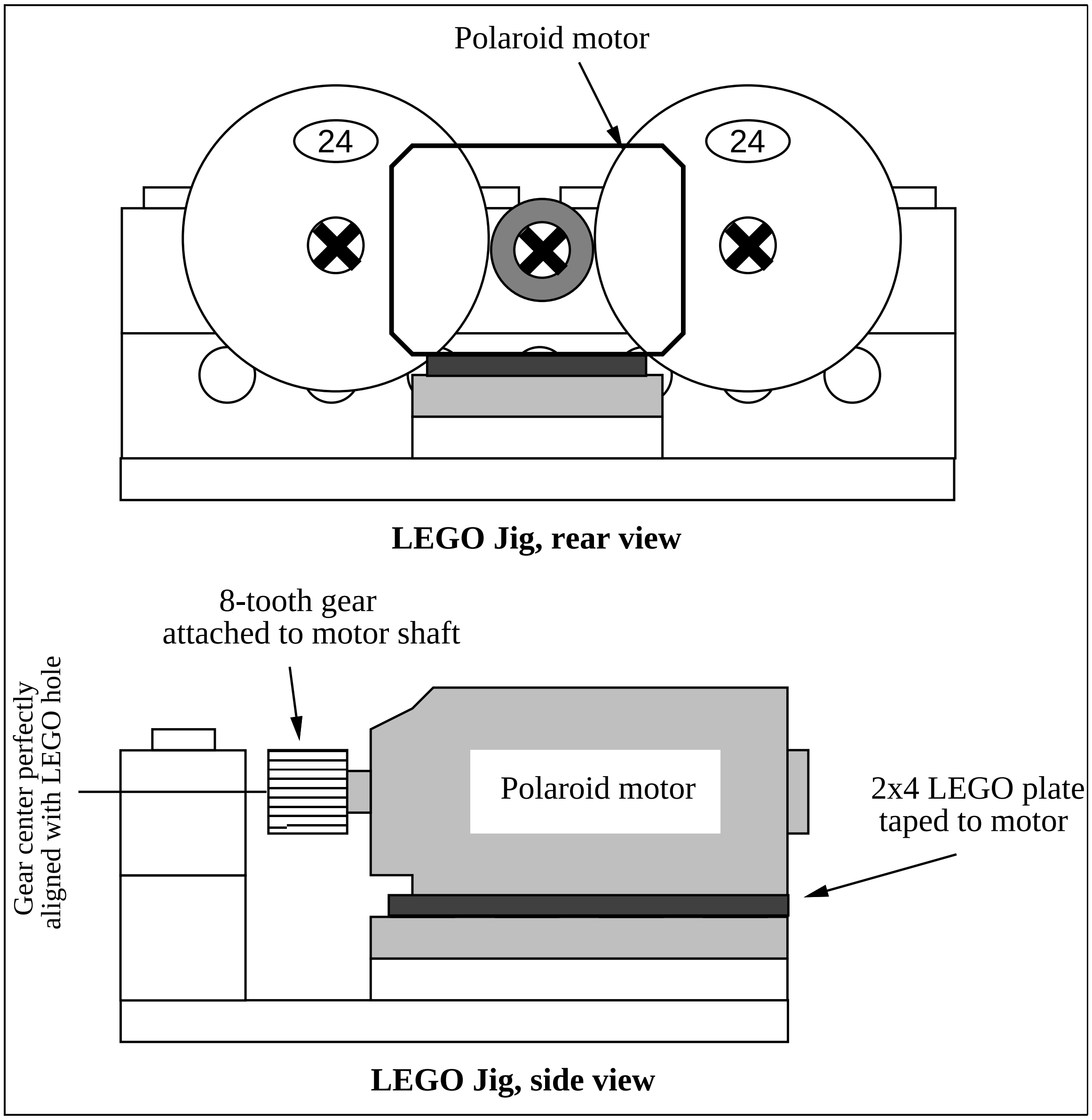

Polaroid motor
24
24
LEGO Jig, rear view
8-tooth gear attached to motor shaft
Gear center perfectly aligned with LEGO hole
Polaroid motor
2x4 LEGO plate taped to motor
LEGO Jig, side view

Figure 4.21: LEGO Jig for Mounting Polaroid Motor

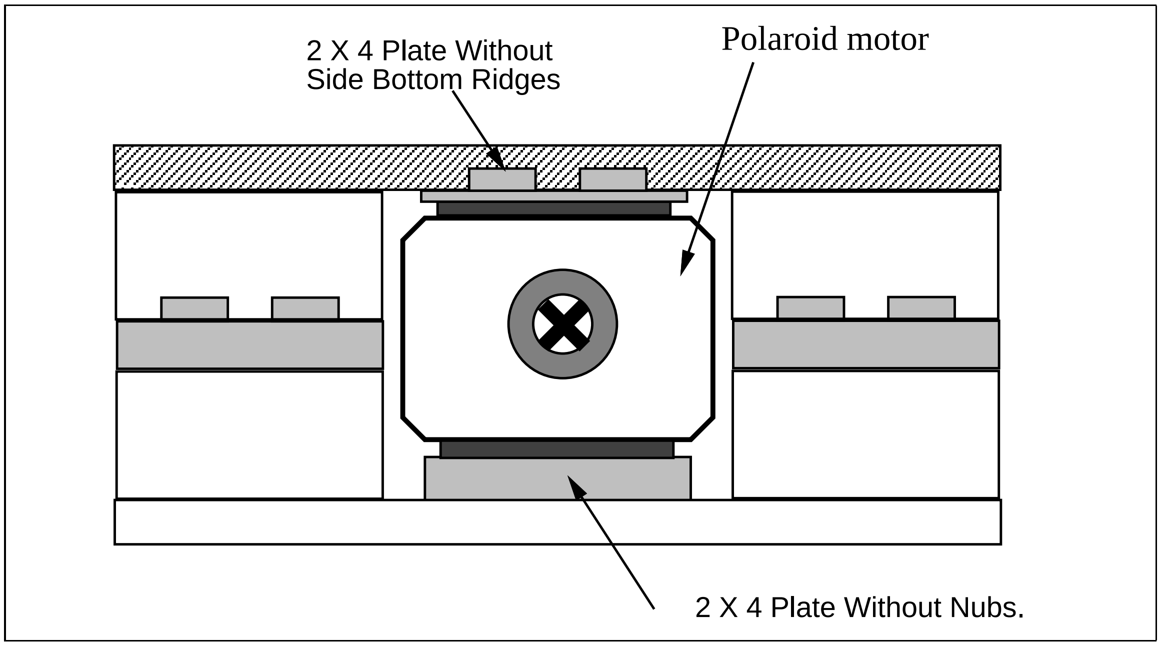


Figure 4.22: LEGO Jig for Mounting Polaroid Motor

- ◯ Attach the piece to the motor such that the motor can be locked into place when another piece is attached across the top of the motor. It is probably a good idea to lock in your motors in a similar fashion when building your machine.

Wiring a Cable and Plug to the Polaroid Motor

- ◯ Motor cables may be constructed with either two strands of ribbon cable wire or the twisted pair red/black cable. Cut an 8 inch to 12 inch length of whichever wire is preferable.
- ◯ Strip and tin both ends of the wire.
- ◯ On the side of the motor there should be two metal lead/pads. Solder one wire lead to each pad. After proper soldering, hot glue may be used to hold the wire to the side of the motor for a stress relief.
- ◯ Motor plugs may be wired for bidirectional or unidirectional use, as shown in Figure 4.16. (For most purposes, motors will need to be operated bidirectionally.)

  Cut a two- or three-long strip of male socket headers as will be needed.
- ◯ Using the connector plug wiring technique shown in Figure 4.17 through Figure 4.20, wire the motor plug. Polarity does not matter since the plug may be

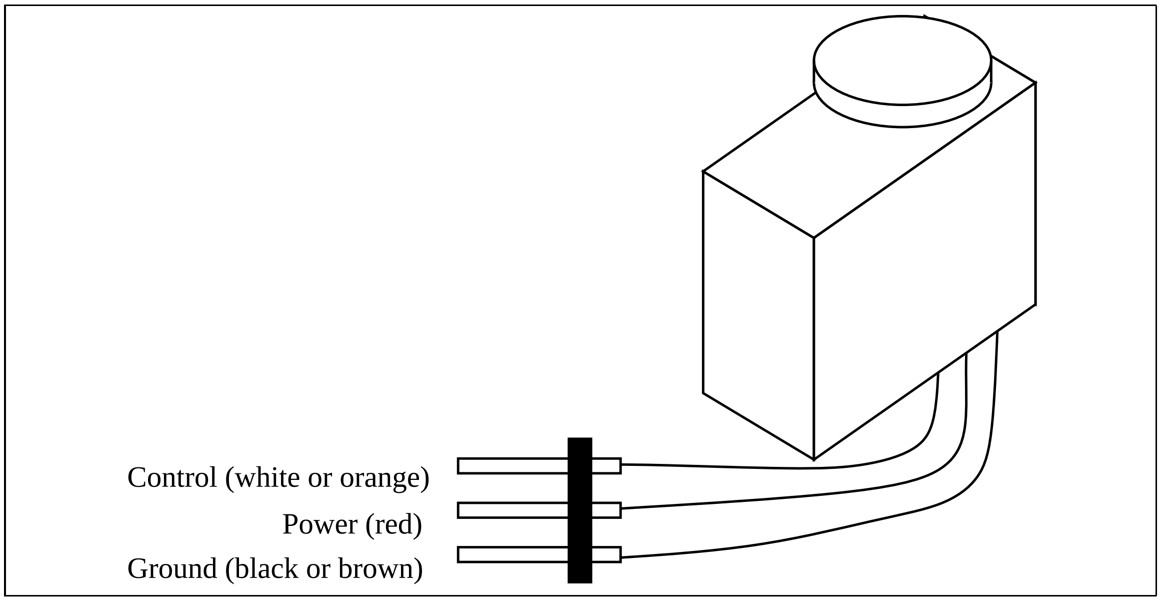


Figure 4.23: Servo Motor and Integral Connector Plug

inserted into a motor power jack in either orientation.

4.8.2 Servo Motor

Figure 4.23 illustrates a typical servo motor similar to the one provided in the 6.270 kit. The servo motor has a short cable that terminates in a three-lead connector, as illustrated. The functions of these lead are power, ground, and the control signal.

Chapter 5

Sensor Design

Close your eyes. Plug your ears. Hold your nose. Tie your hands behind your back. Shut your mouth. Tie your shoelaces together. Spin yourself around a few times. Now walk. How does it feel? That's exactly what your robot feels: nothing–without sensors. You have been given many types of sensors that can be used in a variety of ways to give your robot information about the world around it. In this chapter, we'll explain each of the sensors you have, how it works, what it's good for, and how to build it.

Before we can teach you what sensors do, we need to make one point very clear. Sensors are not magical boxes. The phrase "Sensors indicate a large LEGO robot 2 meters off our port bow, Captain!" will never appear. All information you get from sensors must be decoded by *you*, the human builder and programmer.

Sensors convert information about the environment into a form that can be used by the computer. The sensors that are on the robot can be related to sensors found in humans. Touch sensors embedded in your skin, visual sensors in your retina, and hair cells in your ears convert information about the environment into neural code that your brain can understand. Your brain needs to understand the neural code before you can react. Since you will be programming the robot, you will need to understand the output of the sensors before you can program your robot to react to different stimuli.

Take time to play with each of the sensors you have been given. Figure out how they work. Look at the range of values they returns, and under what conditions it gives those values. The time you spend here will greatly ease your integration of hardware and software later. The better you understand your sensors, the easier it will be for you to write intelligible control software that will make your robot appear intelligent. So as you read about the sensors, you should assemble a bunch of sensors as shown.

Sensors provide feedback to your program about the environment. Feedback is important in any controlled situation. Rather than using open-loop, or timed programs that simply follow a pattern but have no real knowledge of the world, sensors can provide the feedback necessary to let a robot make decisions about how to act in its environment. The feedback mechanism is very important in an environment that is continually changing. During the rounds of the contest, the objects on the playing field will be changing their location (i.e., the other robot moves, the drawbridge closes, or you bump into a block). We strongly encourage you to use closed-loop feedback design when planning and implementing your strategy. There will be a smaller chance of random errors completely messing up your game if you use sensors wisely. See Chapter 8 for more information on the control problems you may encounter.

5.1 Sensor Assembly

You should have read the section on the previous chapter on the types of connectors used with the 6.270 board. This is an important concept to understand before building your sensors.

When building your sensors, **do not make your wires too long**. Excess wiring has a tendency to get caught in gears and other mechanisms. Start out with sensor wires no longer than 1 foot long and when your finally decide on your robot configuration, you can modify to length. Just build a few of each type so you can play with them.

Start out with building simple sensors like one or two switches. The more complicated ones will be the analog sensors that use IR.

5.2 Analog vs. Digital Sensors

The sensors you have can be split into two basic types: analog and digital. Analog sensors can be plugged into the analog sensor ports, which return values between 0 and 255. Digital sensors can be plugged into either the digital ports or the analog ports, but will always return either 0 or 1.

ANALOG $0 <= x <= 255$
DIGITAL 0 or 1

Each type of sensor has its own unique uses. Digital sensors, such as pushbuttons, can tell you when you've hit a wall. Digital sensors always answer a question about the environment with a *yes* or *no*. "Have I hit the wall?" Yes if the switch is closed, or no if the switch is still open.

Analog sensors, such as photoresistors, can tell you how far the sensor has bent, or how much light is hitting the sensor. They answer questions with more detail. Analog sensors, however can be converted to digital sensors using thresholding. Instead of asking the question *How much is the sensor bent?* you can ask the question: *Is the sensor bent more than half way?* The threshold can be determined by playing around with the specific sensor.

5.3 Location of Digital and Analog Ports

The digital ports on the main board are labeled from 0-7 and are shown in figure 4.13. There are also four analog ports on the main board, but when you use the expansion board, the analog ports get remapped to the connectors on the right side of the expansion board. The ports are all arranged in the same format. The inner most row of pins are the signals, followed by a space, then microprocessor power, and finally on the outer side is the ground.

5.4 Digital Sensors

Digital inputs all have *pull-up* resistors connected to them as shown in figure 5.1. Digital switches are wired such that the sensor is wired across the signal pin and ground. This means that when the digital sensors is closed, the signal is grounded or LOW. When the switch is open, the signal pin outputs +5V, or HIGH. This value is INVERTED by software, so reading the digital port with the switch open returns 0, while reading the digital port with the switch closed returns 1. With nothing plugged in, the value of a digital port should be 0.

Digital sensors can be used in the analog ports on the 6.270 Controller board as well, relieving any restrictions the small number of digital inputs may cause. Typical analog values for digital sensors are somewhat above 250 for an open switch, and less than 20 for a closed switch. When using the IC command, `digital(port)`–where port is an analog port number (i.e., greater than 7)–the sensor value is compared to a threshold value, and the command returns a 0 if the analog value is above the threshold or a 1 if the analog value is below it (remember the inversion of the actual signal that digital does?). This threshold's default value is 127, but it can be changed (See the section on IC commands for information on this).

A good way to get digital information from an analog sensor is to plug the analog sensor into a analog port and call it with the `digital(port)` command. For example, a reflectance sensor would return a 0 for black or a 1 for white if read with the digital command–provided the threshold is properly set. This can reduce some of the programming complexity by abstracting away the thresholding. You should however experiment with the sensors to determine the range of thresholds you get and under what conditions these thresholds are valid.

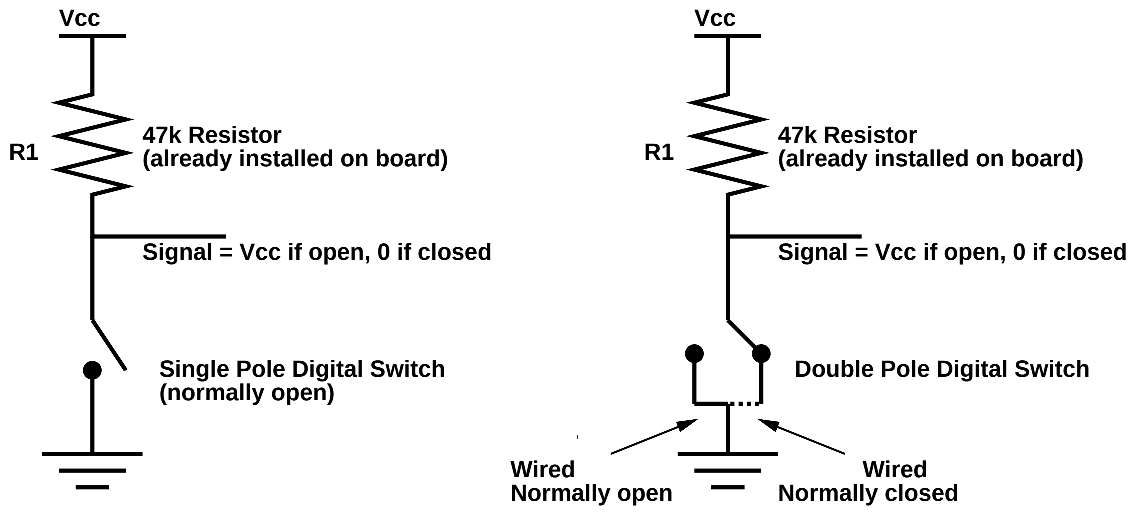

Vcc
R1
47k Resistor
(already installed on board)
Signal = Vcc if open, 0 if closed
Single Pole Digital Switch
(normally open)
Vcc
R1
47k Resistor
(already installed on board)
Signal = Vcc if open, 0 if closed
Double Pole Digital Switch
Wired
Normally open
Wired
Normally closed

Figure 5.1: Generic Digital Sensor Schematics.

It is not recommended to plug analog sensors into digital ports, however, because the digital ports threshold to conventional logic levels which cannot be adjusted to suit each analog sensor. The valid analog readings may fall into the invalid range for digital logic.

Here are some mountings and uses for some digital sensors in the 6.270 kit.

5.4.1 Dip Switches

There are four dip switches on the Expansion Board. They can be used to select user program options during testing. One dip switch will be used in the starting code for the contest to determine the side your robot starts on and at which frequencies it transmits and receives the modulated IR. They can also be useful for outside control of program parameters, like enabling certain functions or selecting programs to run. While these switches are connected to the analog port, they are really digital switches.

5.4.2 Micro-Switches

The standard kit includes three types of small switches, two micro switches and a small push button. These make great object detectors, so long as you are only interested in answering the question, *Am I touching something right now?* with a *yes* or *no*. This is often enough for responding to contact with a wall or the other robot or for actuator position sensing. Using a switch in this manner (called a “limit” switch) can be a good way to protect drive mechanisms which self destruct when over driven. This could be handy for limiting the motion of hinged joints or linear actuators by

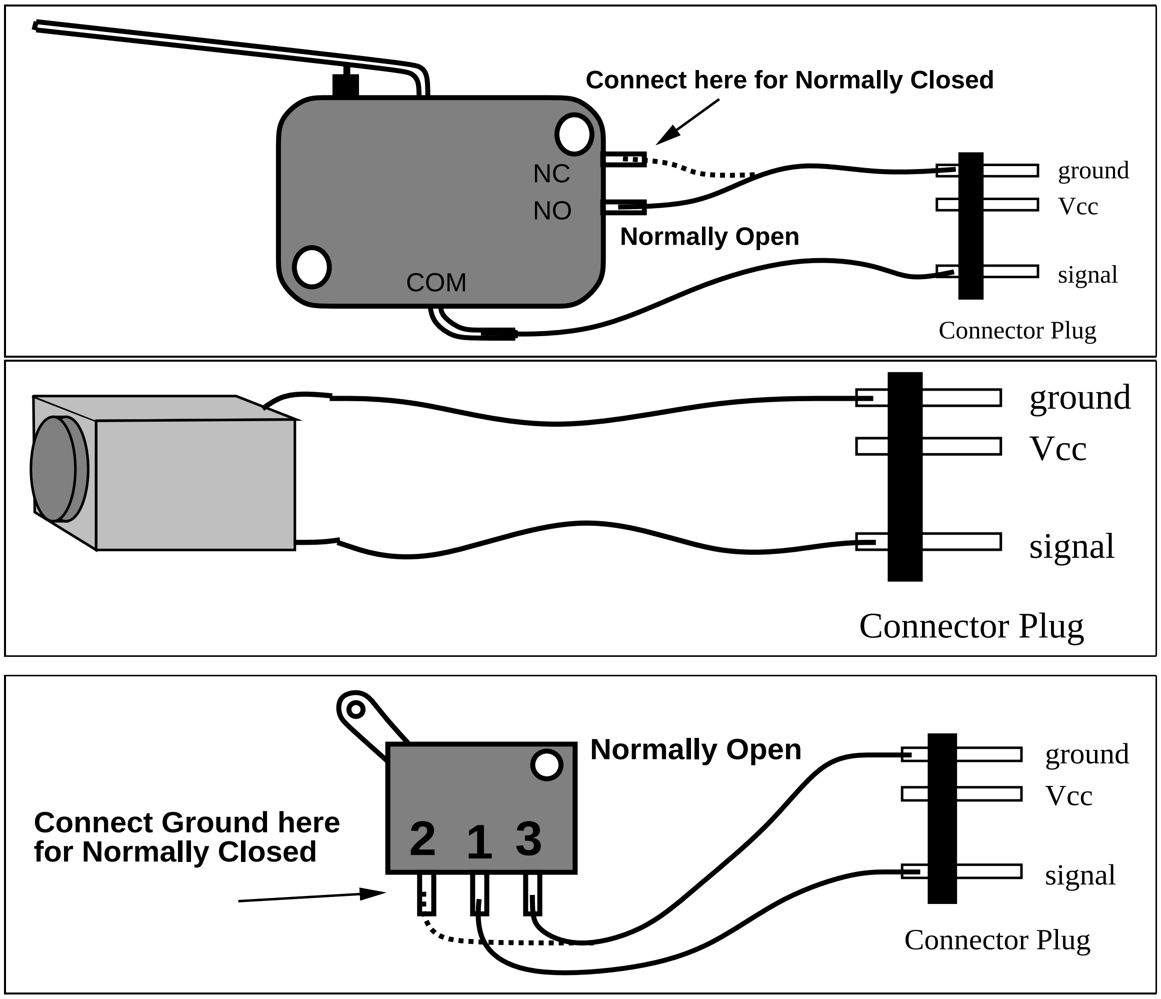


Figure 5.2: Microswitch Assemblies

requiring that a switch be open (or closed, depending upon the situation) before running the motor and monitoring it while things are moving. They could also be used for extended user interface for testing and development purposes.

The two micro switches are double pull, which means they can be wired so that they return a one or a zero when not depressed. The only major difference is how you think about the device in your code. Reading a sensor can be thought of as asking a question. Here, the question could be, "Are you open?" or "Are you closed?" If you wire the switch normally open, the answers are yes and no, respectively, where they would be no and yes for a switch wired normally closed, all for the same situation where the switch is not depressed.

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

In some cases, a slight advantage may result from one arrangement, because there may be a difference between the position where the open side makes contact and the closed side breaks contact. When this is the case, the choice of normally open or normally closed will affect how sensitive the switch is to outside forces. This can allow you to make a very touchy sensing device or help block out noise. The small black switches with the white lever arm respond to a shorter arm movement when wired normally open and require a little more movement to cause a transition in the normally closed configuration.

Bouncing is a problem found in many switches. At the point where the switch goes from open to close or vice versa, the output from the switch is very glitchy. The switch may output several transitions. Bounciness occurs especially when the switch is used in a sensitive mode. One way to *debounce* the switch is to add a delay between samples of the digital input. If the sampling is sparse enough, the bouncing section of the data will not be collected.

5.4.3 Sharp IR Detector

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

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

While it may not seem like a digital sensor because most of the light sensor we deal with are analog, it is a bona fide digital sensor because it detects infrared light modulated at 40kHz. Inside the tin can, there is a IR detector, amplifier, and a demodulator. The sensor returns a HIGH when there is no 40kHz light, and is LOW when it see the 40kHz light.

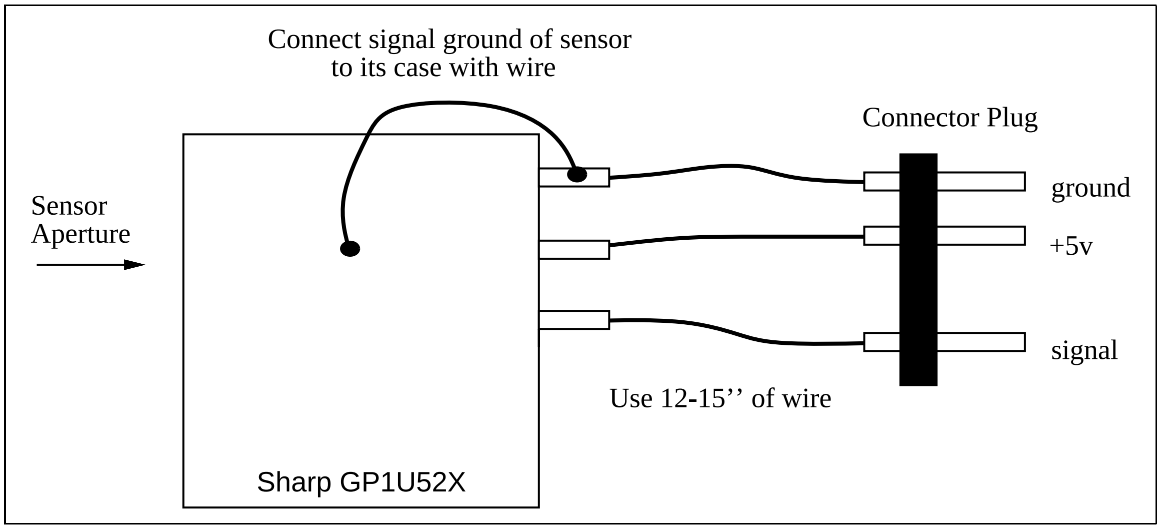


Figure 5.3: Sharp IR sensor assembly

There is a lot of infrared light that is ambient in the air. Some components of this light are at 40kHz, and straight output from the sensor would look very glitchy. The sun produces a lot of IR light, and in the sun, the sensor output bounces all over the place. To eliminate the effect of the stray IR light, the IR emitters are modulated at 100 or 125 Hz (see section A.7 for more information on the IR transmission) and the output of the IR Detectors is demodulated to look for these frequencies. The 40kHz frequency is known as the carrier frequency, and the other frequency is the modulated frequency.

You can use the IC command `ir_counts(port)` to count the number of successive detected periods of the modulated frequency. A count larger than 10 indicates a detection. You may need to play around with what values of the counts are needed for detection. These sensors can only be used in digital ports 4-7.

5.5 Analog Sensors

The analog ports all have a *pull up resistor* which is a 47KΩ resistor between +5 volts and the signal input. The analog readings are generated by measuring the amount of current flow through the pull up resistor. If no current flows through the resistor, the voltage at the signal input will be +5 volts and the analog value will be 255. The voltage at the signal pin can be simply calculated by:

$$V_{sig} = 5 - 47\Omega \times i$$

Reading the value of an analog port **without** a sensor will return a value above 250. With the sensor plugged in, the value should be less. This is one good way to

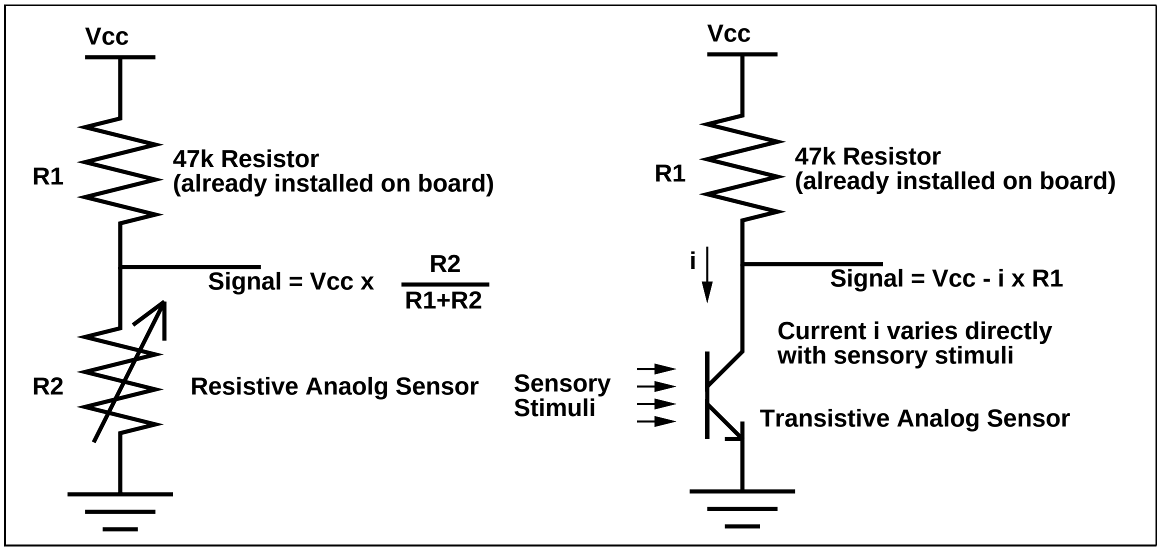

Vcc
R1
47k Resistor
(already installed on board)
Signal = Vcc x R2 / R1+R2
R2
Resistive Anaolg Sensor
Vcc
R1
47k Resistor
(already installed on board)
i
Signal = Vcc - i x R1
Current i varies directly
with sensory stimuli
Sensory
Stimuli
Transistive Analog Sensor

Figure 5.4: Analog Sensors Schematics

check if one sensor fell out: write a piece of code that checks the values of the analog ports that you have sensors plugged into. If that value is above 250 or so, have it tell you to check the sensor.

5.5.1 Resistive Sensors

The resistance of resistive analog sensors, like the bend sensors or potentiometers, change with changes in the environment, either an increase in light, or a physical deformation. The change in resistance causes a change in the voltage at the signal input by the voltage divider relation.

$$V_{sig} = \frac{R_{sensor}}{47\Omega + R_{sensor}} \times 5V$$

5.5.2 Transistive Analog Sensor

Transitive analog sensors, like the photo transistors and reflectance sensors, work like a water faucet. Providing more of what the sensor is looking for opens the setting of the valve, allowing more current to flow. This makes the voltage the voltage at the signal decrease. A photo transistor reads around 10 in bright light and 240 in the dark.

One problem that may occur with transitive sensors is that the voltage drop across the resistor may not be large enough when the transistor is open. Some transitive devices only allow a small amount of current to flow through the transistor. A larger range for the sensor can be accomplished by putting a larger pull-up resistor. By having a larger resistor, the voltage drop across the pull-up resistor will be proportional to the resistance.

We will give example uses and mountings for each type of sensor. Keep in mind that these are only simple examples and are not the only possible uses for them. It's up to you to make creative use of the sensors you've been given.

5.5.3 Potentiometers

Kits contain several sizes of potentiometers, also known as pots or variable resistors. There are rotary and linear pots. As the knob is turned or the handle slid, the resistance increases or decreases. This will produce different analog values.

Potentiometers should be wired with Vcc and ground on the two outside pins, and the signal wire on the center tap. This will, in effect, place the resistance of the potentiometer in parallel with the 47KΩpull-up on the expansion board and is more stable than just using one side and the center tab to make a plain variable resistor.

Potentiometers have a variety of uses. In the past, they have been used for menuing programs and angle measurement for various rotating limbs or scanning beacons. They can be used with a motor to mimic servos, but that's a difficult task. It is important to notice that the pots are not designed to turn more than about 270 degrees. Forcing them farther is likely to break them.

A potentiometer can be attached to a LEGO beam such that it can be used in place of a bend sensor. The rotation of the beam will produce a rotation in the potentiometer. See if you can come up with an assembly that can be used in place of a bend sensor. The advantage to such a sensor is that it is much sturdier than the bend sensor. The disadvantage is that it is bulkier.

Linear Pots

A linear potentiometer can be used to measure precise linear motion, such as a gate closing, or a cocking mechanism for firing balls or blocks.

Frob-knob

The frob knob is the small white dial on the lower left corner of the Expansion Board. It returns values between 0 and 255 and provides a handy user input for adjusting parameters on the fly or for menuing routines to select different programs. You may find it useful to glue a small LEGO piece to the frob knob to make turning it easier.

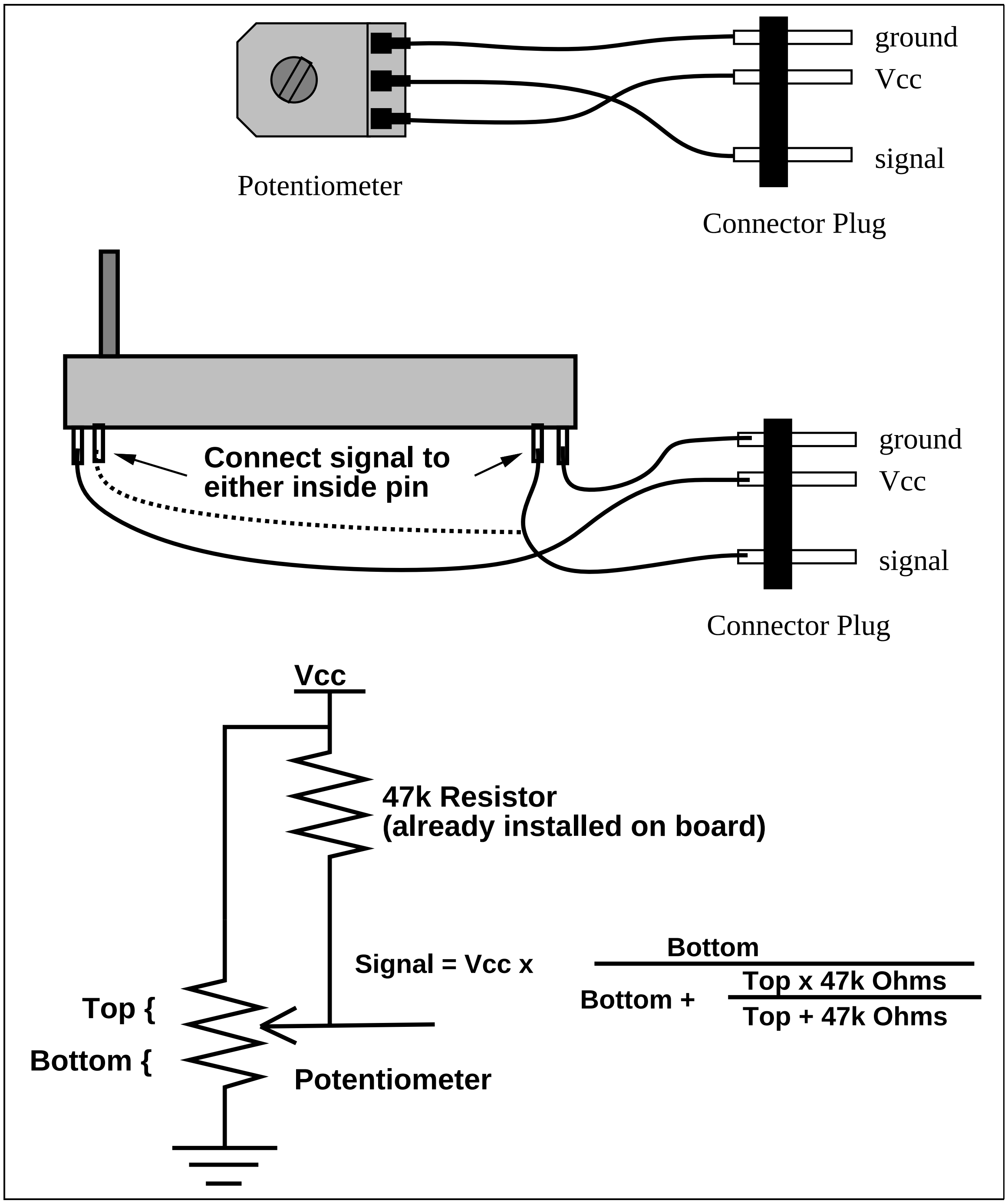

ground
Vcc
signal
Potentiometer
Connector Plug
Connect signal to either inside pin
ground
Vcc
signal
Connector Plug
Vcc
47k Resistor (already installed on board)
Signal = Vcc x Bottom / (Bottom + (Top x 47k Ohms)/(Top + 47k Ohms))
Top {
Bottom {
Potentiometer

Figure 5.5: Potentiometer Assemblies

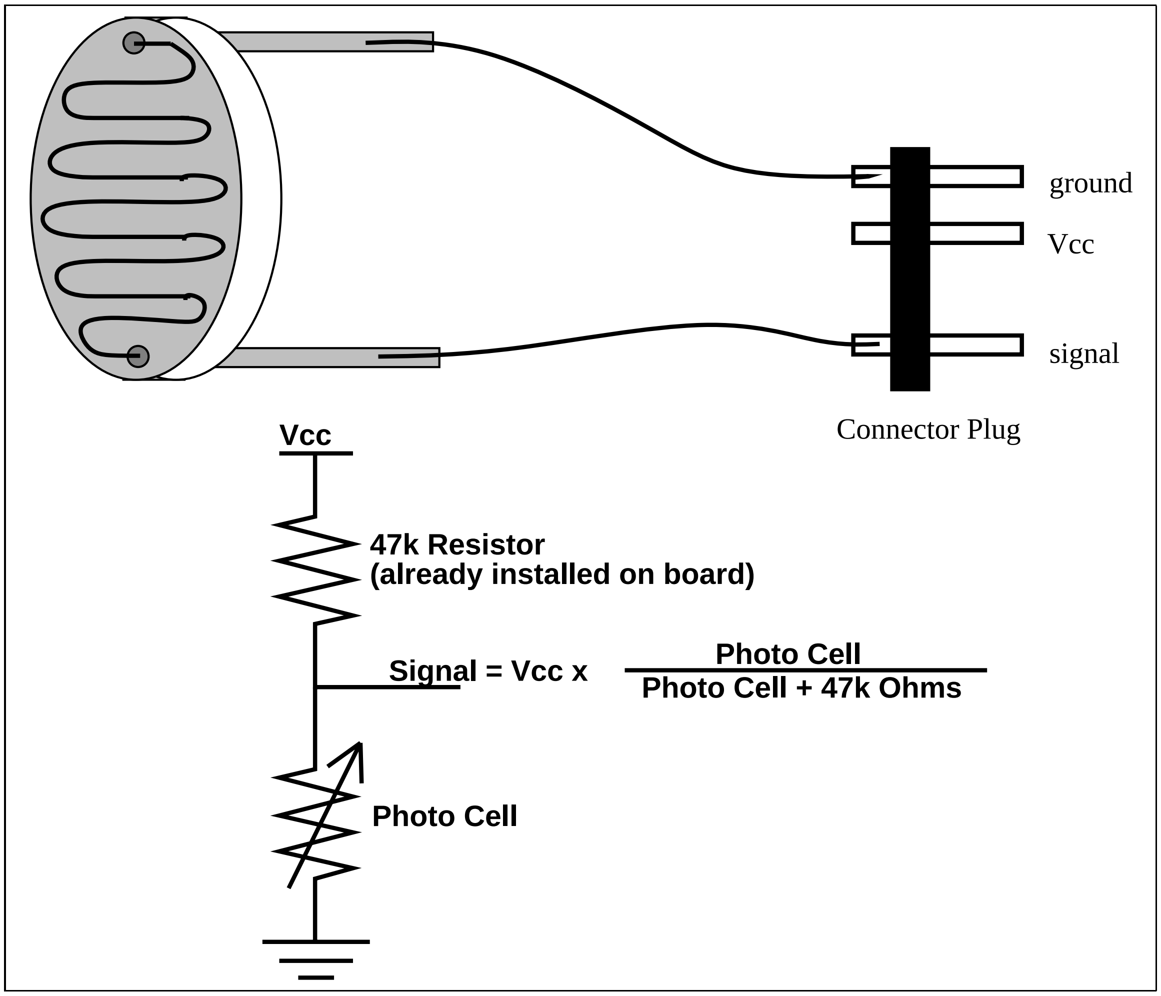

ground
Vcc
signal
Connector Plug
Vcc
47k Resistor
(already installed on board)
Signal = Vcc x Photo Cell / Photo Cell + 47k Ohms
Photo Cell

Figure 5.6: Photocell Light Sensor

5.5.4 Photoresistors

The photocell is a special type of resistor which responds to light. The more light hitting the photocell, the lower the resistance it has. The output signal of the photocell is an analog voltage corresponding to the amount of light hitting the cell. Higher values correspond to less light.

A photoresistor changes its resistive value based on the amount of light that strikes it. As the light hitting it increases, the resistive value decreases. They are somewhat sensitive to heat, but stand up to abuse well. Try not to overheat when soldering wires to them.

As with all the light-sensing devices, *shielding* is *very* important. A properly shielded sensor can make the difference between valid and invalid values reported by that sensor. The idea is simple: restrict the amount of light striking the sensor to the direction you expect the light to be coming from. You do not want light from external sources (i.e., camera flashes or spot lights) to interfere with your robot. Black heat shrink tubing often works well to shield the photoresistor from external light sources.

One good way to get a feel for how these sensors work, and how your robot and software interact, is to make a light-sniffer. With two or more photoresistors, try to create a simple robot that can move around a room, either avoiding light or avoiding shadows in a controlled manner. Ambient light conditions play a major role in how to interpret the data from any light sensors. A combination of photocells, one pointed up and one pointed down, may be used to adjust for ambient light levels, which may be useful in some applications.

Photoresistors are probably the only sensor required to be on your robot. A starting light will be used to start each contest round, and the robot must be able to sense that light. You must place one photoresistor on the underside of your robot, probably near the center. Be sure to shield it as much as possible from the overhead ambient light. We will provide starting code that reads the value of that sensor to start the match.

Mounting the photoresistors doesn't tend to be difficult. You can use a small amount of hot glue to attach the photocell to a LEGO brick, or double-sticky tape will also work. Be inventive.

5.5.5 Photo Transistors and IR LEDs

Phototransistors are usually tuned to a specific wavelength of light. The wavelength is usually near visible red, or in the infrared spectrum. They have similar properties to the photoresistors. The main difference is that the phototransistors are usually tuned to a specific wavelength. The other important difference is that the time delay for a change in light conditions is much smaller for a phototransistor. This can be useful in doing fast control looking for polarized light. The time constant for a phototransistor is much faster than a photoresistor, so it may be used in situations where timing is critical.

An IR LED is a type of diode which emits radiation in the infrared range. This part could be used as a component in a breakbeam sensor or a reflectance sensor.

These instructions are for two kinds of phototransistors, each of which are packaged in cylindrical brass-colored cans with a glass lens. The first kind is packaged individually, with no wires attached, and with three leads. The second is surplus parts, with wires already attached, and with each phototransistor paired with an LED. (Note: surplus parts are usually overstocked or obsolete parts that didn't sell through retail channels. See the book's appendix on ordering electronics parts.) The individual Phototransistors cost 6.270 about $1 each, about the same as an entire surplus assembly bundle of wires and phototransistors and LEDs.

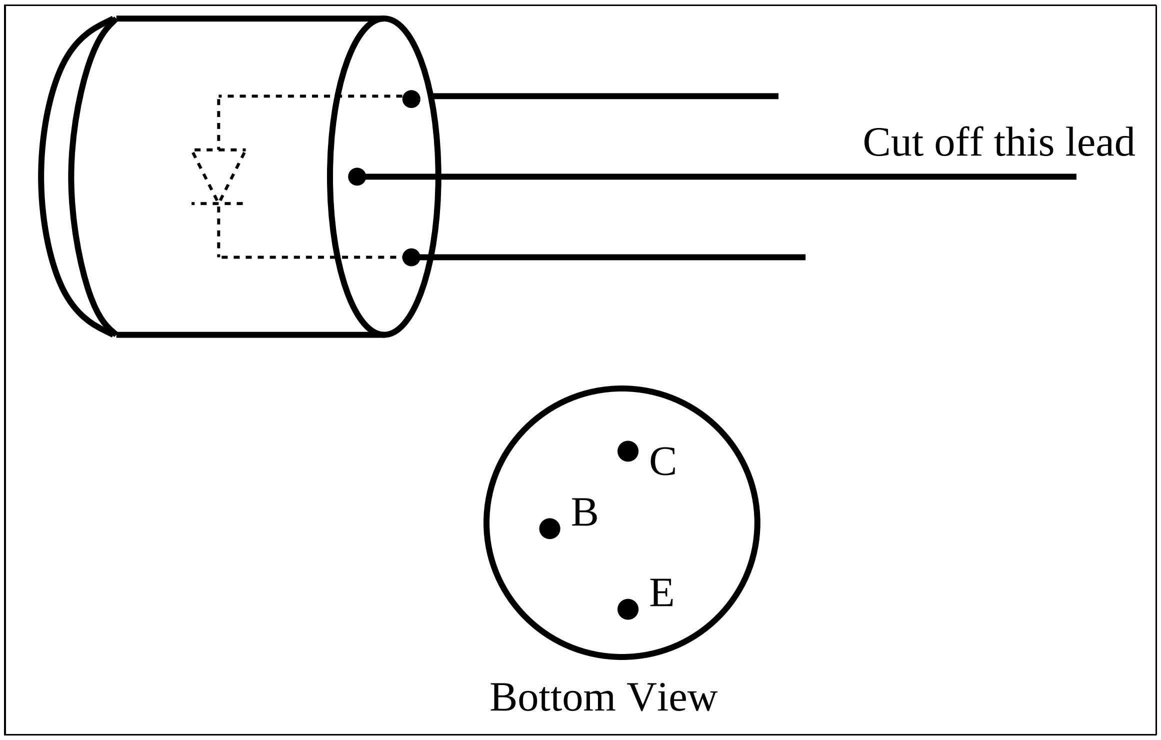


Figure 5.7: The LEDs

How to tell them apart

Be careful to differentiate the phototransistors from the LEDs: the phototransistors have relatively flat lenses, while the LEDS lenses are more convex. Fig 5.7 shows one of the LEDs. Also, the two different kinds of phototransistors (surplus vs virgin manufactured) have very different characteristics, and cannot be used in sensors interchangeably. The surplus phototransistors respond almost exclusively to infrared light and have a "resistance" of approximately 100 kΩ when activated and 1 MΩ when not activated. The individual, un-wired phototransistors, on the other hand, respond to visible light as well as infrared, and have "resistances" about one hundred times smaller.

Interfacing to the Board

These phototransistors require pull-up resistors, a resistor connected between Vcc and the signal line, to work properly. In past years, all of these sensors required 47k pull-up resistors, but that is no longer the case. Each individually packaged phototransistor now can be used with a 220k pull-up, while the "bundle of wires" phototransistors work well with 100Ωpull-ups. This may present A slight problem if you have already installed **RP6**, one of the 47k pull-up resistor packs on your expansion boards. Fear not! By installing the pull-up resistors on the connector as shown in Fig 5.8. For the individually packaged phototransistors, the 2.2k resistor on the connector will be in parallel with the 47k pull-up on the board. Since resistors in parallel add reciprocally, the combination of the two will electrically look like a 2.2k resistor (approximately). However, if you have the "bundle of wires" phototransistors, you will have to cut a trace on the bottom side of the expansion board to disable the 47k pull-up resistor, since it would otherwise dominate. Warning! Once you cut a trace, that analog port should be used only for the 220k phototransistors. This means that you will have to be sure to plug these sensors into the correct analog ports each time you use them. Ask a TA before you cut this trace!

Visible Light sensor

The phototransistors respond very well to visible (far-red, we hypothesize) light as well as infrared. They should be wired with a 2k to 4k resistor for best results (we recommend 2.2k). Because they respond to visible light, they are extremely susceptible to interference from ambient light. You may be able to use them as floor-color sensors using just ambient light, but if you want to use them for break-beam sensing, they will have to be very well-shielded.

IR Photo Transistor

The "bundle-of-wires" phototransistors are much more predictable. They should be wired with a resistor of 100k to 300k (we recommend 220k). They barely respond at all to visible frequencies of light. They respond particularly well to the LEDs with which they are bundled, as well as to the grey IR LEDs. Both LEDs are highly directional, and you should be able to get good break-beam results up to 5 or 6 cm apart (2 inches). This might prove especially useful in ball-firing mechanisms, for example. Note that both LEDs and phototransistors are just the right size to fit in LEGO axle-holes!

5.5.6 Polarizing Film

Polarized film has fine printed or etched straight lines. The polarizing film allows the light to travel in parallel perpendicular planes rather than in all directions. Assume for this section that the lines are running up and down, and therefore the light waves will be traveling up and down. If a second film is placed such that the lines are horizontal, the light traveling past the first filter will not pass through the second filter.

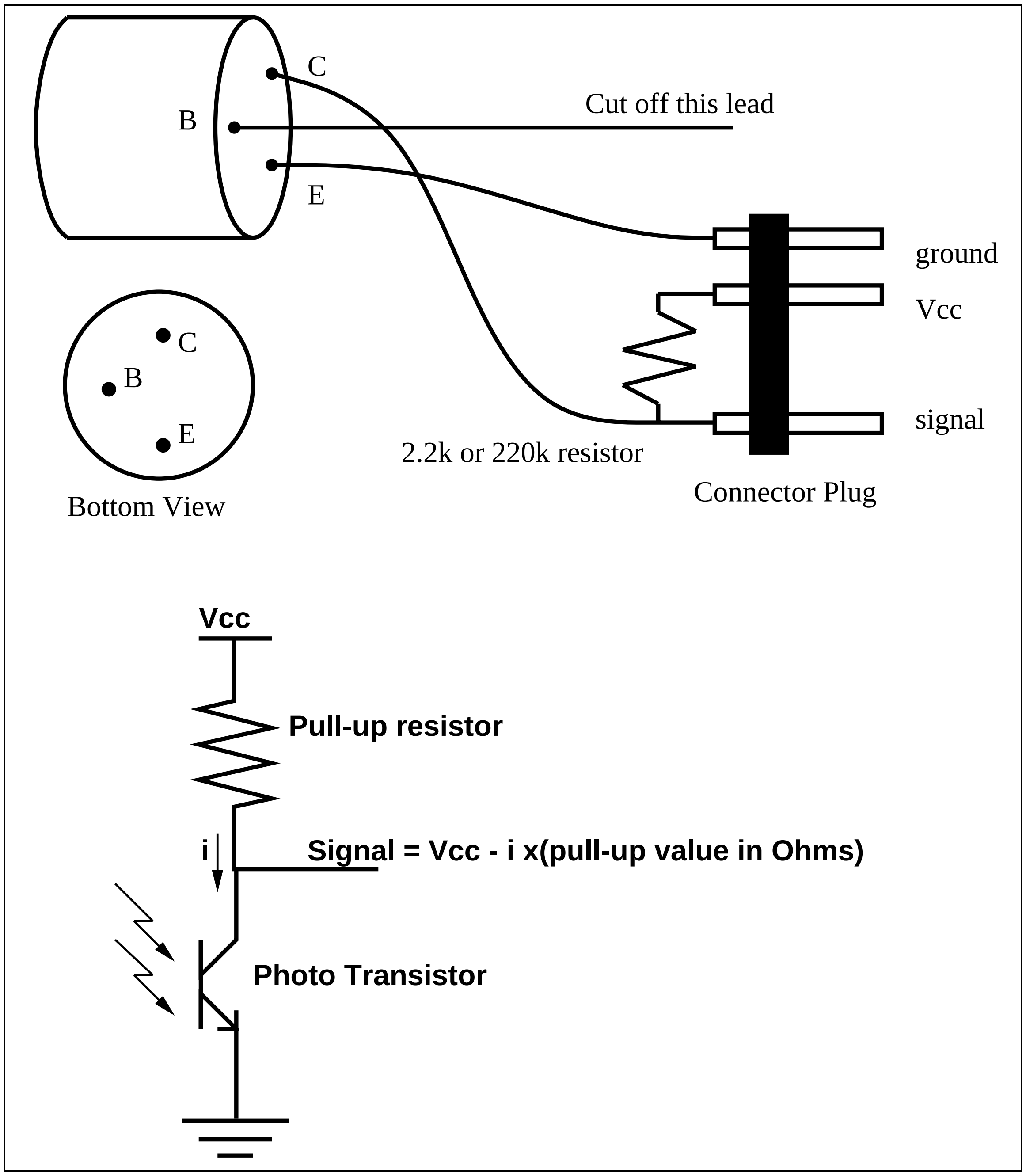


Figure 5.8: Phototransistor body and connector

Two pieces of film which are perpendicular to each other will block out most of the light. Parallel pieces will allow maximum light to go through.

The Polarizing film can be used to enhance the photo transistors and photo resistors. The beacons at each end of the playing field are emitting polarized light. One side is polarized at positive 45 degrees from the vertical and the other side is at negative 45 degrees. You can detect the difference between one side and the other by placing a piece of polarizing film in front of a phototransistor or photoresistor.

5.5.7 Reflectance Sensors

A reflectance sensor is made up of a combination of an infrared or red LED and a phototransistor that is sensitive to the wavelength of light being emitted by the LED. Over dark surfaces, the light is absorbed, whereas over light surfaces, the light is reflected back to the phototransistor.

A reflectance sensor (figure 5.9) can be made using discrete components.

The reflectance sensors are useful for detecting what color the floor is. They could also be used as object detectors, but they are very near sighted and quite responsive to outside lighting. In any application, good shielding is an absolute requirement if any reliability is desired. The sensors are very sensitive to distance from the reflecting surface. Distances greater than an inch will give very poor reading, and distances that are too small will not allow the right to be reflected. The angle at which the light is reflected to the surface is important and can produce better or worse results at different distances.

5.5.8 Motor-Force Sensors

There are four motor force sensors built into the 6.270 Controller board, attached to motors 0 through 3. These sensors are included to detect when the motors might be stalled. When the motors stall, they draw a large amount of current, which appears as a large voltage on the analog inputs to the 6811. When a motor_force value increases sharply, that's a good sign, but not guaranteed, that the motor may be stalled. The value that it reaches will depend on the load attached to the motor. Experiment by stalling the motor yourself while printing the values on the LCD to determine a threshold that's right for your robot.

Motor force values are not very accurate when you are driving the motors at anything less than 100%. Driving the motors at lower speeds will cause the motor force value to oscillate wildly, so it is recommended that you only use this information when you are driving a motor at full speed.

5.5.9 Breakbeam Sensors

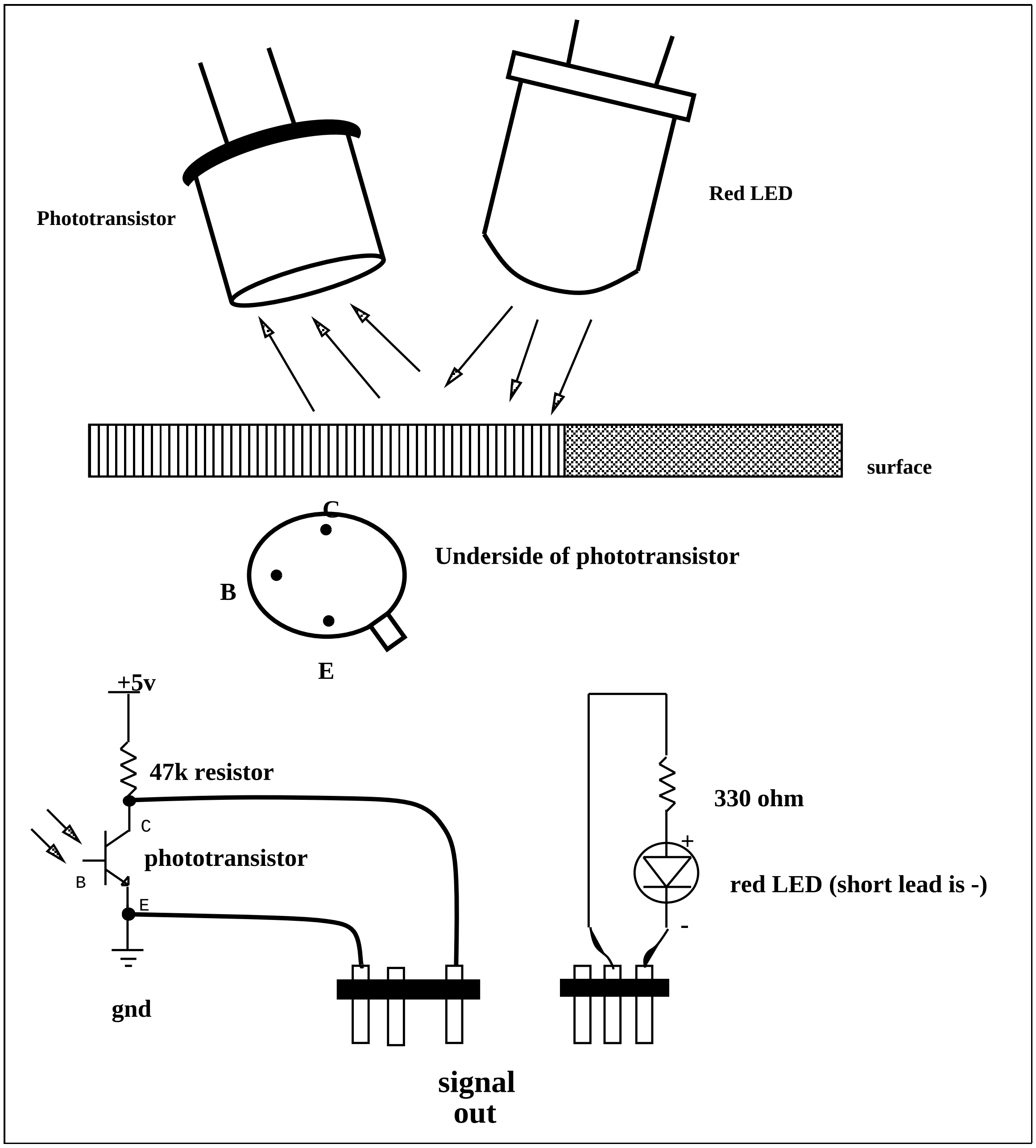


Figure 5.9: Reflectance Sensor

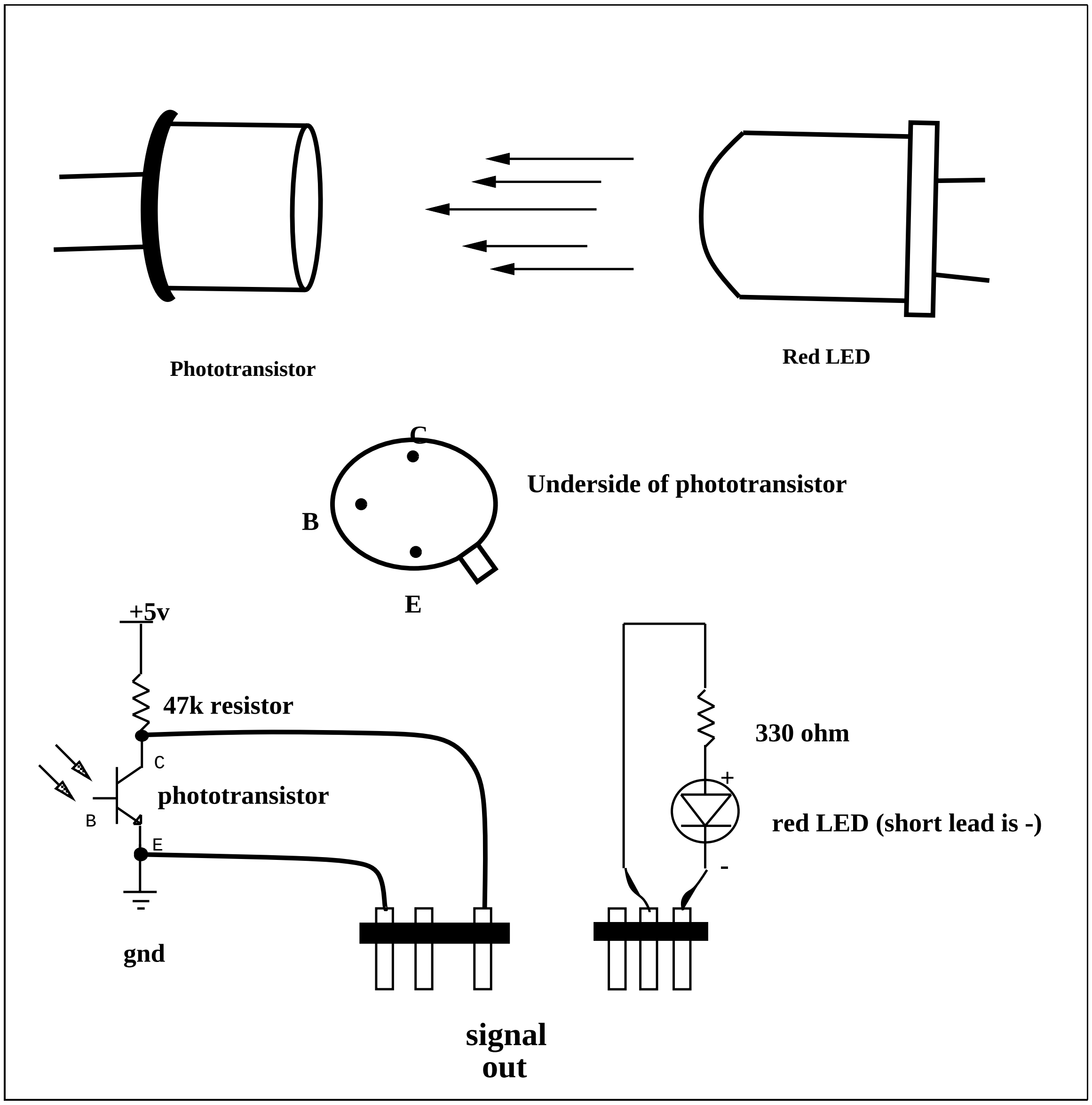


Figure 5.10: Breakbeam Sensor using discrete components.

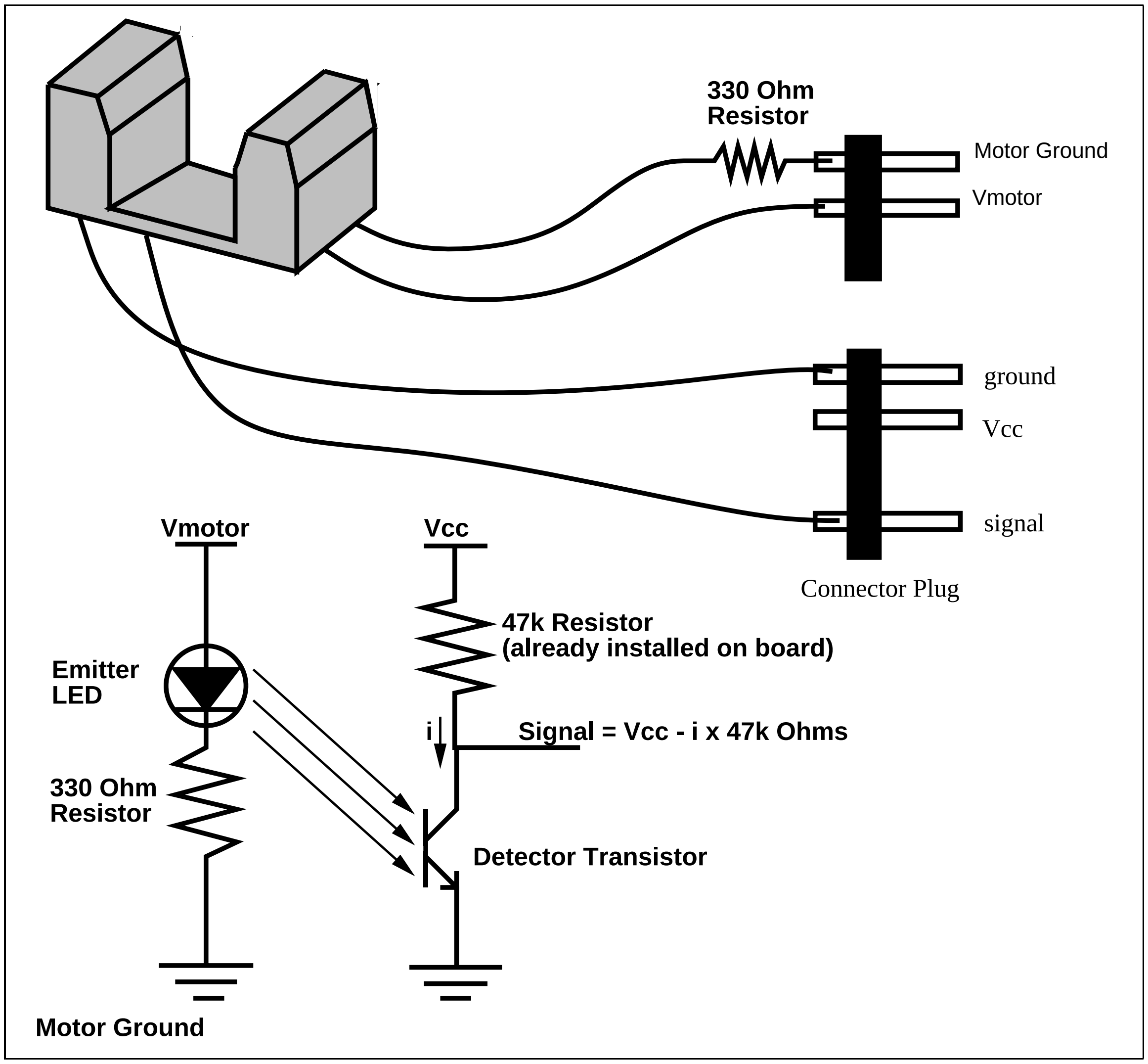


Figure 5.11: Breakbeam Assembly

Breakbeam sensors are another form of light sensors. Instead of looking for reflected light, the photosensor looks for direct light as shown in Figure 5.10.

The sensor is useful in detecting opaque objects that prevent the light beam from passing through. This can be useful in detecting block between gripper, or when block passes through a passageway. The sensor does not need to detect the block very quickly so the phototransistor can be plugged into the analog port.

The breakbeam sensors can also be used for counting holes or slots in a disk as it rotates (see Figure 5.12), allowing distance traveled to be measured. Since this requires a very fast sampling, the sampling needs to be done at the assembly language level. We have implemented shaft-encoder routines to do the fast sampling. But in order to use these routines the sensors should be plugged into the lower two digital ports if the rate at which the holes or slots go by is very high.

Before you use the analog sensors in the digital switch you must make sure that there is a full swing in the analog reading from when the light goes through to when the light is blocked.

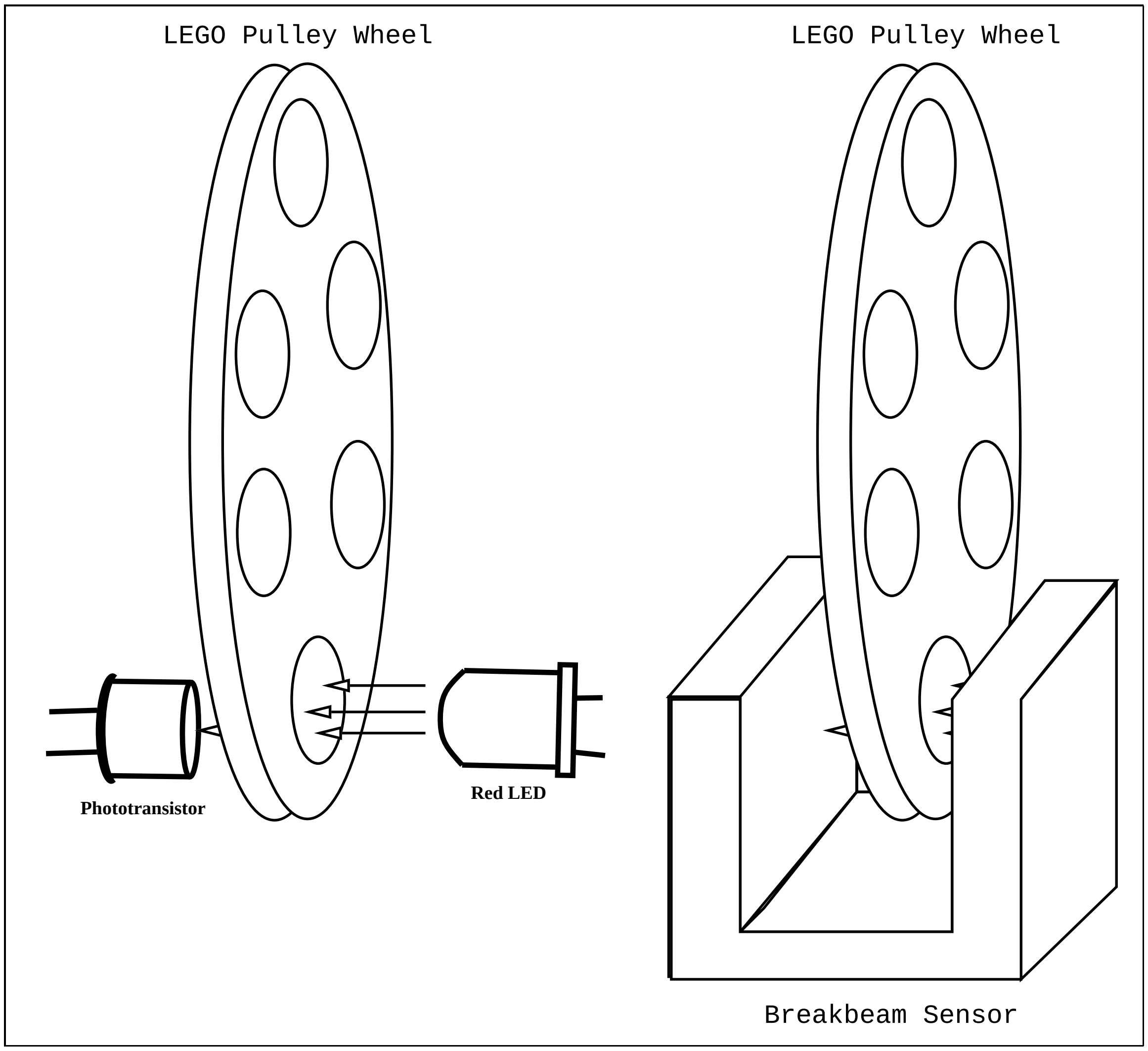


Figure 5.12: Shaft encoding using a LEGO pulley Wheel

Chapter 6

IC Manual

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

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

- **Interpreted execution** that allows run-time error checking. For example, IC does array bounds checking at run-time to protect against some programming errors.

- **Ease of design.** Writing a compiler for a stack machine is significantly easier than writing one for a typical processor. Since IC's p-code is machine-independent, porting IC to another processor entails rewriting the p-code interpreter, rather than changing the compiler.

- **Small object code.** Stack machine code tends to be smaller than a native code representation.

- **Multi-tasking.** Because the pseudo-code is fully stack-based, a process's state is defined solely by its stack and its program counter. It is thus easy to task-switch simply by loading a new stack pointer and program counter. This task-switching is handled by the run-time module, not by the compiler.

Since IC's ultimate performance is limited by the fact that its output p-code is interpreted, these advantages are taken at the expense of raw execution speed.

The current version of IC was designed and implemented by Randy Sargent, Anne Wright, and Carl Witty, with the assistance of Fred Martin. As of this writing, there are five related 6811 systems in use: the 1991 6.270 Board (the "Revision 2" board), the 1991 Sensor Robot, the Rev. 2.1 6.270 Board (from 1992), the Rev. 2.2 6.270 Board (from 1993), and the Rev. 2.21 6.270 Board (from 1994). This writing covers the Rev. 2.21 board only; documentation for the other systems is available elsewhere.

6.1 Getting Started

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

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

   `add 6.270`

2. **Plug the board into the computer.** Connect a modular phone cable between the board and the host computer. The connection at the host end depends on the type of the host. *Before the board is turned on*, check that the board's green LED (labelled SER RCV) is lit. If it is not lit, there is a problem with the connection.

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

   - Switch the board on. Hold down the CHOOSE button while hitting the reset button. The yellow LED (labelled SER XMIT) should turn off. *If the yellow LED is lit, the board is not ready to be initialized – try again.*
   - From the Unix prompt, type

     `init_bd`

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

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

   (a) The board will emit a brief beep;

(b) A version message will be printed on the LCD screen (e.g., "`IC vX.XX`");

(c) The yellow LED will turn on brightly.

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

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

   ic

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

6.2 Using IC

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

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

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

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

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

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

To evaluate a series of expressions, create a C block by beginning with an open curly brace "`{`" and ending with a close curly brace "`}`". The following example creates a local variable `i` and prints 10 (the sum of `i + 7`) to the 6811's LCD screen:

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

6.2.1 IC Commands

IC responds to the following commands:

- **Load file.** The command `load` <*filename*> compiles and loads the named file. The board must be attached for this to work. IC looks first in the local directory and then in the IC library path for files.

  Several files may be loaded into IC at once, allowing programs to be defined in multiple files.

- **Unload file.** The command `unload` <*filename*> unloads the named file, and re-downloads remaining files.

- **List files, functions, globals, or defines.** The command `list files` displays the names of all files presently loaded into IC. The command `list functions` displays the names of presently defined C functions. The command `list globals` displays the names of all currently defined global variables. The command `list defines` displays the names and values of all currently defined preprocessor macros.

- **Kill all processes.** The command `kill_all` kills all currently running processes.

- **Print process status.** The command `ps` prints the status of currently running processes.

- **Help.** The command `help` displays a help screen of IC commands.

- **Quit.** The command `quit` exits IC. CTRL-C can also be used.

6.2.2 Line Editing

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

To scan forward and backward in the command history, type CTRL-P or $\boxed{\uparrow}$ for backward, and CTRL-N or $\boxed{\downarrow}$ for forward.

An earlier line in the command history can be retrieved by typing the exclamation point followed by the first few characters of the line to retrieve, and then the space bar. For example, if you had previously typed the command `C> load foo.c`, then typing `C>!lo` followed by a space would retrieve the line `C> load foo`.

Figure 6.1 shows the keystroke mappings understood by IC.

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

Keystroke	**Function**
$\boxed{\text{DEL}}$	backward-delete-char
CTRL-A	beginning-of-line
CTRL-B	backward-char
$\boxed{\leftarrow}$	backward-char
CTRL-D	delete-char
CTRL-E	end-of-line
CTRL-F	forward-char
$\boxed{\rightarrow}$	forward-char
CTRL-K	kill-line
ESC D	kill-word
ESC $\boxed{\text{DEL}}$	backward-kill-word
$\boxed{\uparrow}$, CTRL-P	history-last
$\boxed{\downarrow}$, CTRL-N	history-next

Figure 6.1: IC Command-Line Keystroke Mappings

6.2.3 The `main()` Function

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

To reset the board *without* running the `main()` function (for instance, when hooking the board back to the computer), hold down the board's ESCAPE button while pressing reset. The board will reset without running `main()`.

6.3 A Quick C Tutorial

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

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

All functions must have a return type. Since `main` does not return a value, it uses `void`, the null type, as its return type. Other types include integers (`int`) and floating point numbers (`float`). This *function declaration* information must precede each function definition.

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

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

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

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

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

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

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

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

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

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

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

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

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

    h = sqrt((float)(square(a) + square(b)));

    return(h);
}
```

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

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

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

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

This concludes the brief C tutorial.

6.4 Variables, Constants, and Data Types

Variables and constants are the basic data objects in a C program. Declarations list the variables to be used, state what type they are, and may set their initial value.

6.4.1 Variables

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

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

Declaration

In C, variables can be declared at the top level (outside of any curly braces) or at the start of each block (a functional unit of code surrounded by curly braces). In general, a variable declaration is of the form:

```
<type> <variable name>;
or
<type> <variable name>=<initialization data>;
```

`<type>` can be `int, long, float, char,` or `struct <struct name>`, and determines the *primary type* of the variable declared. This form changes somewhat when dealing with pointer and array declarations, which are explained in a later section, but in general this is the way you declare variables.

Local and Global Scopes

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

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

Variable Initialization

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

All global variable declarations must be initialized to constant values. Local variables may be initialized to the value of arbitrary expressions including any globals, function calls, function arguments, or locals which have already been initialized.

Here is a small example of how initialized declarations are used.

```
int i=50;      /* declare i as global integer -- initial value 50 */
long j=100L;   /* declare j as global long -- initial value 100 */
int foo()
{
   int x;       /* declare x as local integer with initial value 0 */
   long y=j;    /* declare y as local integer with initial value j */
}
```

Local variables are initialized whenever the function containing them runs.

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

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

Persistent Global Variables

A special *persistent* form of global variable, has been implemented for IC . A persistent global may be initialized just like any other global, but its value is only initialized when the code is downloaded and not on any other reset conditions. If no initialization information is included for a persistent its value will be initialized to zero on download, but left unchanged on all other reset conditions.

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

```
persistent int i=500;
```

creates a global integer called `i` with the initial value 500.

Persistent variables keep their state when the board is turned off and on, when `main` is run, and when system reset occurs. Persistent variables will lose their state when code is downloaded as a result of loading or unloading a file. However, it is possible to read the values of your persistents in IC if you are still running the same IC session from which the code was downloaded. In this manner you could read the final values of calibration persistents, for example, and modify the initial values given to those persistents appropriately.

Persistent variables were created with two applications in mind:

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

6.4.2 Constants

Integers

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

Long Integers

Long integer constants are created by appending the suffix "`l`" or "`L`" (upper- or lower-case alphabetic L) to a decimal integer. For example, `0L` is the long zero. Either the upper or lower-case "L" may be used, but upper-case is the convention for readability.

Floating Point Numbers

Floating point numbers may use exponential notation (e.g., “`10e3`” or “`10E3`”) or may contain a decimal period. For example, the floating point zero can be given as “`0.`”, “`0.0`”, or “`0E1`”, but not as just “`0`”. *Since the 6811 has no floating point hardware, floating point operations are much slower than integer operations, and should be used sparingly.*

Characters and Character Strings

Quoted characters return their ASCII value (e.g., `'x'`).

Character string constants are defined with quotation marks, e.g., `"This is a character string."`.

NULL

The special constant NULL has the value of zero and can be assigned to and compared to pointer or array variables (which will be described in later sections). In general, you cannot convert other constants to be of a pointer type, so there are many times when NULL can be useful.

For example, in order to check if a pointer has been initialized you could compare its value to NULL and not try to access its contents if it was NULL. Also, if you had a defined a linked list type consisting of a value and a pointer to the next element, you could look for the end of the list by comparing the next pointer to NULL.

6.4.3 Data Types

IC supports the following data types:

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

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

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

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

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

Pointers IC pointers are 16-bit numbers which represent locations in memory. Values in memory can be manipulated by calculating, passing and *dereferencing* pointers representing the location where the information is stored.

Arrays Arrays are used to store homogeneous lists of data (meaning that all the elements of an array have the same type). Every array has a length which is determined at the time the array is declared. The data stored in the elements of an array can be set and retrieved in the same manner that other variables can be.

Structures Structures are used to store non-homogeneous but related sets of data. Elements of a structure are referenced by name instead of number and may be of any supported type. Structures are useful for organizing related data into a coherent format, reducing the number of arguments passed to functions, increasing the effective number of values which can be returned by functions, and creating complex data representations such as directed graphs and linked lists.

6.4.4 Pointers

The address where a value is stored in memory is known as the *pointer* to that value. It is often useful to deal with pointers to objects, but great care must be taken to insure that the pointers used at any point in your code really do point to valid objects in memory. Attempts to refer to invalid memory locations could corrupt your memory. Most computing environments that you are probably used to return helpful messages like "Segmentation Violation" or "Bus Error" on attempts to access illegal memory. However, no such safety net exists in the 6.270 system and invalid pointer dereferencing is very likely to go undetected and cause serious damage to your data, your program, or even the pcode interpreter.

Pointer Safety

In past versions of IC you could not return pointers from functions or have arrays of pointers. In order to facilitate the use of structures, these features have been added to the current version. With this change, the number of opportunities to misuse pointers have increased. However, if you follow a few simple precautions you should do fine.

First, you should always check that the value of a pointer is not equal to NULL (a special zero pointer) before you try to access it. Variables which are declared to be pointers are initialized to NULL, so many uninitialized values could be caught this way.

Second, you should never use the pointer to a local variable in a manner which could cause it to be accessed after the function in which it was declared terminates. When a function terminates the space where its values were being stored is recycled. Therefore not only may dereferencing such pointers return incorrect values, but assigning to those addresses could lead to serious data corruption. A good way to prevent this is to never return the address of a local variable from the function which declares it and never store those pointers in an object which will live longer than the function itself (a global pointer, array, or struct). Global variables and variables local to main will not move once declared and their pointers can be considered to be secure.

The type checking done by ic will help prevent many mishaps, but it will not catch all errors, so be careful.

Pointer Declaration and Use

A variable which is a pointer to an object of a given type is declared in the same manner as a regular object of that type, but with an extra $*$ in front of the variable name.

The value stored at the location the pointer refers to is accessed by using the $*$ operator before the expression which calculates the pointer. This process is known as dereferencing.

The address of a variable is calculated by using the & operator before that variable, array element, or structure element reference.

There are two main differences between how you would use a variable to be a given type and a variable declared as a pointer to that type.

For the following explanation, consider X and Xptr as defined as follows:

```
long X;
long *Xptr;
```

- Space Allocation – Declaring an object of a given type, as `X` is of type `long`, allocates the space needed to store that value. Because an IC long takes four bytes of memory, four bytes are reserved for the value of `X` to occupy. However, a pointer like `Xptr` does not have the same amount of space allocated for it that is needed for an object of the type it points to. Therefore it can only safely refer to space which has already been allocated for globals (in a special section of memory reserved for globals) or locals (temporary storage on the stack).

- Initial Value – It is always safe to refer to a non-pointer type, even if it hasn't been initialized. However pointers have to be specifically assigned to the address of legally allocated space or to the value of an already initialized pointer before they are safe to use.

So, for example, consider what would happen if the first two statements after X and Xptr were declared were the following:

```
X=50L;
*Xptr=50L;
```

The first statement is valid: it sets the value of `X` to 50L. The second statement would be valid if `Xptr` had been properly initialized, but in this case it is not. Therefore, this statement would corrupt memory.

Here is a sequence of commands you could try which illustrate how pointers and the and & operators are used. It also shows that once a pointer has been set to point at a place in memory, references to it actually share the same memory as the object it points to:

```
X=50L; /* set the memory allocated for X to the value 50 */
Xptr=&X; /* set Xptr to point to X */
*Xptr; /* see that the value pointed at by Xptr is 50 */
X=100L; /* set X to the value 100 */
*Xptr; /* see that the value pointed at by Xptr changed to 100 */
*Xptr=200L; /* set the value pointed at by Xptr to 200 */
X; /* see that the value in X changed to 200 */
```

Passing Pointers as Arguments

Pointers can be passed to functions and functions can change the values of the variables that are pointed at. This is termed *call-by-reference*; a reference, or pointer, to a variable is given to the function that is being called. This is in contrast to *call-by-value*, the standard way that functions are called, in which the value of a variable is given the to function being called.

The following example defines an `average_sensor` function which takes a port number and a pointer to an integer variable. The function will average the sensor and store the result in the variable pointed at by `result`.

Function arguments are declared to be pointers by prepending a star to the argument name, just as is done for other variable declarations.

```
    void average_sensor(int port, int *result)
    {
      int sum= 0;
      int i;

      for (i= 0; i< 10; i++) sum += analog(port);

      *result=  sum/10;
    }
```

Notice that the function itself is declared as a `void`. It does not need to return anything, because it instead stores its answer in the pointer variable that is passed to it.

The pointer variable is used in the last line of the function. In this statement, the answer `sum/10` is stored at the location pointed at by `result`. Notice that the asterisk is used to get assign a value to the *location* pointed by `result`.

Returning Pointers from Functions

Pointers can also be returned from functions. Functions are defined to return pointers by preceding the name of the function with a star, just like any other type of pointer declaration.

```
int right,left;

int *dirptr(int dir)
{
   if(dir==0) {
      return(&right);
   }
   if(dir==1) {
      return(&left);
   }
   return(NULL);
}
```

The function `dirptr` returns a pointer to the global `right` when its argument `dir` is 0, a pointer to `left` when its argument is 1, and NULL if its argument is other than 0 or 1.

6.4.5 Arrays

IC supports arrays of characters, integers, long integers, floating-point numbers, structures, pointers, and array pointers (multi-dimensional arrays). While unlike regular C arrays in a number of respects, they can be used in a similar manner. The main reasons that arrays are useful are that they allow you to allocate space for many instances of a given type, send an arbitrary number of values to functions, and iterate over a set of values.

Arrays in IC are different and incompatible with arrays in other versions of C. This incompatibility is caused by the fact that references to ic arrays are checked to insure that the reference is truly within the bounds of that array. In order to accomplish this checking in the general case, it is necessary that the size of the array be stored with the contents of the array. *It is important to remember that an array of a given type and a pointer to the same type are incompatible types in IC whereas they are largely interchangeable in regular C.*

Declaring and Initializing Arrays

Arrays are declared using square brackets. The following statement declares an array of ten integers:

```
int foo[10];
```

In this array, elements are numbered from 0 to 9. Elements are accessed by enclosing the index number within square brackets: `foo[4]` denotes the fifth element of the array `foo` (since counting begins at zero).

Arrays are initialized by default to contain all zero values. Arrays may also be initialized at declaration by specifying the array elements, separated by commas, within curly braces. If no size value is specified within the square brackets when the array is declared but initialization information is given, the size of the array is determined by the number of elements given in the declaration. For example,

```
int foo[]= {0, 4, 5, -8,  17, 301};
```

creates an array of six integers, with `foo[0]` equaling 0, `foo[1]` equaling 4, etc.

If a size is specified and initialization data is given, the length of the initialization data may not exceed the specified length of the array or an error results. If, on the other hand, you specify the size and provide fewer initialization elements than the total length of the array, the remaining elements are padded with zeros.

Character arrays are typically text strings. There is a special syntax for initializing arrays of characters. The character values of the array are enclosed in quotation marks:

```
char string[]= "Hello there";
```

This form creates a character array called `string` with the ASCII values of the specified characters. In addition, the character array is terminated by a zero. Because of this zero-termination, the character array can be treated as a string for purposes of printing (for example). Character arrays can be initialized using the curly braces syntax, but they will not be automatically null-terminated in that case. In general, printing of character arrays that are *not* null-terminated will cause problems.

Passing Arrays as Arguments

When an array is passed to a function as an argument, the array's pointer is actually passed, rather than the elements of the array. If the function modifies the array values, the array will be modified, since there is only one copy of the array in memory.

In normal C, there are two ways of declaring an array argument: as an array or as a pointer to the type of the array's elements. In IC array pointers are incompatible with pointers to the elements of an array so such arguments can only be declared as arrays.

As an example, the following function takes an index and an array, and returns the array element specified by the index:

```
int retrieve_element(int index, int array[])
{
    return array[index];
}
```

Notice the use of the square brackets to declare the argument `array` as a pointer to an array of integers.

When passing an array variable to a function, you are actually passing the value of the array pointer itself and not one of its elements, so no square brackets are used.

```
{
int array[10];

retrieve_element(3, array);
}
```

Multi-dimensional Arrays

A two-dimensional array is just like a single dimensional array whose elements are one-dimensional arrays. Declaration of a two-dimensional array is as follows:

```
int k[2][3];
```

The number in the first set of brackets is the number of 1-D arrays of `int`. The number in the second set of brackets is the length of each of the 1-D arrays of `int`. In this example, `k` is an array containing two 1-D arrays; `k[0]` is a 1-D array of `int` of length 3; `k[0][1]` is an `int`. Arrays of with any number of dimensions can be generalized from this example by adding more brackets in the declaration.

Determining the size of Arrays at Runtime

An advantage of the way IC deals with arrays is that you can determine the size of arrays at runtime. This allows you to do size checking on an array if you are uncertain of its dimensions and possibly prevent your program from crashing.

Since `_array_size` *is not a standard C feature, code written using this primitive will only be able to be compiled with IC .*

The `_array_size` primitive returns the size of the array given to it regardless of the dimension or type of the array. Here is an example of declarations and interaction with the _array_size primitive (don't worry about the multi dimensional arrays, they will be explained next section):

```
int i[4]={10,20,30};
int j[3][2]={{1,2},{2,4},{15}};
int k[2][2][2];

_array_size(i); /* returns 4 */
_array_size(j);  /* returns 3 */
_array_size(j[0]);  /* returns 2 */
_array_size(k); /* returns 2 */
_array_size(k[0]); /* returns 2 */
```

6.4.6 Structures

Structures are used to store non-homogeneous but related sets of data. Elements of a structure are referenced by name instead of number and may be of any supported type. Structures are useful for organizing related data into a coherent format, reducing the number of arguments passed to functions, increasing the effective number of values which can be returned by functions, and creating complex data representations such as directed graphs and linked lists.

The following example shows how to define a structure, declare a variable of structure type, and access its elements.

```
struct foo {
   int i;
   int j;
};

struct foo f1;

void set_f1(int i,int j)
{
   f1.i=i;
   f1.j=j;
}
void get_f1(int *i,int *j)
{
   *i=f1.i;
   *j=f1.j;
}
```

The first part is the structure definition. It consists of the keyword `struct`, followed by the name of the structure (which can be any valid identifier), followed by a list of named elements in curly braces. This definition specifies the structure of the type `struct foo`.

Once there is a definition of this form, you can use the type `struct foo` just like any other type. The line `struct foo f1;` is a global variable declaration which declares the variable `f1` to be of type `struct foo`.

The dot operator is used to access the elements of a variable of structure type. In this case, `f1.i` and `f1.j` refer to the two elements of `f1`. You can treat the quantities `f1.i` and `f1.j` just as you would treat any variables of type `int` (the type of the elements was defined in the structure declaration at the top to be `int`).

Pointers to structure types can also be used, just like pointers to any other type. However, with structures, there is a special short-cut for referring to the elements of the structure pointed to.

```
struct foo *fptr;

void main()
{
   fptr=&f1;
   fptr->i=10;
   fptr->j=20;
}
```

In this example, `fptr` is declared to be a pointer type type `struct foo`. In `main`, it is set to point to the global `f1` defined above. Then the elements of the structure pointed to by `fptr` (in this case these are the same as the elements of `f1`), are set. The arrow operator is used instead of the dot operator because fptr is a pointer to a variable of type `struct foo`. Note that `(*fptr).i` would have worked just as well as `fptr->i`, but it would have been clumsier.

Note that only pointers to structures, not the structures themselves, can be passed to or returned from functions.

6.4.7 Complex Initialization examples

Complex types – arrays and structures – may be initialized upon declaration with a sequence of constant values contained within curly braces and separated by commas. Arrays of character may also be initialized with a quoted string of characters.

For initialized declarations of single dimensional arrays, the length can be left blank and a suitable length based on the initialization data will be assigned to it. *Multi-dimensional arrays must have the size of all dimensions specified when the array is declared.* If a length is specified, the initialization data may not overflow that length in any dimension or an error will result. However, the initialization data may be shorter than the specified size and the remaining entries will be initialized to 0.

Following is an example of legal global and local variable initializations:

```
/* declare many globals of various types */
int i=50;

int *ptr=NULL;

float farr[3]={ 1.2, 3.6, 7.4 };
int tarr[2][4]={ { 1, 2, 3, 4 }, { 2, 4, 6, 8} };

char c[]=''Hi there how are you?'';
char carr[5][10]={``Hi'',''there'',''how'',''are'',''you''};

struct bar {
    int i;
    int *p;
    long j;} b={5, NULL, 10L};
struct bar barr[2] = { { 1, NULL, 2L }, { 3 } };

/* declare locals of various types */
int foo()
{
  int x;                 /* create local variable x
                            with initial value 0    */
  int y= tarr[0][2];     /* create local variable y
                            with initial value 3    */
  int *iptr=&i;          /* create a local pointer to integer
                            which points to the global i */
  int larr[2]={10,20};   /* create a local array larr
                            with elements 10 and 20 */
  struct bar lb={5,NULL,10L}; /* create a local variable of type
                            struct bar with i=5 and j=10 */
  char lc[]=carr[2];     /* create a local string lc with
                            initial value ``how'' */
  ...
}
```

6.5 Operators, Expressions, and Statements

Operators act upon objects of a certain type or types and specify what is to be done to them. Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

6.5.1 Operators

Each of the data types has its own set of operators that determine which operations may be performed on them.

Integers

The following operations are supported on integers:

- **Arithmetic.** addition `+`, subtraction `-`, multiplication `*`, division `/`.
- **Comparison.** greater-than `>`, less-than `<`, equality `==`, greater-than-equal `>=`, less-than-equal `<=`.
- **Bitwise Arithmetic.** bitwise-OR `|`, bitwise-AND `&`, bitwise-exclusive-OR `^`, bitwise-NOT `~`.
- **Boolean Arithmetic.** logical-OR `||`, logical-AND `&&`, logical-NOT `!`.

  When a C statement uses a boolean value (for example, `if`), it takes the integer zero as meaning false, and any integer other than zero as meaning true. The boolean operators return zero for false and one for true.

  Boolean operators `&&` and `||` may stop executing as soon as the truth of the final expression is determined. For example, in the expression `a && b`, if `a` is false, then `b` does not need to be evaluated because the result must be false. The `&&` operator therefore may not evaluate `b`.

Long Integers

A subset of the operations implemented for integers are implemented for long integers: arithmetic addition `+`, subtraction `-`, and multiplication `*`, and the integer comparison operations. Bitwise and boolean operations and division are not supported.

Floating Point Numbers

IC uses a package of public-domain floating point routines distributed by Motorola. This package includes arithmetic, trigonometric, and logarithmic functions.

The following operations are supported on floating point numbers:

- **Arithmetic.** addition `+`, subtraction `-`, multiplication `*`, division `/`.
- **Comparison.** greater-than `>`, less-than `<`, equality `==`, greater-than-equal `>=`, less-than-equal `<=`.
- **Built-in Math Functions.** A set of trigonometric, logarithmic, and exponential functions is supported, as discussed in Section 6.11 of this document.

Characters

Characters are only allowed in character arrays. When a cell of the array is referenced, it is automatically coerced into a integer representation for manipulation by the integer operations. When a value is stored into a character array, it is coerced from a standard 16-bit integer into an 8-bit character (by truncating the upper eight bits).

6.5.2 Assignment Operators and Expressions

The basic assignment operator is `=`. The following statement adds 2 to the value of `a`.

```
a = a + 2;
```

The abbreviated form

```
a += 2;
```

could also be used to perform the same operation.

All of the following binary operators can be used in this fashion:

```
+   -   *   /   %   <<   >>   &   ^   |
```

6.5.3 Increment and Decrement Operators

The increment operator "`++`" increments the named variable. For example, the statement "`a++`" is equivalent to "`a= a+1`" or "`a+= 1`".

A statement that uses an increment operator has a value. For example, the statement

```
a= 3;
printf("a=%d a+1=%d\n", a, ++a);
```

will display the text "`a=3 a+1=4`."

If the increment operator comes after the named variable, then the value of the statement is calculated *after* the increment occurs. So the statement

```
a= 3;
printf("a=%d a+1=%d\n", a, a++);
```

would display "`a=3 a+1=3`" but would finish with `a` set to 4.

The decrement operator "`--`" is used in the same fashion as the increment operator.

6.5.4 Data Access Operators

- `&`: A single ampersand preceding a variable, an array reference, or a structure element reference returns a pointer to the location in memory where that information is being stored. This should not be used on arbitrary expressions as they do not have a stable place in memory where they are being stored.
- `*`: A single star preceding an expression which evaluates to a pointer returns the value which is stored at that address. This process of accessing the value stored within a pointer is known as dereferencing.
- `[expr]`: an expression in square braces following an expression which evaluates to an array (an array variable, the result of a function which returns an array pointer, etc.) checks that the value of the expression falls within the bounds of the array and references that element.
- `.`: A dot between a structure variable and the name of one of its fields returns the value stored in that field.
- `->`: An arrow between a pointer to a structure and the name of one of its fields in that structure acts the same as a dot does, except it acts on the structure pointed at by it's left hand side. Where f is a structure of a type with e as an element name, the two expressions `f.i` and `(&f)->i` are equivalent.

6.5.5 Precedence and Order of Evaluation

The following table summarizes the rules for precedence and associativity for the C operators. Operators listed earlier in the table have higher precedence; operators on the same line of the table have equal precedence.

Operator	**Associativity**
`() []`	left to right
`! ~ ++ -- - (` *type* `)`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	right to left
`= += -=` etc.	right to left
`,`	left to right

6.6 Control Flow

IC supports most of the standard C control structures. One notable exception is the `switch` statement, which is not supported.

6.6.1 Statements and Blocks

A single C statement is ended by a semicolon. A series of statements may be grouped together into a *block* using curly braces. Inside a block, local variables may be defined.

6.6.2 If-Else

The `if else` statement is used to make decisions. The syntax is:

```
if ( expression )
  statement-1
else
  statement-2
```

expression is evaluated; if it is not equal to zero (e.g., logic true), then *statement-1* is executed.

The `else` clause is optional. If the `if` part of the statement did not execute, and the `else` is present, then *statement-2* executes.

6.6.3 While

The syntax of a `while` loop is the following:

`while (` *expression* `)`
  statement

`while` begins by evaluating *expression*. If it is false, then *statement* is skipped. If it is true, then *statement* is evaluated. Then the expression is evaluated again, and the same check is performed. The loop exits when *expression* becomes zero.

One can easily create an infinite loop in C using the `while` statement:

`while (1)`
  statement

6.6.4 For

The syntax of a `for` loop is the following:

`for (` *expr-1* `;` *expr-2* `;` *expr-3* `)`
  statement

This is equivalent to the following construct using `while`:

expr-1 `;`
`while (` *expr-2* `) {`
  statement
  expr-3 `;`
`}`

Typically, *expr-1* is an assignment, *expr-2* is a relational expression, and *expr-3* is an increment or decrement of some manner. For example, the following code counts from 0 to 99, printing each number along the way:

```
int i;
for (i= 0; i < 100; i++)
  printf("%d\n", i);
```

6.6.5 Break

Use of the `break` provides an early exit from a `while` or a `for` loop.

6.7 LCD Screen Printing

IC has a version of the C function `printf` for formatted printing to the LCD screen.

The syntax of `printf` is the following:

`printf(` *format-string* `, [` *arg-1* `] , ... , [` *arg-N* `] )`

This is best illustrated by some examples.

6.7.1 Printing Examples

Example 1: Printing a message. The following statement prints a text string to the screen.

```
printf("Hello, world!\n");
```

In this example, the format string is simply printed to the screen.

The character "`\n`" at the end of the string signifies *end-of-line.* When an end-of-line character is printed, the LCD screen will be cleared when a subsequent character is printed. Thus, most `printf` statements are terminated by a `\n`.

Example 2: Printing a number. The following statement prints the value of the integer variable `x` with a brief message.

```
printf("Value is %d\n", x);
```

The special form `%d` is used to format the printing of an integer in decimal format.

Example 3: Printing a number in binary. The following statement prints the value of the integer variable `x` as a binary number.

```
printf("Value is %b\n", x);
```

The special form `%b` is used to format the printing of an integer in binary format. Only the *low byte* of the number is printed.

Example 4: Printing a floating point number. The following statement prints the value of the floating point variable `n` as a floating point number.

```
printf("Value is %f\n", n);
```

The special form `%f` is used to format the printing of floating point number.

Example 5: Printing two numbers in hexadecimal format.

```
printf("A=%x  B=%x\n", a, b);
```

The form `%x` formats an integer to print in hexadecimal.

6.7.2 Formatting Command Summary

Format Command	Data Type	Description
`%d`	`int`	decimal number
`%x`	`int`	hexadecimal number
`%b`	`int`	low byte as binary number
`%c`	`int`	low byte as ASCII character
`%f`	`float`	floating point number
`%s`	`char array`	char array (string)

6.7.3 Special Notes

- The final character position of the LCD screen is used as a system "heartbeat." This character continuously blinks back and forth when the board is operating properly. If the character stops blinking, the board has failed.
- Characters that would be printed beyond the final character position are truncated.
- When using a two-line display, the `printf()` command treats the display as a single longer line.
- Printing of long integers is not presently supported.

6.8 Preprocessor

The preprocessor processes a file before it is sent to the compiler. The IC preprocessor allows definition of macros, and conditional compilation of sections of code. Using preprocessor macros for constants and function macros can make IC code more efficient as well as easier to read. Using `#if` to conditionally compile code can be very useful, for instance, for debugging purposes.

6.8.1 Preprocessor Macros

Preprocessor macros are defined by using the `#define` preprocessor directive at the start of a line. The scope of IC macros is to globals and functions. If a macro is defined anywhere in any of the files loaded into IC it can be used anywhere in any file. The following example shows how to define preprocessor macros.

```
#define RIGHT_MOTOR 0
#define LEFT_MOTOR  1

#define GO_RIGHT(power) (motor(RIGHT_MOTOR,(power)))
#define GO_LEFT(power)  (motor(LEFT_MOTOR,(power)))

#define GO(left,right) {GO_LEFT(left); GO_RIGHT(right);}

void main()
{
   GO(0,0);
}
```

Preprocessor macro definitions start with the `#define` directive at the start of a line, and continue to the end of the line. After `#define` is the name of the macro, such as `RIGHT_MOTOR`. If there is a parenthesis directly after the name of the macro, such as the `GO_RIGHT` macro has above, then the macro has arguments. The `GO_RIGHT` and `GO_LEFT` macros each take one argument. The `GO` macro takes two arguments. After the name and the optional argument list is the body of the macro.

Each time a macro is invoked, it is replaced with its body. If the macro has arguments, then each place the argument appears in the body is replaced with the actual argument provided.

Invocations of macros without arguments look like global variable references. Invocations of macros with arguments look like calls to functions. To an extent, this is how they act. However, macro replacement happens before compilation, whereas global references and function calls happen at run time. Also, function calls evaluate their arguments before they are called, whereas macros simply perform text replacement. For example, if the actual argument given to a macro contains a function call, and the macro instantiates its argument more than once in its body, then the function would be called multiple times, whereas it would only be called once if it were being passed as a function argument instead.

Appropriate use of macros can make IC programs more efficient and easier to read. It allows constants to be given symbolic names without requiring storage and access time as a global would. It also allows macros with arguments to be used in cases when a function call is desirable for abstraction, without the performance penalty of calling a function.

Macros defined in files can be used at the command line. Macros can also be defined at the command line to be used in interactively, but these will not affect loads or compilation. To obtain a list of the currently defined macros, type `list defines` at the IC prompt.

6.8.2 Conditional compilation

It is sometimes desirable to conditionally compile code. The primary example of this is that you may want to perform debugging output sometimes, and disable it at others. The IC preprocessor provides a convenient way of doing this by using the `#ifdef` directive.

```
void go_left(int power)
{
   GO_LEFT(power);
#ifdef DEBUG
   printf(``Going Left\n'');
   beep();
#endif
}
```

In this example, when the macro `DEBUG` is defined, the debugging message "Going Left" will be printed and the board will beep each time `go_left` is called. If the macro is not defined, the message and beep will not happen. Each `#ifdef` must be followed by an `#endif` at the end of the code which is being conditionally compiled. The macro to be checked can be anything, and `#ifdef` blocks may be nested.

Unlike regular C preprocessors, macros cannot be conditionally defined. If a macro definition occurs inside an `#ifdef` block, it will be defined regardless of whether the `#ifdef` evaluates to true or false. The compiler will generate a warning if macro definitions occur within an `#ifdef` block.

The `#if`, `#else`, and `#elif` directives are also available, but are outside the scope of this document. Refer to a C reference manual for how to use them.

6.8.3 Comparison with regular C preprocessors

The way in which IC deals with loading multiple files is fundamentally different from the way in which it is done in standard C. In particular, when using standard C, files are compiled completely independently of each other, then linked together. In IC on the other hand, all files are compiled together. This is why standard C needs function prototypes and `extern` global definitions in order for multiple files to share functions and globals, while IC does not.

In a standard C preprocessor, preprocessor macros defined in one C file cannot be used in another C file unless defined again. Also, the scope of macros is only from the point of definition to the end of the file. The solution then is to have the prototypes, `extern` declarations, and macros in header files which are then included at the top of each C file using the `#include` directive. This style interacts well with the fact that each file is compiled independent of all the others.

However, since declarations in IC do not file scope, it would be inconsistent to have a preprocessor with file scope. Therefore, for consistency it was desirable to give IC macros the same behavior as globals and functions. Therefore, preprocessor macros have global scope. If a macro is defined anywhere in the files loaded into IC it is defined everywhere. Therefore, the `#include` and `#undef` directives did not seem to have any appropriate purpose, and were accordingly left out.

The fact that `#define` directives contained within `#if` blocks are defined regardless of whether the `#if` evaluates to be true or false is a side effect of making the preprocessor macros have global scope.

Other than these modifications, the IC preprocessor should be compatible with regular C preprocessors.

6.9 The IC Library File

Library files provide standard C functions for interfacing with hardware on the robot controller board. These functions are written either in C or as assembly language drivers. Library files provide functions to do things like control motors, make tones, and input sensors values.

IC automatically loads the library file every time it is invoked. Depending on which 6811 board is being used, a different library file will be required. IC may be configured to load different library files as its default; for the purpose of the 6.270 contest, the on-line version of IC will be configured appropriately for the board that is in use.

As of this writing, there are five related 6811 systems in use: the 1991 6.270 Board (the "Revision 2" board), the 1991 Sensor Robot, the Rev. 2.1 6.270 Board (from 1992), the Rev. 2.2 6.270 Board (from 1993), and the Rev. 2.21 6.270 Board (from 1994). This writing covers the Rev. 2.21 board only; documentation for the other systems is available elsewhere.

On the MIT Athena system, IC library files are located in the following directory:

```
/mit/6.270/lib/ic
```

To understand better how the library functions work, study of the library file source code is recommended. The main library file for the Rev. 2.2 6.270 Board is named `lib_r22.lis`.

6.9.1 Output Control

DC Motors

Motor ports are numbered from 0 to 5; ports for motors 0 to 3 are located on the Microprocessor Board while motors 4 and 5 are located on the Expansion Board.

Motor may be set in a "forward" direction (corresponding to the green motor LED being lit) and a "backward" direction (corresponding to the motor red LED being lit).

The functions `fd(int m)` and `bk(int m)` turn motor `m` on forward or backward, respectively, at full power. The function `off(int m)` turns motor `m` off.

The power level of motors may also be controlled. This is done in software by a motor on and off rapidly (a technique called *pulse-width modulation.* The `motor(int m, int p)` function allows control of a motor's power level. Powers range from `100` (full on in the forward direction) to `-100` (full on the the backward direction). The system software actually only controls motors to seven degrees of power, but argument bounds of -100 and $+100$ are used.

`void fd(int m)`

Turns motor `m` on in the forward direction. Example: `fd(3);`

`void bk(int m)`

Turns motor `m` on in the backward direction. Example: `bk(1);`

`void off(int m)`

Turns off motor `m`. Example: `off(1);`

`void alloff()`

`void ao()`

Turns off all motors. `ao` is a short form for `alloff`.

`void motor(int m, int p)`

Turns on motor `m` at power level `p`. Power levels range from `100` for full on forward to `-100` for full on backward.

Servo Motor

Servos are motors with internal position feedback which you can accurately command to a given orientation. Servos will actively seek to move to and remain at the orientation they are commanded to go to. Servos are useful for aiming sensors or moving actuators through a limited arc. They are generally able to sweep through about 180 degrees and no more.

Library routines allows control of a single servo motor. The servo motor has a three-wire connection: power, ground, and control. There is a dedicated connection for the servo on the main board at the top of the bank of connectors which are above and to the left of the main power switch. A three prong connector with ground on the left, power in the middle, and control on the right should be used to plug the servo into its connector. So long as you are sure to get power in the middle, the servo will not be damaged by plugging it in backwards, but will simply not work until it is plugged in properly.

The position of the servo motor shaft is controlled by a rectangular waveform that is generated on the A7 pin. The duration of the positive pulse of the waveform determines the position of the shaft. The acceptable width of the pulse varies for different models of servos, but is approximately 700 timer cycles minimum and 4000 timer cycles maximum, where the 6811's timer runs at 2MHz. The pulse is repeated approximately every 20 milliseconds.

`void servo_on()`

Turns the servo signal on. You must call this function before the servo will move.

`void servo_off()`

Turns servo signal off. The servo will no longer try to move to any particular position and will move freely. When you are not actively using the servo, turning it off will save power and processor cycles.

`int servo(int period)`

Sets the high time of the servo signal to `period` timer cycles so long as that falls within the acceptable range for the servo. Otherwise it truncates the value to the closest the servo is physically able to go to. It returns the thresholded version of the period you gave it. Remember that servos have a finite reaction time which, while very fast to human senses of time, is very slow to a processor. If you are resetting the servo angle in a tight loop it may well never catch up with you.

`int servo_rad(float angle)`

Sets the commanded orientation of the servo to approximately the angle in radians that it is given and returns the pulse width in timer counts which the servo was actually commanded with. The minimum pulse width is defined to be zero radians and the maximum is defined to be π radians.

`int servo_deg(float angle)`
Sets the commanded orientation of the servo to approximately the angle in degrees that it is given and returns the pulse width in timer counts which the servo was actually commanded with. The minimum pulse width is defined to be zero degrees and the maximum is defined to be 180 degrees.

`int radian_to_pulse(float angle)`
Converts the angle given in radians to the corresponding pulse width in timer counts. Input range is 0.0 to 3.14.

`int degree_to_pulse(float angle)`
Converts the angle given in degrees to the corresponding pulse width in timer counts. Input range is 0.0 to 180.0.

Unidirectional Drivers

LED Drivers There are two output ports located on the Expansion Board that are suitable for driving LEDs or other small loads. These ports draw their power from the motor battery and hence will only work when that battery is connected.

The following commands are used to control the LED ports:

`void led_out0(int s)`
Turns on LED0 port if `s` is non-zero; turns it off otherwise.

`void led_out1(int s)`
Turns on LED1 port if `s` is non-zero; turns it off otherwise.

Expansion Board Motor Ports Motor ports 4 and 5, located on the Expansion Board, may also be used to control unidirectional devices, such as a solenoid, lamp, or a motor that needs to be driven in one direction only. Each of the two motor ports, when used in this fashion, can independently control two such devices.

To use the ports unidirectionally, the two-pin header directly beneath the motor 4 and 5 LEDs is used.

`void motor4_left(int s)`
Turns on left side of motor 4 port if `s` is non-zero; turns it off otherwise.

`void motor4_right(int s)`
Turns on right side of motor 4 port if `s` is non-zero; turns it off otherwise.

`void motor5_left(int s)`
Turns on left side of motor 5 port if `s` is non-zero; turns it off otherwise.

`void motor5_right(int s)`
Turns on right side of motor 5 port if `s` is non-zero; turns it off otherwise.

6.9.2 Sensor Input

Digital Input

`int digital(int p)`
Returns the value of the sensor in sensor port `p`, as a true/false value (1 for true and 0 for false).

Sensors are expected to be *active low*, meaning that they are valued at zero volts in the active, or true, state. Thus the library function returns the inverse of the actual reading from the digital hardware: if the reading is zero volts or logic zero, the `digital()` function will return true.

`int dip_switch(int sw)`
Returns value of DIP switch `sw` on interface board. Switches are numbered from 1 to 4 as per labelling on actual switch. Result is 1 if the switch is in the position labelled "on," and 0 if not.

`int dip_switches()`
Returns value on DIP switches as a four-bit binary number. Left-most switch is most significant binary digit. "On" position is binary one.

`int choose_button()`
Returns value of button labelled CHOOSE: 1 if pressed and 0 if released.
Example:

```
/*  wait until choose button pressed  */
while (!choose_button()) {}
```

`int escape_button()`
Returns value of button labelled ESCAPE.
Example:

```
    /*  wait for button to be pressed; then
        wait for it to be released so that
        button press is debounced          */
    while (!escape_button()) {}
    while (escape_button()) {}
```

6.9.3 Analog Inputs

`int analog(int p)`
Returns value of sensor port numbered `p`. Result is integer between 0 and 255.

If the `analog()` function is applied to a port that is implemented digitally in hardware, then the value 255 is returned if the *hardware* digital reading is 1 (as if a digital switch is open, and the pull up resistors are causing a high reading), and the value 0 is returned if the *hardware* digital reading is 0 (as if a digital switch is closed and pulling the reading near ground).

Ports are numbered as marked on the Microprocessor Board and Expansion Board.

`int frob_knob()`
Returns a value from 0 to 255 based on the position of the potentiometer labeled FROB KNOB.

`int motor_force(int m)`
Returns value of analog input sensing current level through motor `m`. Result is integer between 0 and 255, but typical readings range from about 40 (low force) to 100 (high force).

The force-sensing circuitry functions properly only when motors are operated at full speed. The circuit returns invalid results when motors are pulse-width modulated because of spikes that occur in the feedback path.

The force-sensing circuitry is implemented for motors 0 through 3.

Infrared Subsystem

The infrared subsystem is composed of two parts: an infrared transmitter, and infrared receivers. Software is provided to control transmission frequency and detection of infrared light at two frequencies.

Infrared Transmission

`void ir_transmit_on()`
Enables transmission of infrared light through IR OUT port.

`void ir_transmit_off()`
Disables transmission of infrared light through IR OUT port.

`void set_ir_transmit_period(int period)`
Sets infrared transmission period. `period` determines the delay in half-microseconds between transitions of the infrared waveform. If `period` is set to 10,000, a frequency of 100 Hz. will be generated. If `period` is set to 8,000, a frequency of 125 Hz. will be generated. The decoding software is capable of detecting transmissions on either of these two frequencies only.

`void set_ir_transmit_frequency(int frequency)`
Sets infrared transmission frequency. `frequency` is measured in hertz.

Upon a reset condition, the infrared transmission frequency is set for 100 Hz. and is disabled.

Infrared Reception In a typical 6.270 application, one robot will be broadcasting infrared at 100 Hz. and will set its detection system for 125 Hz. The other robot will do the opposite. Each robot must physically shield its IR sensors from its own light; then each robot can detect the emissions of the other.

The infrared reception software employs a *phase-locked loop* to detect infrared signals modulated at a particular frequency. This program generates an internal squarewave at the desired reception frequency and attempts to lock this squarewave into synchronization with a waveform received by an infrared sensor. If the error between the internal wave and the external wave is below some threshold, the external wave is considered "detected." The software returns as a result the number of consecutive detections for each of the infrared sensor inputs.

While enabled, the infrared reception software requires a great deal of processor time. Therefore, it is desirable to disable the IR reception whenever it is not being actively used.

Up to four infrared sensors may be used. These are plugged into positions 4 through 7 of the digital input port. These ports and the remainder of the digital input port may be used without conflict for standard digital input while the infrared detection software is operating.

The following library functions control the infrared detection system:

`void ir_receive_on()`
Enables the infrared reception software. The default is disabled. When the software is enabled, between 20% and 30% of the 6811 processor time will be spent performing the detection function; therefore it should only be enabled if it is being

used. *You must wait at least 100 milliseconds after starting the reception before the data is valid.*

`void ir_receive_off()`
Disables the infrared reception software.

`void set_ir_receive_frequency(int f)`
Sets the operating frequency for the infrared reception software. `f` should be 100 for 100 Hz. or 125 for 125 Hz. Default is 100.

`int ir_counts(int p)`
Returns number of consecutive squarewaves at operating frequency detected from port `p` of the digital input port. Result is number from 0 to 255. `p` must be 4, 5, 6, or 7

Random noise can cause spurious readings of 1 or 2 detections. The return value of `ir_counts()` should be greater than three before it is considered the result of a valid detection. *You must wait at least 100 milliseconds after starting the reception before the* `ir_counts()` *data is valid.*

Shaft Encoders

Shaft encoders can be used to count the number of times a wheel spins, or in general the number of digital pulses seen by an input. Two types of shaft encoders can be made using 6.270 sensors: optical encoders which use optical switches whose beam is periodically broken by a slotted wheel, or magnetic encoders which uses hall effect sensors which change state when a magnet on a shaft rotates past.

Shaft encoders are implemented using the input timer capture feature on the 6811. Therefore processing time is only used when a pulse is actually being recorded, and even very fast pulses can be counted. Because digital ports 0 and 1 are the only two input capture channels available for use on the 6.270 board, only two channels of shaft encoding work well. It is possible to use ports 2 and 3 as well, but doing so uses a lot of CPU time.

The encoding software for the 6.270 board keeps a running count of the number of pulses each enabled encoder has seen. The number of counts is set to 0 when a channel is first enabled and when a user resets that channel. Because the counters are only 16-bits wide, they will overflow and the value will appear negative after 32,767 counts have been accumulated without a reset.

Shaft Encoder Files As shaft encoders are an optional feature and not part of the standard hardware of the 6.270 board, the library routines which read them are not loaded on start up.

In order to load the following routines for use in your programs, load the file `encoders.lis`. This file is in the standard 6.270 library directory so IC will find it by this name.

Shaft Encoder Routines The actions of the shaft encoders are commanded and the results are read using the following routines. The argument `encoder` to each of the routines specifies which shaft encoder the function should affect. This value should be 0 for digital port 0 or one for digital port 1. Arguments out of the range 0 to 1 have no useful effect.

`void enable_encoder(int encoder)`

Enables the given encoder to start counting pulses and resets its counter to zero. By default encoders start in the disabled state and must be enabled before they start counting.

`void disable_encoder(int encoder)`

Disables the given encoder and prevents it from counting. Each shaft encoder uses processing time every time it receives a pulse while enabled, so they should be disabled when you no longer need the encoder's data.

`void reset_encoder(int encoder)`

Resets the counter of the given encoder to zero. For an enabled encoder, it is more efficient to reset its value than to use `enable_encoder()` to clear it.

`int read_encoder(int encoder)`

Returns the number of pulses counted by the given encoder since it was enabled or since the last reset, whichever was more recent.

6.9.4 Time Commands

System code keeps track of time passage in milliseconds. Library functions are provided to allow dealing with time in milliseconds (using long integers), or seconds (using floating point numbers).

`void reset_system_time()`

Resets the count of system time to zero milliseconds.

`long mseconds()`

Returns the count of system time in milliseconds. Time count is reset by hardware reset (i.e., pressing reset switch on board) or the function `reset_system_time()`. `mseconds()` is implemented as a C primitive (not as a library function).

`float seconds()`

Returns the count of system time in seconds, as a floating point number. Resolution is one millisecond.

`void sleep(float sec)`

Waits for an amount of time equal to or slightly greater than `sec` seconds. `sec` is a floating point number.

Example:

```
/*  wait for 1.5 seconds  */
sleep(1.5);
```

`void msleep(long msec)`

Waits for an amount of time equal to or greater than `msec` milliseconds. `msec` is a long integer.

Example:

```
/*  wait for 1.5 seconds  */
msleep(1500L);
```

6.9.5 Tone Functions

Two simple commands are provided for producing tones on the standard beeper.

`void beep()`

Produces a tone of 500 Hertz for a period of 0.3 seconds. Returns when the tone is finished.

`void tone(float frequency, float length)`

Produces a tone at pitch `frequency` Hertz for `length` seconds. Returns when the tone is finished. Both `frequency` and `length` are floats.

In addition to the simple tone commands, the following functions can be used asynchronously to control the beeper driver.

`void set_beeper_pitch(float frequency)`

Sets the beeper tone to be `frequency` Hz. The subsequent function is then used to turn the beeper on.

`void beeper_on()`

Turns on the beeper at last frequency selected by the former function. The beeper remains on until the `beeper_off` function is executed.

`void beeper_off()`

Turns off the beeper.

6.9.6 Menuing and Diagnostics Functions

These functions are not loaded automatically, but they are available for you to use if you wish in the standard 6.270 library directory. They provide a standardized user interface for prompting users for input using the choose and select buttons and the frob knob. You may wish to use this library for debugging the state of your robot while away from the terminal or for changing thresholds or gains on the fly.

menu.c

Load `menu.c` to be able to use these functions.

`int select_string(char choices[][],int n)`

Interactively selects a string from an array of string (two-dimensional array of characters) of length n and returns an integer when a button is pressed. If the button pressed was the choose button, it returns the index into the array of the selected string, otherwise it returns -1. Example of use:

```
char a[3][14]={"Analog Port ","Digital Port ","Quit"};
int port,index=select_string(a,3);

if(index>-1 && index<2)
   port=select_int_value(a[index],0,27);
```

`int select_int_value(char s[],int min_val,int max_val)`
`float select_float_value(char s[],float min_val,float max_val)`

Interactively selects and returns a number between `min_val` and `max_val` which is selected by adjusting the frob knob until the appropriate value is displayed then pressing a button. If the escape button was pressed, returns -1 (or -1.0) regardless of the value chosen. Otherwise returns the chosen value. Remember that the frob knob only returns one of 255 values, so if the range is greater than that not all values will be possible choices.

`int chosen_button()`

Checks the user buttons and returns CHOOSE_B if the choose button is pressed, ESCAPE_B if the escape button is pressed, and NEITHER_B if neither button is pressed. If both buttons are pressed, the choose button has priority.

`int wait_button(int mode)`

Waits for either user button to execute the action appropriate to `mode` then returns which button was pressed. The choices for `mode` are: DOWN_B – wait until either button is pressed; UP_B – wait until no buttons are pressed; CYCLE_B – wait until a button is depressed and then all depressed buttons are released before returning.

diagnostic.c

Load `menu.c` and `diagnostic.c` to be able to use these functions. You can easily copy `diagnostic.c` and modify the `control_panel` function to call your own routines.

`void control_panel()`

General purpose control panel to let you view inputs, frob outputs, or set A to D thresholds. Pressing the escape button from the main menu or selecting "Quit" exits the control panel.

`int view_average_port(int port,int samples)`

Displays the analog reading of the given port until a button is pressed. If the button is the choose button, it then samples the reading at the given port, averages `samples` readings together, then prints and returns the average result. If the button pushed was the escape button, it returns -1.

`void view_inputs()`

General purpose input status viewer using the standard menuing routines to show digital inputs, analog inputs, frob knob, dip switches, and motor force inputs. Pressing escape at any time exits the viewer.

`void frob_outputs()`

General purpose output frobber. Uses the standard menuing routines to let you control the motors, led outputs, ir output, and the servo. Pressing escape from the main menu or selecting "Quit" exits the frobber.

6.10 Multi-Tasking

6.10.1 Overview

One of the most powerful features of IC is its multi-tasking facility. Processes can be created and destroyed dynamically during run-time.

Any C function can be spawned as a separate task. Multiple tasks running the same code, but with their own local variables, can be created.

Processes communicate through global variables: one process can set a global to some value, and another process can read the value of that global.

Each time a process runs, it executes for a certain number of *ticks*, defined in milliseconds. This value is determined for each process at the time it is created. The default number of ticks is five; therefore, a default process will run for 5 milliseconds until its "turn" ends and the next process is run. All processes are kept track of in a *process table*; each time through the table, each process runs once (for an amount of time equal to its number of ticks).

Each process has its own *program stack*. The stack is used to pass arguments for function calls, store local variables, and store return addresses from function calls. The size of this stack is defined at the time a process is created. The default size of a process stack is 256 bytes.

Processes that make extensive use of recursion or use large local arrays will probably require a stack size larger than the default. Each function call requires two stack bytes (for the return address) plus the number of argument bytes; if the function that is called creates local variables, then they also use up stack space. In addition, C expressions create intermediate values that are stored on the stack.

It is up to the programmer to determine if a particular process requires a stack size larger than the default. A process may also be created with a stack size *smaller* than the default, in order to save stack memory space, if it is known that the process will not require the full default amount.

When a process is created, it is assigned a unique *process identification number* or *pid*. This number can be used to kill a process.

6.10.2 Creating New Processes

The function to create a new process is `start_process`. `start_process` takes one mandatory argument—the function call to be started as a process. There are two optional arguments: the process's number of ticks and stack size. (If only one optional argument is given, it is assumed to be the ticks number, and the default stack size is used.)

`start_process` has the following syntax:

`int start_process(` *function-call(...)* `,` `[`*TICKS*`]` `,` `[`*STACK-SIZE*`]` `)` `start_process` returns an integer, which is the process ID assigned to the new process.

The function call may be any valid call of the function used. The following code shows the function `main` creating a process:

```
void check_sensor(int n)
{
  while (1)
    printf("Sensor %d is %d\n", n, digital(n));
}

void main()
{
  start_process(check_sensor(2));
}
```

Normally when a C functions ends, it exits with a return value or the "void" value. If a function invoked as a process ends, it "dies," letting its return value (if there was one) disappear. (This is okay, because processes communicate results by storing them in globals, not by returning them as return values.) Hence in the above example, the `check_sensor` function is defined as an infinite loop, so as to run forever (until the board is reset or a `kill_process` is executed).

Creating a process with a non-default number of ticks or a non-default stack size is simply a matter of using `start_process` with optional arguments; e.g.

```
start_process(check_sensor(2), 1, 50);
```

will create a `check_sensor` process that runs for 1 milliseconds per invocation and has a stack size of 50 bytes (for the given definition of `check_sensor`, a small stack space would be sufficient).

6.10.3 Destroying Processes

The `kill_process` function is used to destroy processes. Processes are destroyed by passing their process ID number to `kill_process`, according to the following syntax:

```
int kill_process(int pid)
```

`kill_process` returns a value indicating if the operation was successful. If the return value is `0`, then the process was destroyed. If the return value is `1`, then the process was not found.

The following code shows the `main` process creating a `check_sensor` process, and then destroying it one second later:

```
void main()
{
  int pid;

  pid= start_process(check_sensor(2));
  sleep(1.0);
  kill_process(pid);
}
```

6.10.4 Process Management Commands

IC has two commands to help with process management. The commands only work when used at the IC command line. They are not C functions that can be used in code.

`kill_all`
kills all currently running processes.

`ps`
prints out a list of the process status.

The following information is presented: process ID, status code, program counter, stack pointer, stack pointer origin, number of ticks, and name of function that is currently executing.

6.10.5 Process Management Library Functions

The following functions are implemented in the standard C library.

`void hog_processor()`
Allocates an additional 256 milliseconds of execution to the currently running process. If this function is called repeatedly, the system will wedge and only execute the process that is calling `hog_processor()`. Only a system reset will unwedge from this state. Needless to say, this function should be used with extreme care, and should not be placed in a loop, unless wedging the machine is the desired outcome.

`void defer()`
Makes a process swap out immediately after the function is called. Useful if a process knows that it will not need to do any work until the next time around the scheduler loop. `defer()` is implemented as a C built-in function.

6.11 Floating Point Functions

In addition to basic floating point arithmetic (addition, subtraction, multiplication, and division) and floating point comparisons, a number of exponential and transcendental functions are built in to IC:

`float sin(float angle)`
Returns sine of `angle`. Angle is specified in radians; result is in radians.

`float cos(float angle)`
Returns cosine of `angle`. Angle is specified in radians; result is in radians.

`float tan(float angle)`
Returns tangent of `angle`. Angle is specified in radians; result is in radians.

`float atan(float angle)`
Returns arc tangent of `angle`. Angle is specified in radians; result is in radians.

`float sqrt(float num)`
Returns square root of `num`.

`float log10(float num)`
Returns logarithm of `num` to the base 10.

`float log(float num)`
Returns natural logarithm of `num`.

`float exp10(float num)`
Returns 10 to the `num` power.

`float exp(float num)`
Returns e to the `num` power.

`(float) a ^ (float) b`
Returns `a` to the `b` power.

6.12 Memory Access Functions

IC has primitives for directly examining and modifying memory contents. These should be used with care as it is easy to corrupt memory and crash the system using these functions.

`int peek(int loc)`
Returns the byte located at address `loc`.

`int peekword(int loc)`
Returns the 16-bit value located at address `loc` and `loc`+1. `loc` has the most significant byte, as per the 6811 16-bit addressing standard.

`void poke(int loc, int byte)`
Stores the 8-bit value `byte` at memory address `loc`.

`void pokeword(int loc, int word)`
Stores the 16-bit value `word` at memory addresses `loc` and `loc`+1.

`void bit_set(int loc, int mask)`
Sets bits that are set in `mask` at memory address `loc`.

`void bit_clear(int loc, int mask)`
Clears bits that are set in `mask` at memory address `loc`.

6.13 Error Handling

There are two types of errors that can happen when working with IC: *compile-time* errors and *run-time* errors.

Compile-time errors occur during the compilation of the source file. They are indicative of mistakes in the C source code. Typical compile-time errors result from incorrect syntax or mismatching of data types.

Run-time errors occur while a program is running on the board. They indicate problems with a valid C form when it is running. A simple example would be a divide-by-zero error. Another example might be running out of stack space, if a recursive procedure goes too deep in recursion.

These types of errors are handled differently, as is explained below.

6.13.1 Compile-Time Errors

When compiler errors occur, an error message is printed to the screen. All compile-time errors must be fixed before a file can be downloaded to the board.

6.13.2 Run-Time Errors

When a run-time error occurs, an error message is displayed on the LCD screen indicating the error number. If the board is hooked up to IC when the error occurs, a more verbose error message is printed on the terminal.

Here is a list of the run-time error codes:

Error Code	**Description**
1	no stack space for `start_process()`
2	no process slots remaining
3	array reference out of bounds
4	stack overflow error in running process
5	operation with invalid pointer
6	floating point underflow
7	floating point overflow
8	floating point divide-by-zero
9	number too small or large to convert to integer
10	tried to take square root of negative number
11	tangent of 90 degrees attempted
12	log or ln of negative number or zero
15	floating point format error in printf
16	integer divide-by-zero

6.14 Binary Programs

With the use of a customized 6811 assembler program, IC allows the use of machine language programs within the C environment. There are two ways that machine language programs may be incorporated:

1. Programs may be called from C as if they were C functions.

2. Programs may install themselves into the interrupt structure of the 6811, running repetitiously or when invoked due to a hardware or software interrupt.

When operating as a function, the interface between C and a binary program is limited: a binary program must be given one integer as an argument, and will return an integer as its return value. However, programs in a binary file can declare any number of global integer variables in the C environment. Also, the binary program can use its argument as a pointer to a C data structure.

6.14.1 The Binary Source File

Special keywords in the source assembly language file (or module) are used to establish the following features of the binary program:

Entry point. The entry point for calls to each program defined in the binary file.

Initialization entry point. Each file may have one routine that is called automatically upon a reset condition. (The reset conditions are explained in Section 6.4.1, which discusses global variable initialization.) This initialization routine particularly useful for programs which will function as interrupt routines.

C variable definitions. Any number of two-byte C integer variables may be declared within a binary file. When the module is loaded into IC, these variables become defined as globals in C.

To explain how these features work, let's look at a sample IC binary source program, listed in Figure 6.2.

The first statement of the file (“`ORG MAIN_START`”) declares the start of the binary programs. This line must precede the code itself itself.

The entry point for a program to be called from C is declared with a special form beginning with the text `subroutine_`. In this case, the name of the binary program is `double`, so the label is named `subroutine_double`. As the comment indicates, this is a program that will double the value of the argument passed to it.

When the binary program is called from C, it is passed one integer argument. This argument is placed in the 6811's D register (also known as the “Double Accumulator”) before the binary code is called.

The `double` program doubles the number in the D register. The `ASLD` instruction (“Arithmetic Shift Left Double [Accumulator]”) is equivalent to multiplying by 2; hence this doubles the number in the D register.

The `RTS` instruction is “Return from Subroutine.” All binary programs must exit using this instruction. When a binary program exits, the value in the D register is the return value to C. Thus, the `double` program doubles its C argument and returns it to C.

Declaring Variables in Binary Files

The label `variable_foo` is an example of a special form to declare the name and location of a variable accessible from C. The special label prefix “`variable_`” is followed the name of the variable, in this case, “`foo`.”

This label must be immediately followed by the statement `FDB <number>`. This is an assembler directive that creates a two-byte value (which is the initial value of the variable).

```
/* Sample icb file */

/* origin for module and variables */
        ORG     MAIN_START

/* program to return twice the argument passed to us */
subroutine_double:
        ASLD
        RTS

/* declaration for the variable "foo" */
variable_foo:
        FDB     55

/* program to set the C variable "foo" */
subroutine_set_foo:
        STD     variable_foo
        RTS

/* program to retrieve the variable "foo" */
subroutine_get_foo:
        LDD     variable_foo
        RTS

/* code that runs on reset conditions */
subroutine_initialize_module:
        LDD     #69
        STD     variable_foo
        RTS
```

Figure 6.2: Sample IC Binary Source File: `testicb.asm`

Variables used by binary programs must be declared in the binary file. These variables then become C globals when the binary file is loaded into C.

The next binary program in the file is named "`set_foo`." It performs the action of setting the value of the variable `foo`, which is defined later in the file. It does this by storing the D register into the memory contents reserved for `foo`, and then returning.

The next binary program is named "`get_foo`." It loads the D register from the memory reserved for `foo` and then returns.

Declaring an Initialization Program

The label `subroutine_initialize_module` is a special form used to indicate the entry point for code that should be run to initialize the binary programs. This code is run upon standard reset conditions: program download, hardware reset, or running of the `main()` function.

In the example shown, the initialization code stores the value 69 into the location reserved for the variable `foo`. This then overwrites the 55 which would otherwise be the default value for that variable.

Initialization of globals variables defined in an binary module is done differently than globals defined in C. In a binary module, the globals are initialized to the value declared by the `FDB` statement only when the code is downloaded to the 6811 board (not upon reset or running of main, like normal globals).

However, the initialization routine is run upon standard reset conditions, and can be used to initialize globals, as this example has illustrated.

6.14.2 Interrupt-Driven Binary Programs

Interrupt-driven binary programs use the initialization sequence of the binary module to install a piece of code into the interrupt structure of the 6811.

The 6811 has a number of different interrupts, mostly dealing with its on-chip hardware such as timers and counters. One of these interrupts is used by the 6.270 software to implement time-keeping and other periodic functions (such as LCD screen management). This interrupt, dubbed the "System Interrupt," runs at 1000 Hertz.

Instead of using another 6811 interrupt to run user binary programs, additional programs (that need to run at 1000 Hz. or less) may install themselves into the System Interrupt. User programs would be then become part of the 1000 Hz interrupt sequence.

This is accomplished by having the user program "intercept" the original 6811 interrupt vector that points to 6.270 interrupt code. This vector is made to point at the user program. When user program finishes, it jumps to the start of the 6.270 interrupt code.

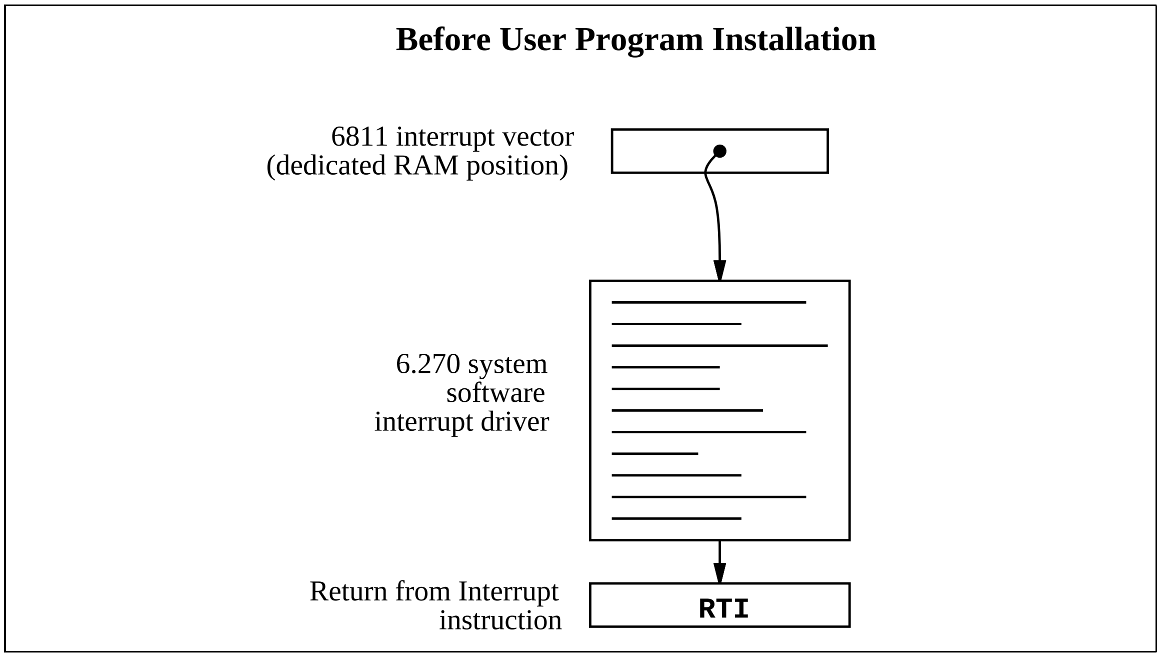


Figure 6.3: Interrupt Structure Before User Program Installation

Figure 6.3 depicts the interrupt structure before user program installation. The 6811 vector location points to system software code, which terminates in a "return from interrupt" instruction.

Figure 6.4 illustrates the result after the user program is installed. The 6811 vector points to the user program, which exits by jumping to the system software driver. This driver exits as before, with the RTI instruction.

Multiple user programs could be installed in this fashion. Each one would install itself ahead of the previous one. Some standard 6.270 library functions, such as the shaft encoder software, is implemented in this fashion.

Figure 6.5 shows an example program that installs itself into the System Interrupt. This program toggles the signal line controlling the piezo beeper every time it is run; since the System Interrupt runs at 1000 Hz., this program will create a continuous tone of 500 Hz.

The first line after the comment header includes a file named "`6811regs.asm`". This file contains equates for all 6811 registers and interrupt vectors; most binary programs will need at least a few of these. It is simplest to keep them all in one file that can be easily included. (This and other files included by the `as11` assembler are located in the assembler's default library directory, which is `/mit/6.270/lib/as11/` on the MIT Athena system.)

The `subroutine_initialize_module` declaration begins the initialization portion of the program. The file "`ldxibase.asm`" is then included. This file contains a few

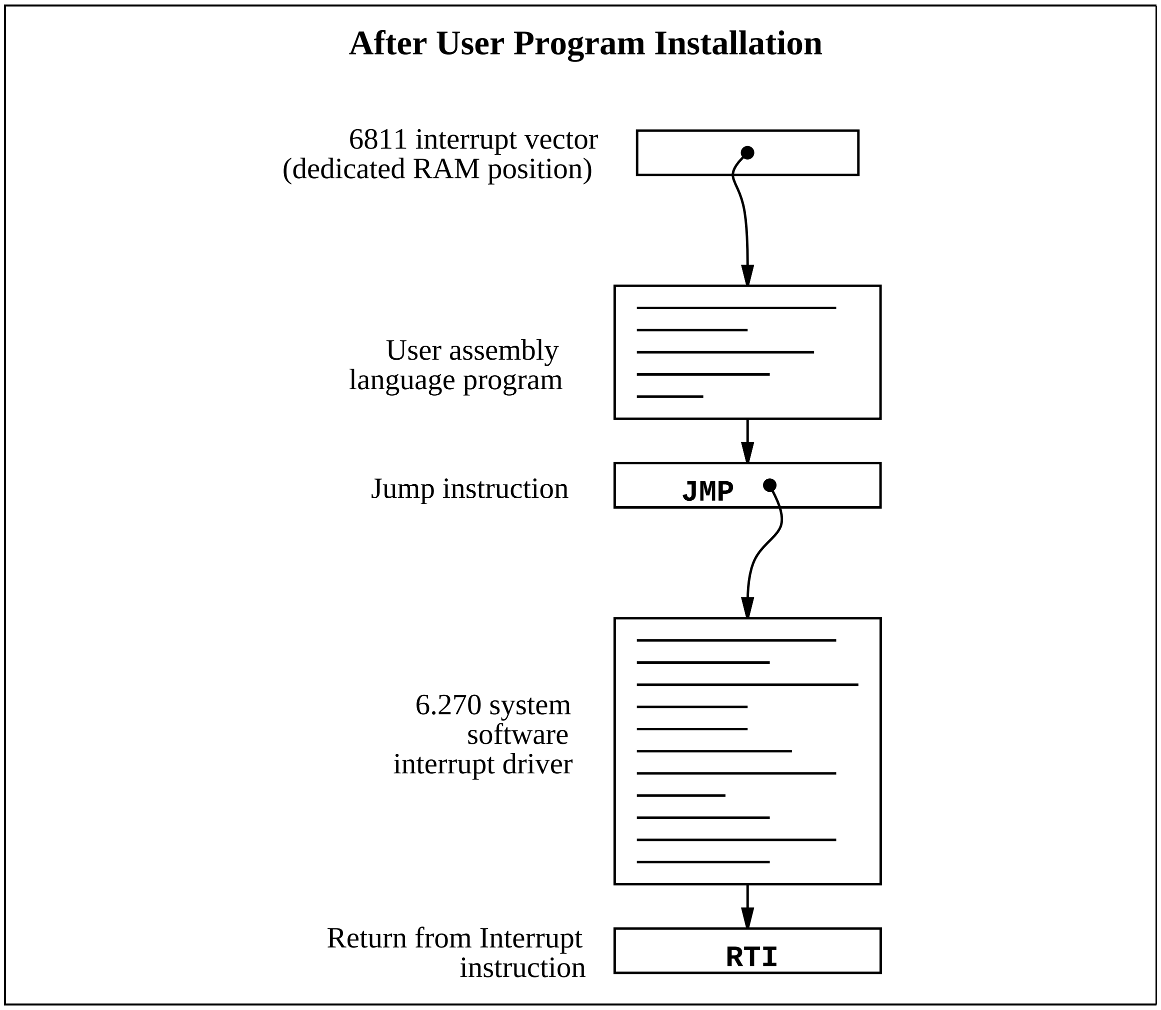

After User Program Installation
6811 interrupt vector (dedicated RAM position)
User assembly language program
Jump instruction
JMP
6.270 system software interrupt driver
Return from Interrupt instruction
RTI

Figure 6.4: Interrupt Structure After User Program Installation

```
* icb file:  "sysibeep.asm"

*
*   example of code installing itself into
*   SystemInt 1000 Hz interrupt
*
*   Fred Martin
*   Thu Oct 10 21:12:13 1991
*

#include <6811regs.asm>

        ORG     MAIN_START

subroutine_initialize_module:

#include <ldxibase.asm>
* X now has base pointer to interrupt vectors ($FF00 or $BF00)

* get current vector; poke such that when we finish, we go there
        LDD     TOC4INT,X            ; SystemInt on TOC4
        STD     interrupt_code_exit+1

* install ourself as new vector
        LDD     #interrupt_code_start
        STD     TOC4INT,X

        RTS

* interrupt program begins here
interrupt_code_start:
* frob the beeper every time called
        LDAA    PORTA
        EORA    #%00001000     ; beeper bit
        STAA    PORTA

interrupt_code_exit:
        JMP     $0000    ; this value poked in by init routine
```

Figure 6.5: `sysibeep.asm`: Binary Program that Installs into System Interrupt

lines of 6811 assembler code that perform the function of determining the base pointer to the 6811 interrupt vector area, and loading this pointer into the 6811 X register.

The following four lines of code install the interrupt program (beginning with the label `interrupt_code_start`) according to the method that was illustrated in Figure 6.4.

First, the existing interrupt pointer is fetched. As indicated by the comment, the 6811's TOC4 timer is used to implement the System Interrupt. The vector is poked into the JMP instruction that will conclude the interrupt code itself.

Next, the 6811 interrupt pointer is replaced with a pointer to the new code. These two steps complete the initialization sequence.

The actual interrupt code is quite short. It toggles bit 3 of the 6811's PORTA register. The PORTA register controls the eight pins of Port A that connect to external hardware; bit 3 is connected to the piezo beeper.

The interrupt code exits with a jump instruction. The argument for this jump is poked in by the initialization program.

The method allows any number of programs located in separate files to attach themselves to the System Interrupt. Because these files can be loaded from the C environment, this system affords maximal flexibility to the user, with small overhead in terms of code efficiency.

6.14.3 The Binary Object File

The source file for a binary program must be named with the `.asm` suffix. Once the `.asm` file is created, a special version of the 6811 assembler program is used to construct the binary object code. This program creates a file containing the assembled machine code plus label definitions of entry points and C variables.

```
S116802005390037FD802239FC802239CC0045FD8022393C
S9030000FC
S116872B05390037FD872D39FC872D39CC0045FD872D39F4
S9030000FC
6811 assembler version 2.1  10-Aug-91
  please send bugs to Randy Sargent (rsargent@athena.mit.edu)
  original program by Motorola.
subroutine_double 872b *0007
subroutine_get_foo 8733 *0021
subroutine_initialize_module 8737 *0026
subroutine_set_foo 872f *0016
variable_foo 872d *0012 0017 0022 0028
```

Figure 6.6: Sample IC Binary Object File: `testicb.icb`

The program `as11_ic` is used to assemble the source code and create a binary object file. It is given the filename of the source file as an argument. The resulting object file is automatically given the suffix `.icb` (for IC Binary). Figure 6.6 shows the binary object file that is created from the `testicb.asm` example file.

6.14.4 Loading an `icb` File

Once the `.icb` file is created, it can be loaded into IC just like any other C file. If there are C functions that are to be used in conjunction with the binary programs, it is customary to put them into a file with the same name as the `.icb` file, and then use a `.lis` file to loads the two files together.

6.14.5 Passing Array Pointers to a Binary Program

A pointer to an array is a 16-bit integer address. To coerce an array pointer to an integer, use the following form:

```
array_ptr= (int) array;
```

where `array_ptr` is an integer and `array` is an array.

When compiling code that performs this type of pointer conversion, IC must be used in a special mode. Normally, IC does not allow certain types of pointer manipulation that may crash the system. To compile this type of code, use the following invocation:

```
ic -wizard
```

Arrays are internally represented with a two-byte length value followed by the array contents.

6.15 IC File Formats and Management

This section explains how IC deals with multiple source files.

6.15.1 C Programs

All files containing C code must be named with the "`.c`" suffix.

Loading functions from more than one C file can be done by issuing commands at the IC prompt to load each of the files. For example, to load the C files named `foo.c` and `bar.c`:

```
C> load foo.c
C> load bar.c
```

Alternatively, the files could be loaded with a single command:

```
C> load foo.c bar.c
```

If the files to be loaded contain dependencies (for example, if one file has a function that references a variable or function defined in the other file), then the second method (multiple file names to one load command) or the following approach must be used.

6.15.2 List Files

If the program is separated into multiple files that are always loaded together, a "list file" may be created. This file tells IC to load a set of named files. Continuing the previous example, a file called `gnu.lis` can be created:

Listing of `gnu.lis`:

```
foo.c
bar.c
```

Then typing the command `load gnu.lis` from the C prompt would cause both `foo.c` and `bar.c` to be loaded.

6.15.3 File and Function Management

Unloading Files

When files are loaded into IC, they stay loaded until they are explicitly unloaded. This is usually the functionality that is desired. If one of the program files is being worked on, the other ones will remain in memory so that they don't have to be explicitly re-loaded each time the one undergoing development is reloaded.

However, suppose the file `foo.c` is loaded, which contains a definition for the function `main`. Then the file `bar.c` is loaded, which happens to also contain a definition for `main`. There will be an error message, because both files contain a `main`. IC will unload `bar.c`, due to the error, and re-download `foo.c` and any other files that are presently loaded.

The solution is to first unload the file containing the `main` that is not desired, and then load the file that contains the new `main`:

```
C> unload foo.c
C> load bar.c
```

6.16 Configuring IC

IC has a multitude of command-line switches that allow control of a number of things. Explanations for these switches can be gotten by issuing the command "`ic -help`".

IC stores the search path for and name of the library files internally; theses may be changed by executing the command "`ic -config`". When this command is run, IC will prompt for a new path and library file name, and will create a new executable copy of itself with these changes.

Chapter 7

LEGO Design

When you're first introduced to the LEGO Technic system, you may be amazed at the number of different kinds of parts you see. There are parts for making moving things — gears, pulleys, axles — as well as parts for making structures. And, at first glance, the function of some of the parts won't be clear.

Once you get into using the LEGO parts, you'll find that they are a powerful way to experiment and play with the design of structures and mechanisms. Unlike the machinist who cuts and drills metal parts, you'll be able to "undo" the things you try out. This means that, also unlike the machinist, you won't have to draw out detailed plans of the finished product before you start trying to make it. You'll learn lots about mechanisms and structures as you play with the LEGO pieces. And you'll learn lots about the LEGO pieces themselves.

In fact, playing with the LEGO pieces is really the best way to learn how to use them. So why should you read this document about using LEGO pieces? Well, quite honestly, if you're the sort of person who likes to learn by exploring things on your own, you might not want to read this document at all. This document might be useful if you're pressed for time and want a shortcut to some of the insights that would otherwise take you a while. It might also be useful if you're a bit intimidated by all the different LEGO parts and would like to see examples of how to use them. This manual isn't a replacement for playing with the LEGO parts though!

If you're not already familiar with the LEGO parts you have, play with them. Make something small. Make something silly. Learning ways to use all the different parts can be fun. Don't worry too much about figuring out what a certain part is "supposed" to be used for – most parts can be used in lots of different ways.

Don't be disheartened if things don't work at your first attempt. Things rarely will, but LEGO parts make it easy to try again in a few different ways.

7.1 The Magic LEGO Dimension

You probably already know that most LEGO bricks come in one of two heights. The short bricks are one third the height of the tall bricks. See Fig. 7.1. But the curse of

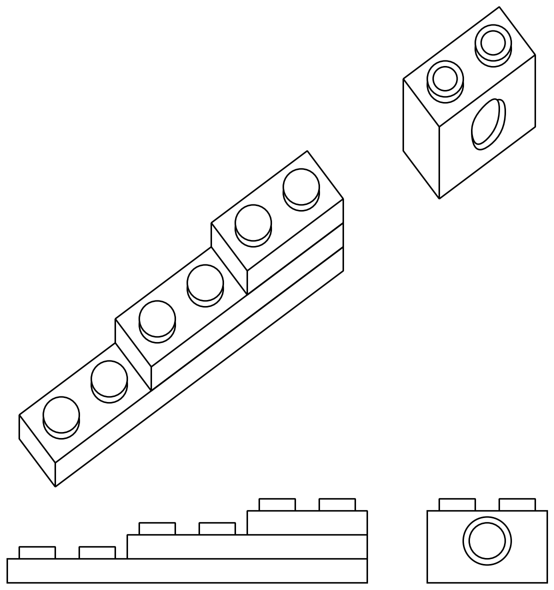


Figure 7.1: LEGO Spacing

LEGO is that neither of these heights is the same the unit of width for the LEGO brick. The curse thwarts efforts to make vertical structures, or geartrains where gears are vertically adjacent.

But armed with the magic LEGO dimension, you can fight back. The ratio of height to width of a tall LEGO piece is 6:5, and, with some creative stacking of LEGO pieces, you can make vertical spacings that are integral multiples of the horizontal spacings. See Fig. 7.2.

7.2 LEGO Gears

Besides the fact that they're just plain cool, why would you want to use gears in your LEGO creation?

You can use gears to translate rotational motion to linear motion (or vice versa), or to transfer motion from one place to another. The main reason for gears, though, is to reduce the speed (thus increasing the torque) of the electric motors.

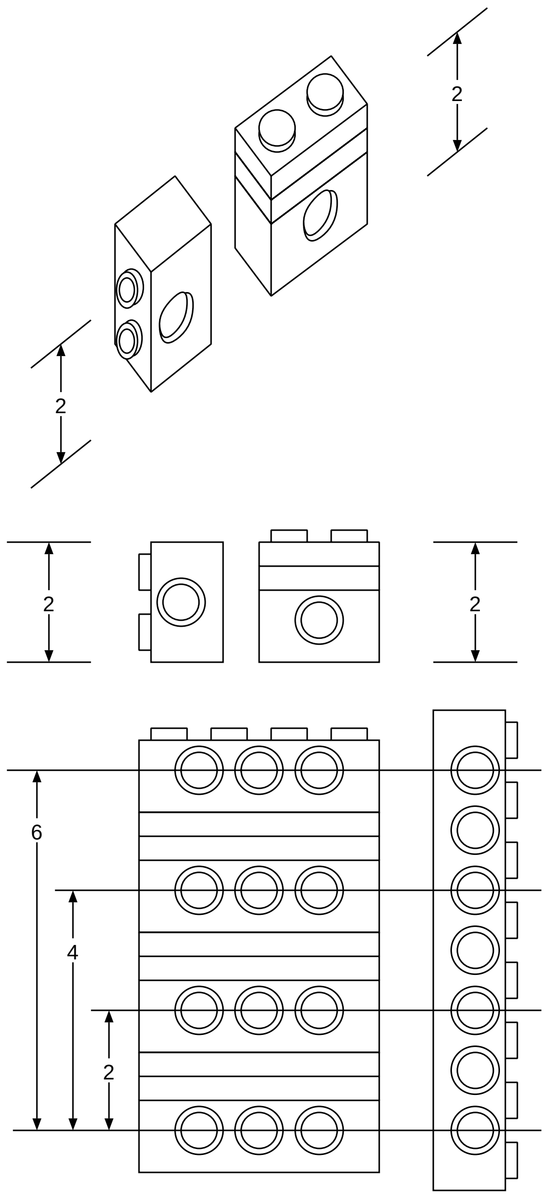


Figure 7.2: Magic LEGO Dimensions

7.2.1 It's all the motor's fault

The little electric motors are really lacking in torque, or, in other words, they can't push very hard. If you try connecting one up directly to a wheel driving your robot around, you'd find the motor too weak to turn the wheel and get your robot to budge one little bit.

But, luckily, even though the motors don't have much torque, they have lots and lots of speed. When you turn your motor on and let the shaft run freely, it probably spins in the neighborhood of 100 to 150 revolutions per second (or 6000-9000 RPM). That's a lot faster than you'd want to drive your robot, anyway! (If your drive wheel is 4 inches in diameter and spins at 9000 RPM, your robot would go

$$\frac{60 \times 9000 \times 4 \times 3.14}{\frac{12}{5280}} = 107 \tag{7.1}$$

miles per hour).

Enter gears. Gears provide you a way to trade speed for torque. You will need to make a gear train to translate a high speed, low torque input into a low speed, high torque output.

Geartrains aren't all that hard to make. But making a really efficient geartrain is an art that takes a while to master. If you're in a hurry and don't have time to discover all the ins and outs of making efficient geartrains on your own, you might be able to make use of the following list of pointers for making geartrains.

7.2.2 Pointers for an efficient geartrain

Use the following gears show in Fig 7.3.

Considerations:

- 16 tooth gear: The 16 tooth gear has a diameter such that it only meshes straightforwardly with other 16 tooth gears. Avoid it.

- 24 tooth crown gear You can make a pretty good right-angle with this gear if you do it right, but it is less efficient than the three good gears. Avoid it.

- Worm gear: I know the worm gear probably seems like the coolest gear you have, but resist the urge to use them if you're trying to make an efficient geartrain! Worm gears are by far the least efficient of the gears you have. Depending on how they are used, they can sap a factor of two or three in energy.

- Pulleys: Pulleys and rubber bands tend to slip when large forces are involved, so if you don't want to lose much efficiency, use them only in the fast-moving parts of the geartrain (since the fast-moving part has the least force). If you

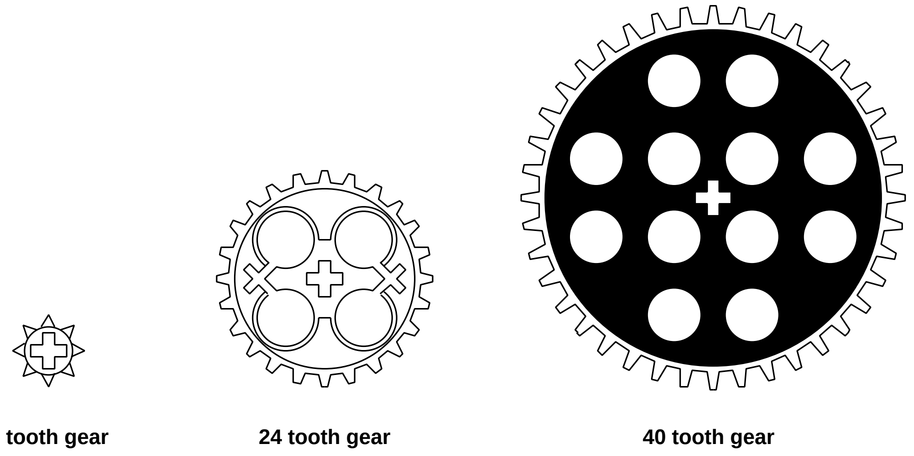


Figure 7.3: Gear Sizing

want your device to be very reliable, you might want to avoid pulleys and rubber bands because the rubber bands can break at inopportune times.

- Chain drive: The chain drive is a little inefficient, but not too bad in the slower stages of a geartrain. There are times when it's invaluable for transferring motion from one place to another. You have to be patient, though – it requires a bit of trial and error to find gear spacing that will work for the chain. If your chain is too loose, it may skip under heavy load, and if it is too tight, you may lose power. Experiment with it. The chains tend to work better on the larger gears.

- Spacing: Try to space the gears from each other by a perfect LEGO horizontal spacing. This is easy to do if you mount the axles horizontally adjacent on a beam. You can also mount gears vertically adjacent if you remember the magic LEGO dimension (1 2/3 vertical = 2 horizontal). Sometimes it is possible to find good meshing distances when the gears are at diagonals, but beware: if the gears are just a little to close together or a little too far apart, the efficiency goes way down.

- Axle supports: Try to have each axle supported by at least two girders as unsupported axles may bend. It's best if the two supporting girders are spaced apart from each other a little bit, but firmly attached to each other by more than one cross-support.

- Try not to have a gear dangling at the end of an unsupported axle. The axles can bend. Either put gears between the girders supporting the axles, or on the outside very close to the girders. If the gear is two or more LEGO units away from the outside of the girders, you may have problems.

- Make sure the axles can slide back and forth a tiny bit. Otherwise, the gears and spacers on the axle are probably pushing up against the girder. This results in surprising amounts of friction which causes the most problem for the geartrain, especially on axles that spin more quickly.

- Gear ratios: Experiment with different gear ratios. The gear ratio determines the important tradeoff between speed and torque.

- Err on the side of too much torque and too little speed. Lots of things can make your geartrain less efficient than when you first tested it. Your batteries might be a bit lower, or a LEGO piece might move slightly to create more friction. And LEGO gears age slightly over time – little bits of plastic wear away and create more friction. So you might choose to design a bit of overkill in the gear train.

7.2.3 How to know if your geartrain is really good

Try backdriving your geartrain. Take off the motor (if it's on), place a wheel on the slow output shaft, and try to turn the wheel. You should be able to make all the gears spin freely from this slow axle. If your gear down is really really good, the gears should continue spinning for a second or two after you let go.

If your geartrain doesn't backdrive, check the following things first:

- Can each axle in the geartrain slide back and forth just a little bit?

- Are the two girders supporting the axles firmly attached to each other by more than one cross-support?

7.2.4 You don't always need an efficient geartrain!

All geartrains don't necessarily need to be efficient! While you may care a lot how efficient your robot drive train is because it has to move the entire mass of the robot, you might not care how efficiently the robot closes its claw or rotates its sensor array to track the opponent.

If you don't care at all about the efficiency of a geartrain, the worm gear allows you to make a more compact geartrain because it gives a faster geardown than do the other gears. (Treat it like having a single tooth in gear ratio calculations! Hooking one up to a 24 tooth gear gives a 24:1 ratio, or a 40 tooth gear gives a 40:1 ratio.)

7.3 Making extremely strong structures

One objection many people have to making things out of LEGO pieces is that LEGO pieces can fall apart. And I think they must fall apart more easily on contest night, too.

Well, believe it or not, it's possible to make a LEGO creations which don't fall apart. Just make use of the **Magic LEGO Dimension** to brace your structures vertically. See Fig 7.4.

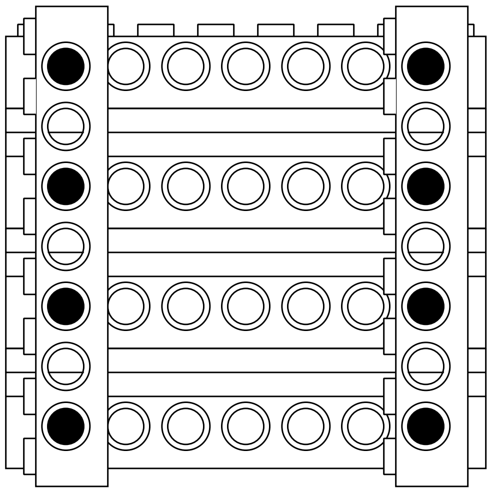


Figure 7.4: Bracing for Success

7.3.1 How to know if your structure is extremely strong

Drop it. If it breaks into a thousand tiny pieces, sorry, it wasn't extremely strong. Advanced LEGO builders can try sledgehammers (just kidding).

7.4 LEGO Challenges

Choose one (or more!) of the following to try and make:

- Make a geartrain with the ratio 135:1
- Make a device to convert rotary motion into a back and forth motion
- What's the largest thing you can build that doesn't fall apart when dropped from waist-level? From shoulder-level?

- Make a car powered by a stretched rubber band. How far can you make the car go?
- Make a ratchet to allow a shaft to turn in only one direction
- Make a catapult or gun capable of shooting a LEGO piece. How can you store the energy to throw the piece? Bonus points if you can hit an organizer or T.A.

Chapter 8

Robot Control

To the uninitiated, the term "robot" conjures up images of a self-contained intelligent machine that completes tasks and makes decisions about its environment. Unfortunately, technology has not quite caught up to society's far seeing prophets. The 6.270 robots that you are constructing are capable of fulfilling some of the requirements mentioned above, given some clever programming and some manner of sensors with the environment. The topic of control deals with how the robot is programmed to allow it to deal intelligently with the immediate environment.

8.1 Types of Control

The most obvious way of controlling a robot is to give it tasks to do, without any concern about the environment around it. This form of control is called *open loop* and consists of a simple signal being given and an action being carried out. The robot does not make any sort of "check" as to whether the correct action was completed, rather it mechanically follows the steps in a prescribed pattern. *Closed loop* control involves giving the robot *feedback* on the task as it progresses to allow it to ascertain whether the task is actually being completed. Consider the example of a car: open loop control would consist of placing a brick on the accelerator pedal and locking the steering wheel, whereas closed loop control may have a driver making corrections for disturbances like curving roads and stop signs (not to mention other vehicles).

8.2 Open Loop Control

A control flow diagram of open loop control consists of some signal ("turn left") being interpreted by the microprocessor ("6811") to drive some actuators ("motors") and create some output ("moved robot").

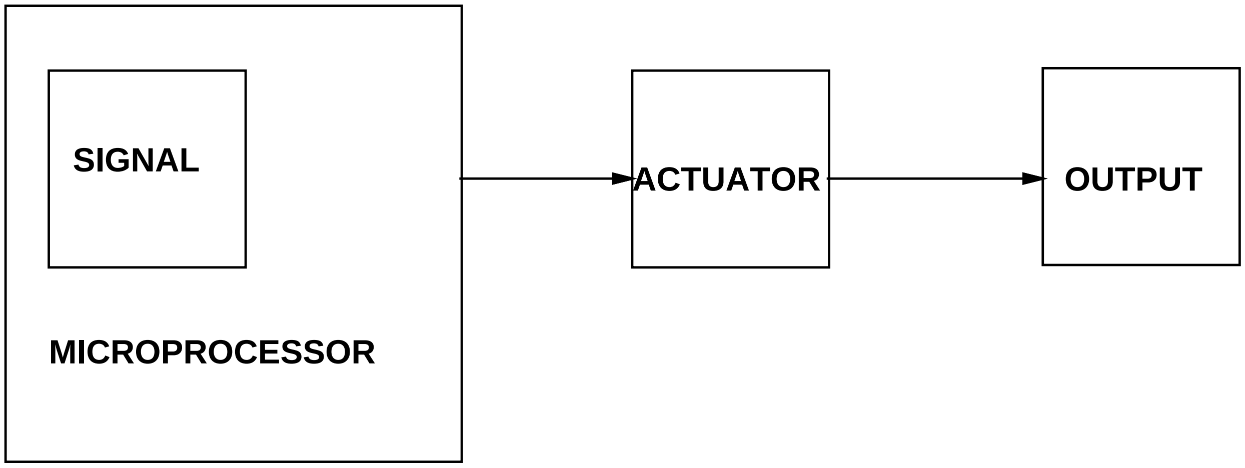


Figure 8.1: Open loop control diagram

Note that there is no link between the output and the moved robot: the microprocessor has no idea of where the robot actually is. This is often what leads to robots getting caught doing silly things at inopportune times. Pattern followers — robots that only use open loop control — often work extremely well in situations that vary little and are readily reproducible. However, the 6.270 tables (during a contest) fit neither of these specifications.

There are a number of reasons that open loop control is used rarely in the real world. The biggest one is the lack of robustness: any small change in the robot or the environment is not corrected for. Consider our hapless automobile, in its current incarnation it is not capable of negotiating turns. Let's create a slightly smarter open loop controller – a 6.270 board hooked up to the relevant control, perhaps the accelerator pedal, brake, and steering wheel. Sending it to the store to pick up some milk would consist of programming a series of commands that negotiate the path.

backwards for 10 seconds (exit driveway)
left for 5 seconds (get parallel to the road)
straight for 30 seconds (to the stop sign)
wait for 5 seconds (a pretense of waiting for the route to be clear)
.
.
.
left for 5 seconds (pull into parking space)

There are several major problems apparent in this fictional trip, even ignoring the possibility of other vehicles on the road. If the tire pressure was low (or the wheels were of a different size than expected) or the fuel was low (or batteries were at low power) the vehicle would start each of the tasks out of the correct position. The left hand turn may place it onto the lawn instead of the street. Even disregarding the fact that the 6.270 board is too short to see over the dashboard, our car-robot is running a blind pattern.

6.270 contest robots run into similar problems when they are controlled in an open loop manner: when they physically stray from the location the microprocessor thinks they are in, they are lost. Changes in the friction of the gear mechanisms, or friction with the table can seriously affect the performance of the robot. Timing for the open loop control will vary with battery power. The frictional and electrical variable will always be different, so the timings will be different for every instance.

However, there are some instances when open loop control is useful. When the robot has no way of sensing how far it is from the desired position, relying on open loop control is the only recourse. For the novice programmer, open loop algorithms are much easier to construct than their closed loop counterparts, and are all that is required for actions which change little from run to run. It might be good to try writing a simple open loop program to determine its limitations before going on to more advanced control.

8.3 Closed Loop Control

Much research has been done on how to "properly" control vehicles and processes using the least power and getting a close as possible to the desired orientation. For this class, the bulk of the knowledge gained in 16.30 and 6.302 is dismissed; the microprocessor allows greater flexibility for the fledgling controls programmer.

The only change to the control flow diagram is the addition of a feedback signal from a sensor shown in Figure 8.2.

The sensor is used to provide information on the current orientation of the robot. This information is used to modify the control input to the robot to correct for the errors in orientation.

A concrete example is a wall following algorithm. The ideal sensors to use are the bend sensors which provide (if interpreted cleverly) the distance the robot is away from the wall. We could place one of these sensors (if we had them — unfortunately, we have not been able to acquire any this year) on the front right side of a robot and bend it slightly so that it always bends in the same direction.

Now if the robot is too close to the wall, we would like it to move away; if it is too far it needs to move closer. Perhaps there is also some ideal range of distances from the wall in which the robot will merrily travel straight. The code to implement this algorithm is fairly straightforward, of more concern is how to determine the necessary values: **CLOSE** and **FAR**. In most cases, trial and error is easiest, placing the robot

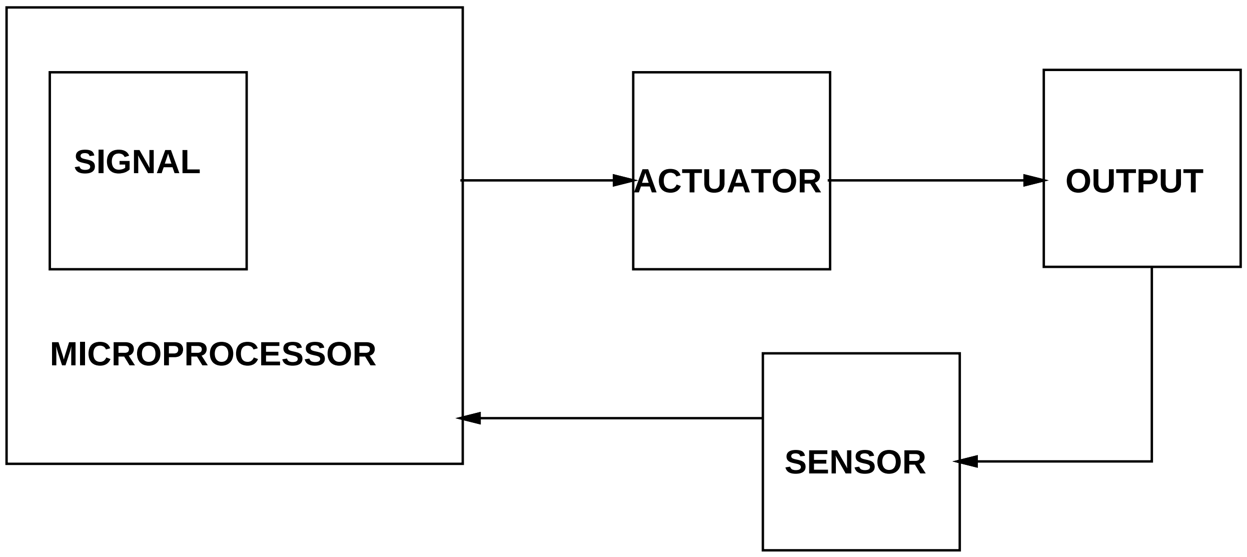


Figure 8.2: Closed loop control diagram

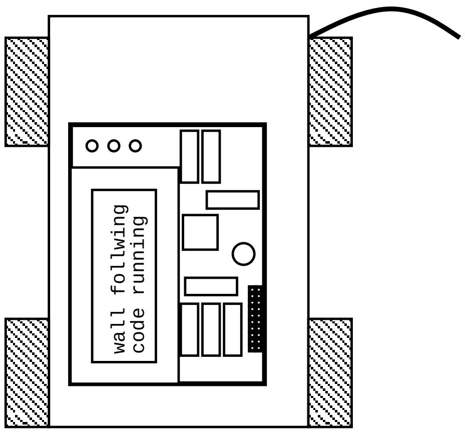


Figure 8.3: Robot with a Bend Sensor

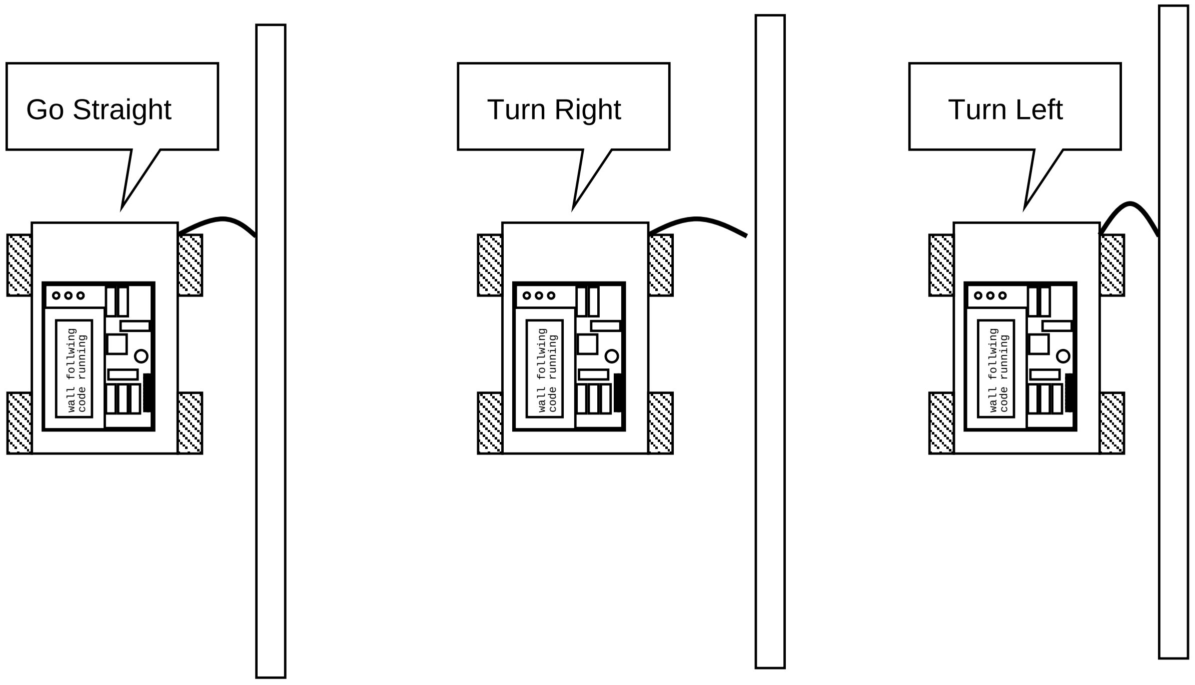

Go Straight
Turn Right
Turn Left
wall follwing code running
wall follwing code running
wall follwing code running

Figure 8.4: Robot moving

the correct distance from the wall, checking the sensor value, then selecting some values nearby for **CLOSE** and **FAR**. If the value of the bend sensor drops below the **FAR** threshold, the robot needs to move closer to the wall, perhaps by running the left motors slightly faster than the right motors. The opposite should occur if the sensor value climbs above **CLOSE**. If the value is in the "dead zone," the robot should continue straight.

There is one problem that we've overlooked in creating this wall following algorithm. The robot does not simply translate towards the wall when asked to move closer; it also rotates. This can lead to the robot jamming into the wall by over-rotating shown in Figure 8.5. The reason that this particular configuration can jam is that we have no feedback as to the orientation of the robot relative to the wall. It may be wiser to measure no only the distance from the wall, but also the angle relative to the wall. Placing another sensor on the back end of the robot is sufficient. Now the distance from the wall can be determined by averaging the two values from the sensors and the orientation of the robot is some function of the difference of the two sensor values. The angle of the robot with respect to the wall can be used to determine if it is running towards or away from the wall.

Note also that the code needs to be modified to handle this extra information coming in. One easy way is to have three steps to wall following:

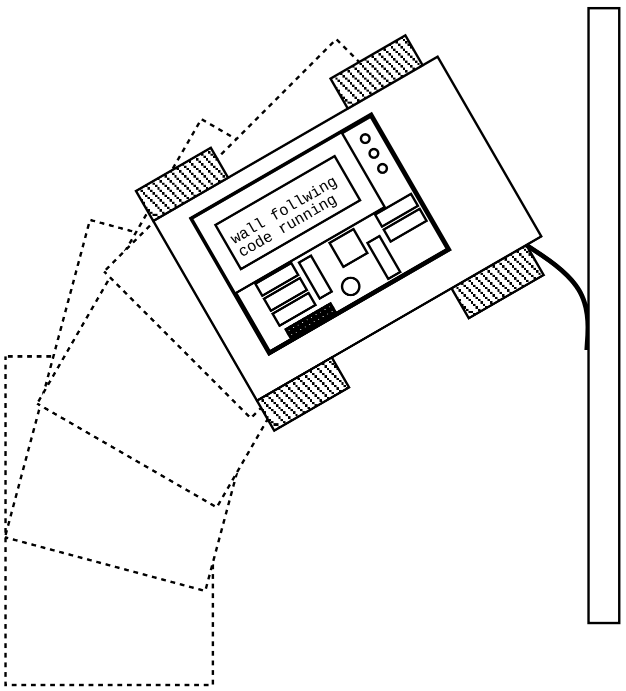


Figure 8.5: Jammed robot

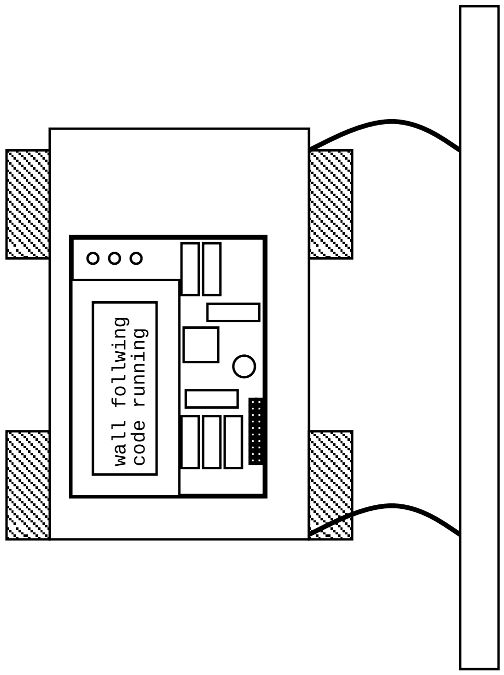


Figure 8.6: Robot with two bend sensors

- Approach the wall with the front sensor until the front sensor is detected.
- Align the robot so that it is parallel to the wall. This step may be very oscillatory where the robot is moving back and forth towards and away from the wall. This is a typical problem in control if the system is not designed carefully.
- Once the machine is aligned it should proceed forward while correcting small deviations in alignment.

A very simple way of implementing this is through a group of **IF-THEN** statements. Since each sensor can be in three possible states: not touching, too close, just right, there will be a total of nine possible states. Try to make simple code using nine if statements to drive your machine along a wall.

These are only suggestions, however, since many people come up with much more clever ways of following walls. You may wish to use previous information, e.g., the angle with respect to the wall, to add more intelligence to your code.

8.4 Closed Loop Control with Coarser Sensors

The bend sensors available in the 6.270 kit are relatively sensitive analog devices; the values correspond to the degree of bend along the length of the strip. Other available sensors are digital in nature, either because of inherent digital attributes (touch switches) or because of the manner in which they are used (breakbeam sensors).

Touch switches are very useful in situations where the optimal orientation either easily obtainable, or only required for a short time. For instance, a gate that needs to be raised to a specific height could be stopped by a touch sensor limit switch. The exact height of the gate is not of too much importance.

Reflectance sensors are inherently analog, however the thresholding done on them to determine the surface color renders them virtually digital. Most code is not concerned with the actual shade of the surface, only whether it is white or some other color (usually black). Setting the values that are considered white and black is a similar process to determining how close or far the robot should travel from the wall; the robot samples values of table surface (both white and white), places them in persistent variable and uses these for reference during the run. The difference is that these values are much more suspect to outside disturbance. Let's find out what problems may arise.

8.4.1 Analog Sensor Problems

Sensors are susceptible to a host of problems, but the somewhat controlled run situations of the 6.270 contest help to compensate for these errors. Below is a diagram of a few of the different sorts of poor data that can be read off of a sensor.

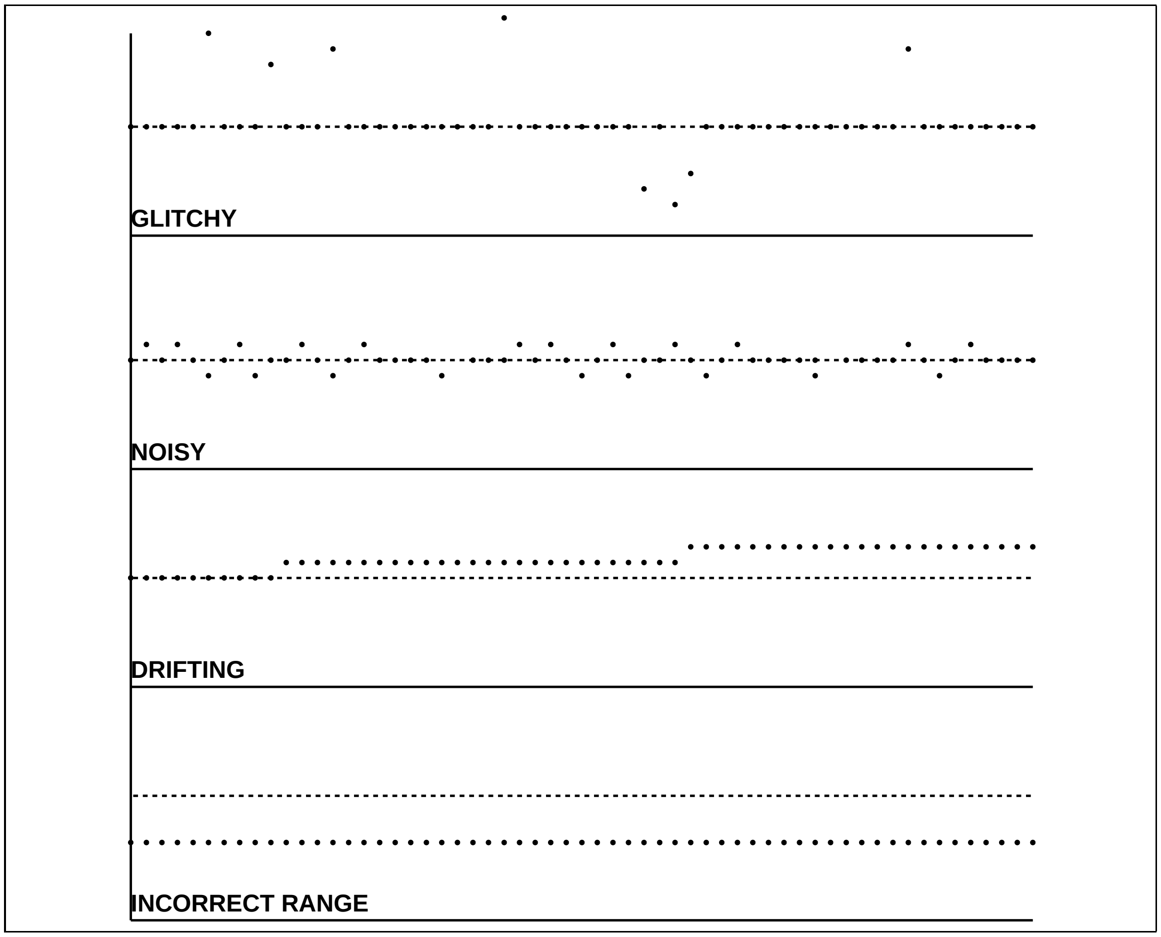


Figure 8.7: Sensor Problems that can arise.

The first problem, that of glitchy data is often inherent in the sensor itself, and must be fixed in software. Throwing out values that are out of the usable range, or looking for unlikely transitions (for instance, the robot suddenly moving too far away from the wall) is a good way to filter out unusable data.

Noisy data also caused by the sensor itself, perhaps due to oversensitivity or due to spurious input from the sensor stimuli. Time averaging the values often leads to cleaner data, and it is reasonable given how much faster the robot electronics run than the mechanical systems.

Drifting data can be caused by sensors retaining some sort of memory. Slowly changing light sources may interfere with color tracking if the sensor is not well shielded. An option is to have the robot capable of changing it's own threshold values during the course of a run. If the robot know it is not touching anything, it could use that value as the unflexed reference for the bend sensor. Or it could recalibrate colors based on what values are sensed over known surfaces. The important thing to note here is that the robot must be certain of being on a calibration surface before changing the threshold values.

Incorrect range is seldom a problem, because the ranges are easy to change in software. If the values are simply too small to show up on the analog robot input it may be necessary to change the input signal from the sensor itself. In the case of a resistive type sensors, simply placing a resistor in parallel or series as necessary can move and amplify the input signal from the sensor. In the case of other types of sensors, more complex amplification electronics may be required. The expansion board on the robot is suitable for mounting an op-amp or transistor to amplify small signals.

8.5 Feed Forward Control

If it is the case that we know some of the disturbances that will appear to the robot, we can correct for them before they affect performance. Consider dropping battery power: if we had some way to sense the energy left in the battery, we could compensate when it was low by having routines run for longer periods of time (since the robot will be moving slower). If the drive train of the robot was changed late in the course (an unwise thing to do), the distance traveled in the same time may change. Rather than changing all of the values, perhaps a few key ones could be changed to allow for a correction factor.

However, feed forward control is not especially useful unless there is no time to correct for the known disturbance beforehand.

8.6 Sensor Integration

There are three major concerns that must be addressed when integrating sensors to a robot: modularity, structural integrity and ease of disassembly.

Modularity refers to how easy it is to move the mounting location of a sensor. If several special LEGO bricks are required to place the sensor in the correct position, it will be difficult to experiment with optimal sensor placement.

Structural integrity is necessary to prevent the sensor from falling off the robot under contest conditions. Precariously mounted bump sensors may knock themselves loose and render themselves useless if not well mounted. Try running the robot over a few bumps and into a few walls to judge the integrity of the sensor mounts.

When a small gear deep in the heart of the robot manages to strip itself, it is helpful to have the entire robot easily fixable. This ideology extends to sensors as well; they are more susceptible to failure than most LEGO pieces.

Finally, please realize that writing control programs for an autonomous robot is no small task. One way to both get an idea of the complexity of the task and some handle on how to solve it, is to play out the code written for the robot. Have one person reading the code, making decisions based on sensor values. Another person reads the current sensor values and a third person (blind-folded perhaps) actuates the robot based on the output of the code. Note that this is analogous to the control flow diagram described early. Though it sounds simplistic, it is useful to have an idea of how limited the robot really is, and how dependent it is on intelligently written control software.

In the past, most successful robots have been mechanically completed early in the course and had over a week devoted solely to creating robust, well written code to control their behavior. Many mechanical deficiencies can be overcome in clever software, but the converse is not necessarily true.

Appendix A

6.270 Hardware

This chapter is partly tutorial and partly technical reference: in additional to documenting the 6.270 hardware, it explains the design in a way that would be understandable to the beginner. The discussion does however assume familiarity with some ideas of digital electronics.

The information presented here should be considered optional, as it is not strictly necessary to know it to build a robot. Hopefully though, this chapter will satisfy most readers' curiosity about how the 6.270 hardware works.

This chapter was revised by Matt Domsch '94 in his "Advanced Undergraduate Project" to reflect changes in the Rev. 2.21 hardware and to provide better schematics.

A.1 The Microprocessor and Memory

At the most primitive level, a computer consists of a microprocessor, which executes instructions, and a memory, in which those instructions (and other data) is stored.

Figure A.1 shows a block diagram of these two components. The diagram shows four types of wires that connect the microprocessor and the memory:

Address Bus. These wires are controlled by the microprocessor to select a particular location in memory for reading or writing.

The 6.270 board uses a memory chip that has 15 address wires. Since each wire has two states (it can be a digital one or a zero), 2 to the 15th power locations are possible. 2^{15} is precisely 32,768 locations; thus, the system has 32K of memory.

Data Bus. These wires are used to pass data between the microprocessor and the memory. When data is written to the memory, the microprocessor drives these wires; when data is read from the memory, the memory drives the wires.

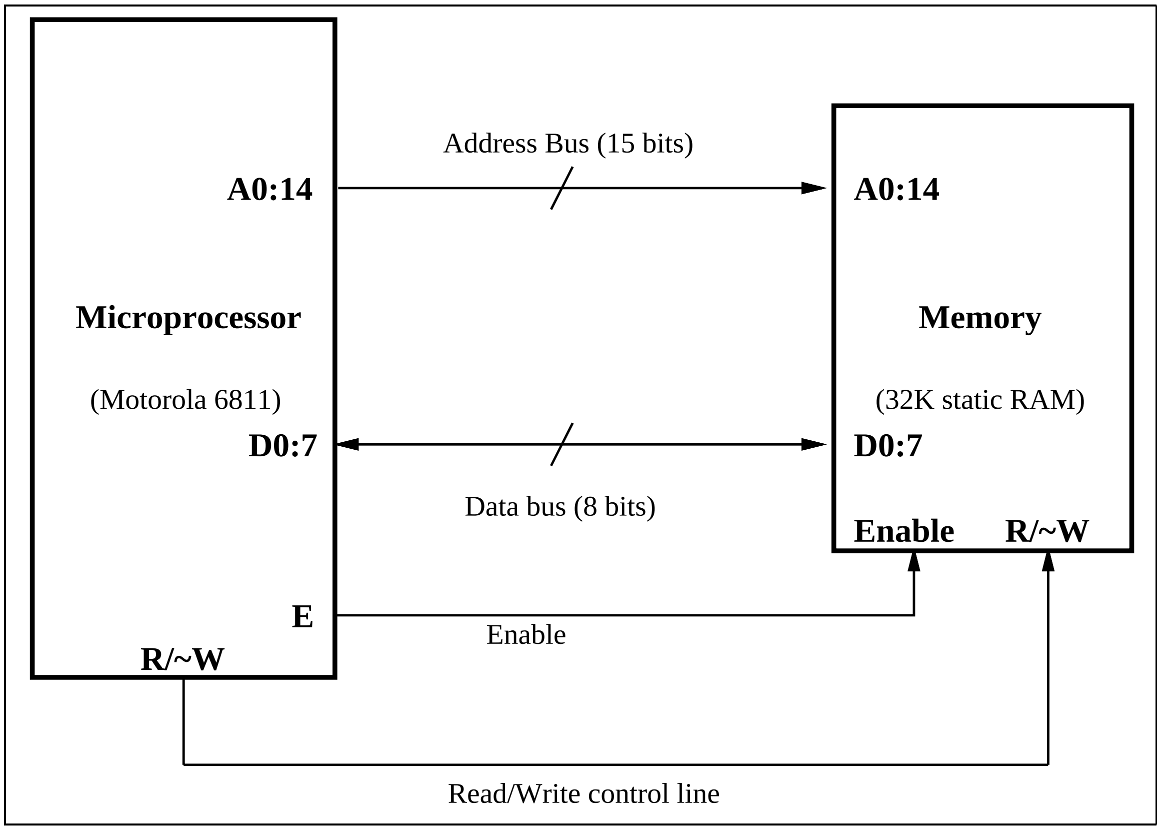


Figure A.1: Block Diagram of Microprocessor and Memory

In our example (and in the 6.270 board), there are eight data wires (or bits). These wires can transfer 2^8 or 256 different values per transaction. This data word of 8 bits is commonly referred to as a *byte*.

Read/Write Control Line. This single wire is driven by the microprocessor to control the function of the memory. If the wire is logic true, then the memory performs a "read" operation. If the wire is logic zero, then the memory performs a "write operation."

Memory Enable Control Line. This wire, also called the *E clock*, connects to the enable circuitry of the memory. When the memory is enabled, it performs either a read or write operation as determined by the read/write line.

A.1.1 Multiplexing Data and Address Signals

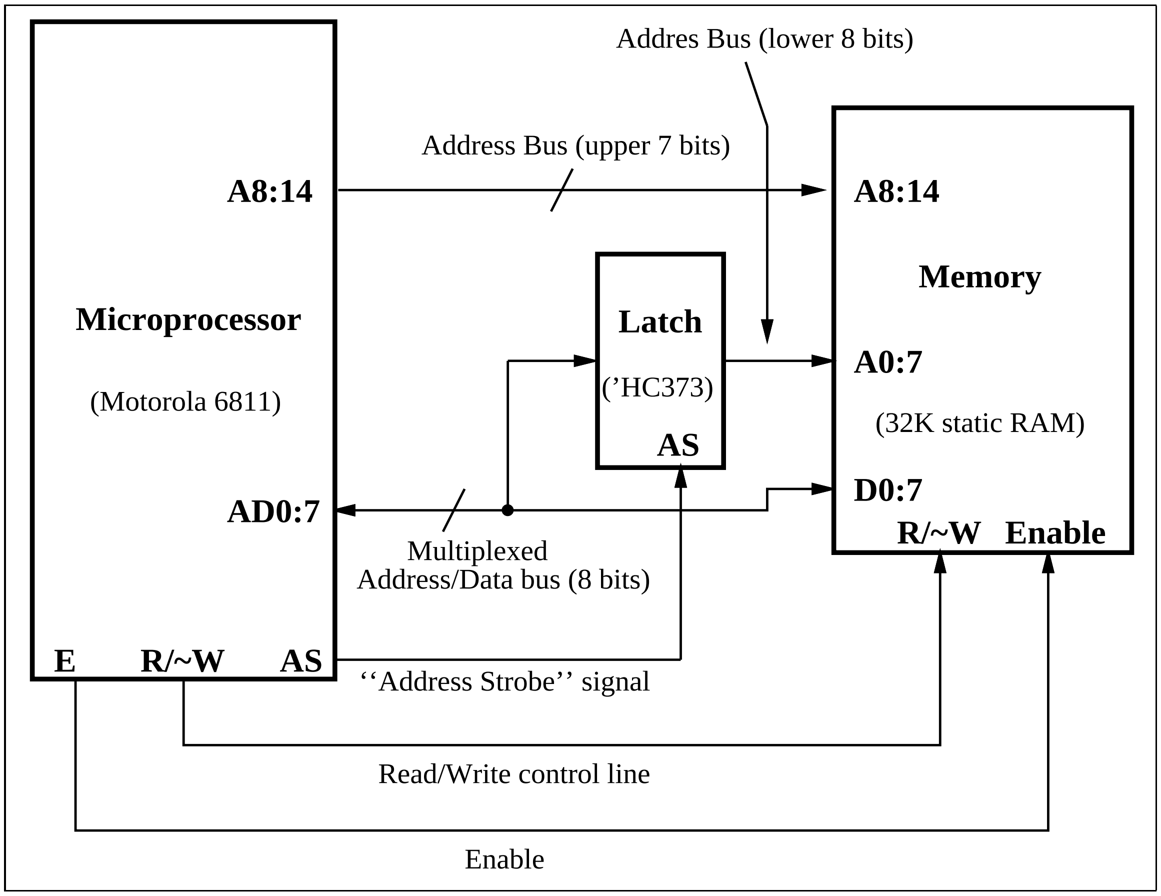


Figure A.2: Block Diagram of Microprocessor and Memory with Latch

Things are a little more complex with the particular microprocessor that is used in the 6.270 board, the Motorola 6811. On the 6811, The eight data bus wires take turns functioning as address wires as well.

When a memory location is needed (for reading or writing), first the data wires function as address wires, transmitting the eight lower-order bits of the address. Then they function as data wires, either transmitting a data byte (for a write cycle) or receiving a data byte (for a read cycle). All this happens very fast; 2 million times per second to be exact.

The memory needs help to deal with the split-personality data/address bus. This help comes in the form of an *8-bit latch.* This chip (the 74HC373) performs the function of latching, or storing, the 8 address values so that the memory will have the full 15-bit address available for reading or writing data.

Figure A.2 shows how the latch is wired. The upper 7 address bits are normal, and run directly from the microprocessor to the memory. The lower 8 bits are the split-personality, or, more technically, *multiplexed* address and data bus. These wires connect to the inputs of the latch and also to the data inputs of the memory.

An additional signal, the *Address Strobe* output of the microprocessor, tells the latch when to grab hold of the address values from the address/data bus.

When the full 15-bit address is available to the memory (7 bits direct from the microprocessor and 8 bits from the latch), the read or write transaction can occur. Because the address/data bus is also wired directly to the memory, data can flow in either direction between the memory and the microprocessor.

This whole process—the transmitting of the lower address bits, the latching of these bits, and then a read or write transaction with the memory—is orchestrated by the microprocessor. The E clock, the Read/Write line, and the Address Strobe line perform in tight synchronization to make sure these operations happen in the correct sequence and within the timing capacities of the actual chip hardware.

A.2 Memory Mapping

So far we have seen how a memory can be connected to the address space of a microprocessor. In a circuit like the one of the 6.270 board, the microprocessor must interact with other devices than the memory—for example, motors and sensors.

A typical solution uses 8-bit latches for input and output. These latches are connected to the data bus of the microprocessor so that they appear like a location in memory. Then, the act of reading or writing from one of these memory locations causes data to be read from or written to a latch—to which the external devices are connected.

Figure A.3 is a block diagram of the 6.270 Robot Controller Board system. Following the present discussion that concerns how the motors and sensors are addressed

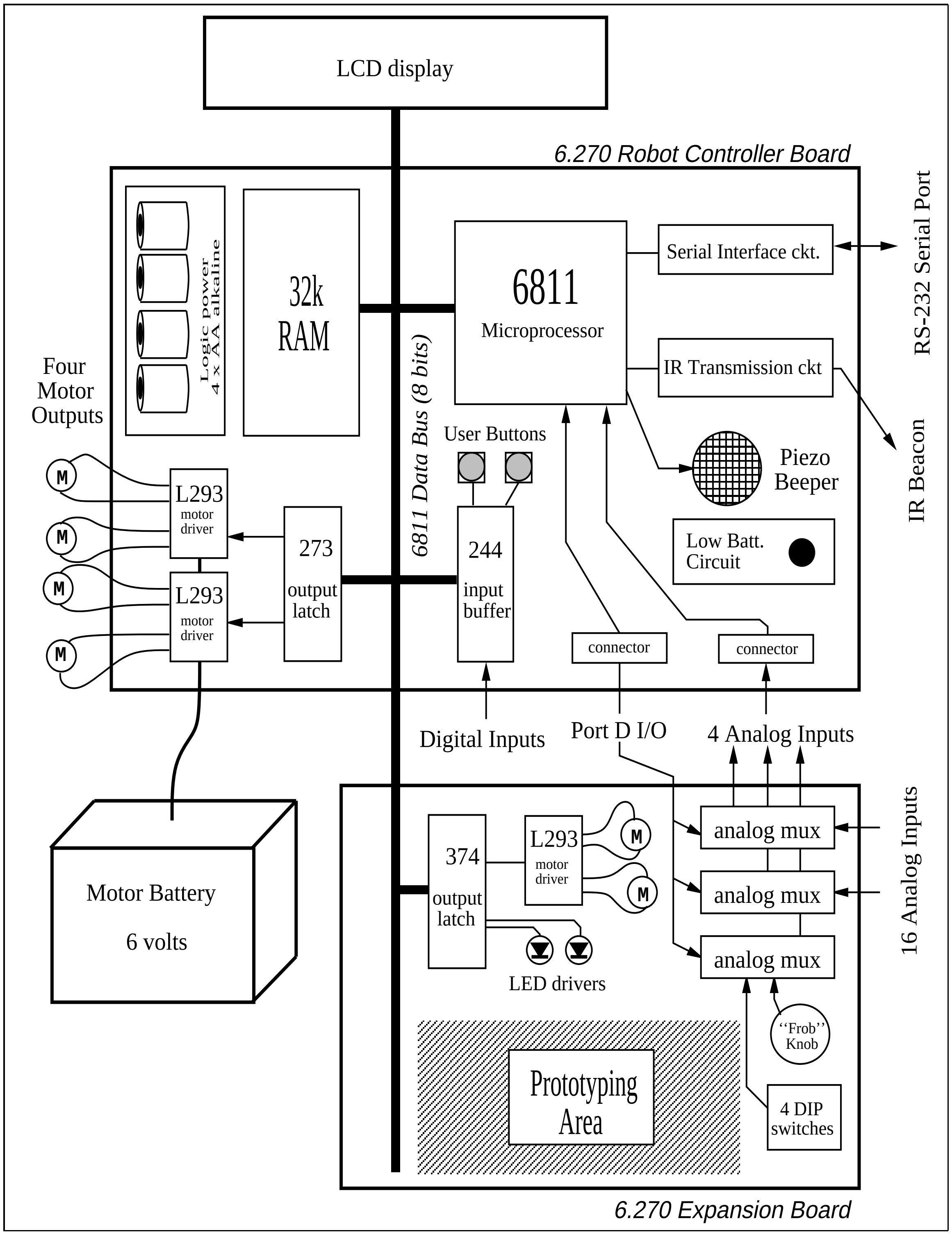


Figure A.3: 6.270 System Block Diagram

by the microprocessor, notice that a chip labelled "273" is connected to the data bus. The '273 has outputs that control the motors (through chips labelled "L293," which will be discussed later). The digital sensors are driven into the data bus by a chip labelled "244." On the expansion board, a 374 chip, another output latch, is used for eight bits of digital output.

These interface latch chips are used in a technique called *memory mapping*. The chips are "mapped" to a particular address in the microprocessor's memory.

The following discussion will show how both the 32k RAM memory and the digital input and output latch chips share the address space of the microprocessor.

A.2.1 Memory-Mapping the RAM

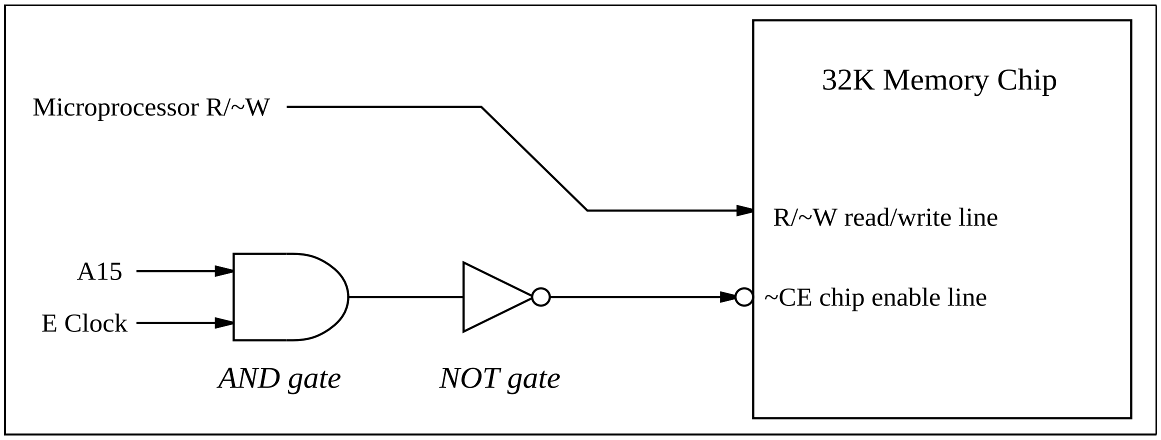


Figure A.4: Enabling the Memory

The 6811 has a total of 16 address bits, yielding 64K bytes of addressable locations (65536, to be exact). Half of this space will be taken up by the 32K memory chip (also known as a *RAM* chip, for "random access memory").

The 6811 has a bank of *interrupt vectors*, which are hardware-defined locations in the address space that the microprocessor expects to find pointers to driver routines. When the microprocessor is reset, it finds the reset vector to determine where it should begin running a program.

These vectors are located in the upper 32K of the address space. Thus, it is logical to map the RAM into this upper block, so that the RAM may be used to store these vectors.

The technique used to map the memory to the upper 32K block is fairly simple. Whenever the 6811's A15 (the highest-order address bit) is logic one, an address in the upper 32K is being selected. The other fifteen address bits (A0 through A14) determine that address.

A logic gate is used to enable the memory when A15 is logic one *and* when the E clock is high (since the E clock must control the timing of the enable). Figure A.4 shows a block diagram of this circuit. (The actual circuit to enable the RAM, shown in Figure A.8, is slightly more complex due to considerations of battery-protecting the memory, as explained later.)

Memory chips are part of a class of chips that have *negative true* enable inputs. This means that they are enabled when the enable input is logic zero, not logic one.

There are two methods for denoting an input that is negative true. As shown in Figure A.4, the chip enable input is shown with connecting to a circle. This circle indicates a negative true input. Also, the name for the signal, *CE* is prefixed with a ˜ symbol.

The function of the NOT gate shown in the diagram is to convert the positive-true enable produced by the AND gate into the negative-true signal required by the ˜CE input. (Often these two gates are collapsed into a single NAND gate.)

A.2.2 Memory-Mapping with the 74HC138 Chip

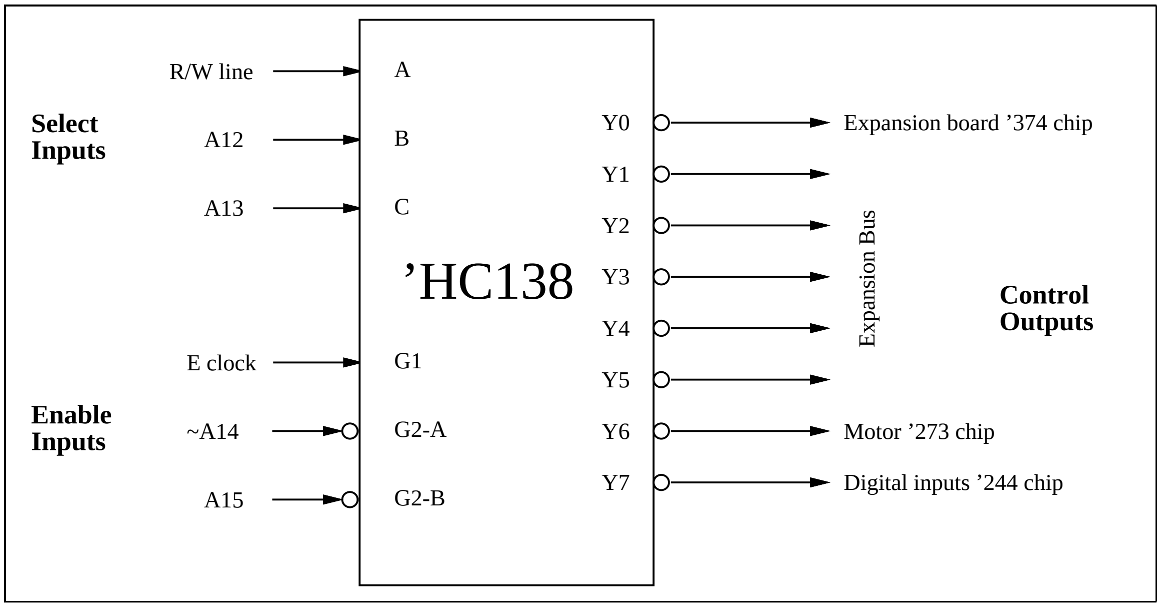


Figure A.5: Wiring the 'HC138 Address Decoder

Figure A.5 shows the 74HC138 chip, which is commonly used in circuits that map devices onto an address space. This chip is a 3-to-8 decoder: a binary number of three digits (the *select inputs*) causes one of eight possible outputs to be selected (the *control outputs*). The chip also has three *enable inputs*, all of which must be enabled to make the chip become active.

The outputs of the '138 chip control the input and output latches shown in the system block diagram. The '138 determines when these latches are activated, either to read data from the data bus (in the case of the '273 output latch), or to write data onto the data bus (in the case of the '244 input latch).

Enable Inputs

The enable inputs of the '138 determine *when the chip will become active,* and thereby turn on one of the input or output latches. These enables inputs are critical because the '138 must not become active at the same time as the RAM chip. If it did, then two devices (the RAM and perhaps a '244) would attempt to drive the data bus simultaneously, causing a problematic situation called *bus contention.*

As shown in Figure A.5, A15, the highest order address bit, is connected to a *negative enable* of the '138. Thus A15 must be *zero* to enable the chip. Since the RAM is enabled only when A15 is one (as was explained earlier), there is no chance that the '138 and the RAM could be active at the same time.

˜A14, which is the logical inverse of A14, is connected to a second negative enable of the '138. Thus when A14 is one, ˜A14 is zero, and the G2-A enable is true. So A14 must be one in order to active the '138.

The final enable input is positive true, and is connected to the 6811 E clock. When A15 is zero and A14 is one, the E clock will turn on the '138 at the appropriate time for standard 6811 read/write cycles.

Select Inputs

Given that the '138 is enabled, the A, B, and C inputs determine which device connected to its outputs will be activated. A, B, and C form a binary number (C is the most significant bit) to determine the selected output.

The A13 and A12 address bits and the 6811 read/write line make the selection. Suppose A13 and A12 are one. The read/write line makes the final choice. This line is one for a read and zero for a write. If a read operation is in progress, then the ABC inputs will form the number 7, and the Y7 output will be activated. As shown in Figure A.5, this output connects to the digital input '244 chip. So, the '244 chip will turn on and will drive a byte onto the data bus. The read operation will complete with this byte having been read from the location in 6811 address space that was selected.

Notice that address bits A0 through A11 have no effect on the operation just described. As long as A15 is zero, A14, A13, and A12 are one, a read operation will cause the '138 to turn on the digital input '244 chip to write a byte onto the data bus. Thus, the digital input chip is selected by a read from any address from $7000 to $7FFF[1]. This is fairly wasteful of the address space of the 6811, but keep in mind that the only circuitry required to arrange this solution was the '138 chip.

Suppose a write operation were to occur in that same range of memory. The relevant upper four address bits would have the same values, but the read/write line would be zero (indicating the write operation). Thus the '138 ABC inputs would form the number 6, and output Y6 would be activated. Y6 is connected to the '273 chip that controls the motors; thus, the '273 would latch the value present on the data bus during the write operation.

As shown in Figure A.5, most of the '138 outputs are still available for future expansion. The 6.270 Expansion Board includes a circuit with one '374 chip, connected to the Y0 output. Outputs Y1 through Y5 are left free for further expansion use.

A.2.3 System Memory Map

Figure A.6 summarizes the memory map solution that has been implemented for the 6.270 Board.

The 32K RAM takes up half of the total address space of the microprocessor. As indicated in the map, it is located in the upper 32K of the microprocessor's memory, from addresses $8000 to $FFFF.

The four digital input and output ports are mapped at locations starting at $4000, $5000, $6000, and $7000.

There is small area of memory that is internal to the 6811 chip itself. This memory consists of 256 bytes located at the start of the address space, from locations $00 to $FF.

The 6811 also has a bank of 64 internal *special function registers*, located at addresses $1000 to $103F. These registers control various hardware features of the 6811 (the analog inputs and serial communications are two examples).

The remainder of this section presents details on the digital input and output circuit wiring.

A.2.4 Digital Inputs

Figure A.7 shows the digital input circuitry. U6, a 74HC244 chip, is used to latch an eight-bit word of sensor inputs and drive the 6811 data bus with that value when the chip is selected.

The '244 chip has two halves which may be separately enabled. The Y7 select is connected to both enable inputs, so that both halves of the chip are always selected simultaneously.

[1]These numbers are expressed in the hexadecimal numbering system, in which each digit represents a four-bit value from zero (0) to fifteen (F)

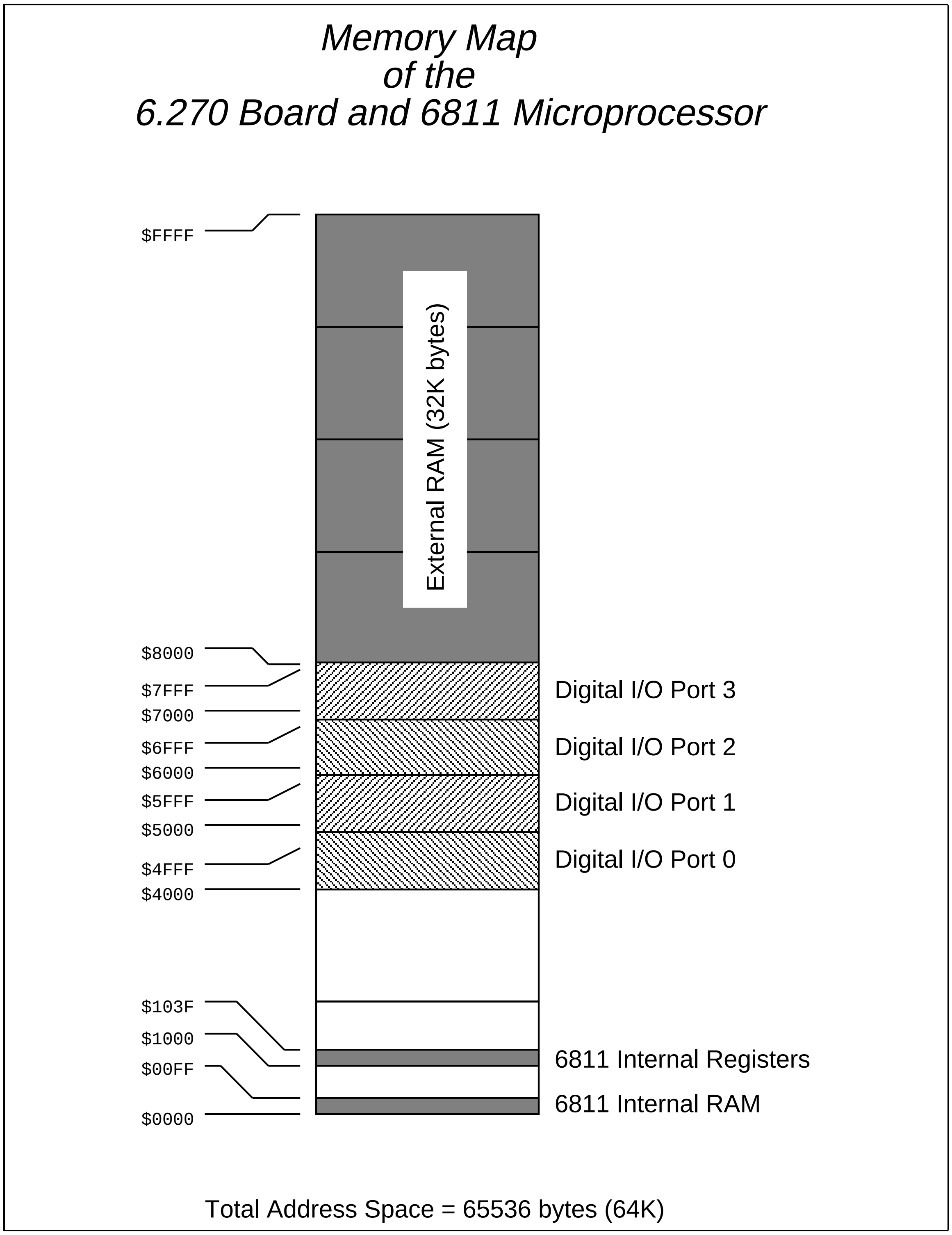


Figure A.6: 6811 System Memory Map

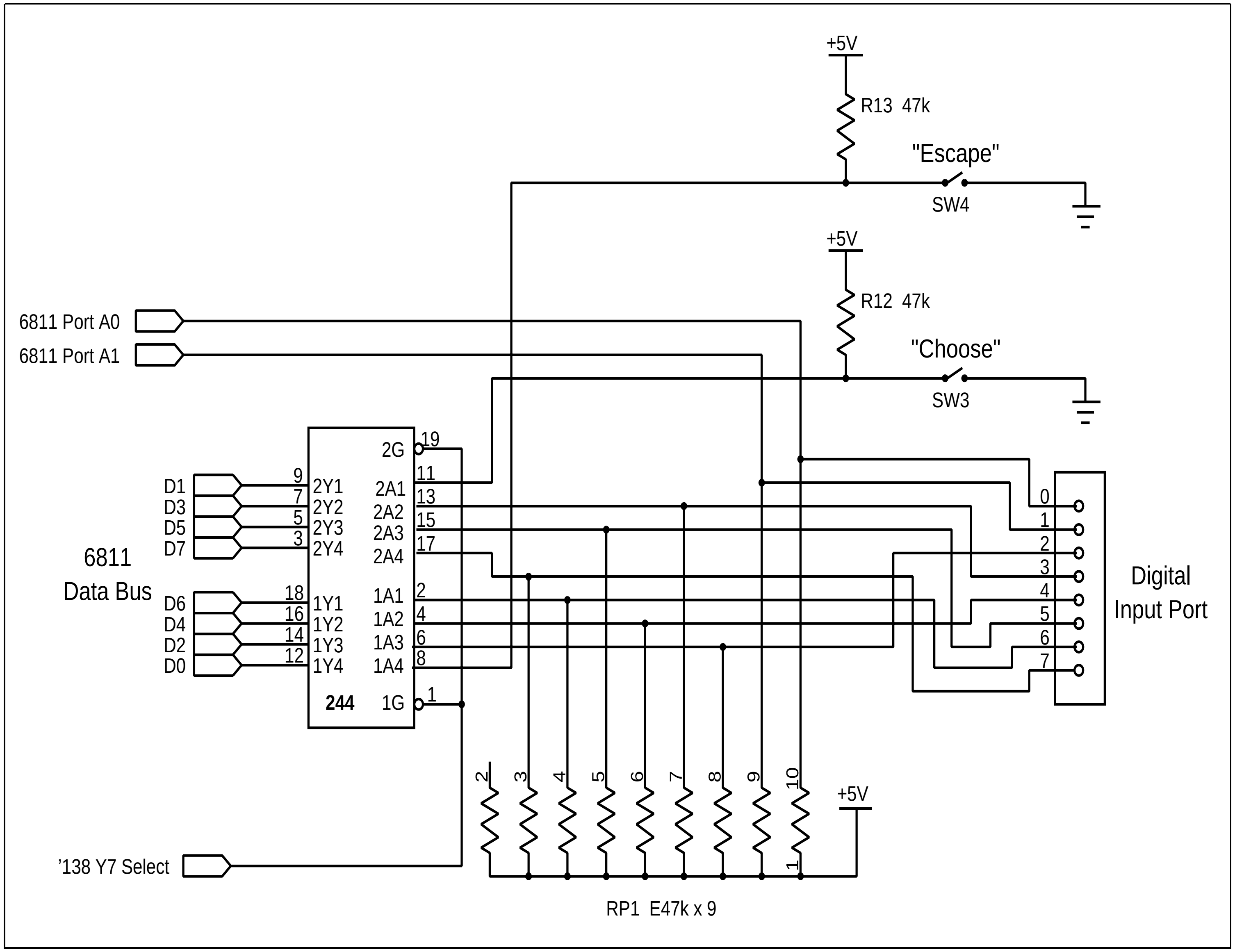


Figure A.7: Digital Input Circuit

The lower two bits of the '244 are connected to the two user buttons (which have been dubbed CHOOSE and ESCAPE). The upper six bits are connected to the digital input header.

The lower two bits of the input header are connected to two timer inputs inputs of the 6811. These inputs can be used to precisely measure waveforms, or can simply be used for digital input. If shaft encoding is used, it is the input capture functions on these two input ports which are used for the encoding. The library functions written to perform digital inputs insulate the user from the fact that the eight pins on the input header are not mapped contiguously to one location in memory.

RP1, a 47K resistor pack, acts as pull-up resistors to the inputs of the '244 chip, making the default values of the inputs one.

A.2.5 Digital Outputs

Figure A.13 shows the complete schematic for the '273 output latch controlling the motors. For the purpose of the discussion to this point, notice that the data inputs of the '273 are connected to the 6811 data bus. The Y6 select signal connects to the clock input of the '273; when Y6 is activated, the '273 latches the value present on the data bus.

The outputs of the '273 connect to the motor driver chips. This circuitry is explained in the following section.

Figure A.15 is the schematic of the motor circuit present on the 6.270 Expansion Board.

A.2.6 6811 and Memory Schematic

Figure A.8 presents the schematic of the 6811, memory, address decoding, and supporting main circuitry on the 6.270 Processor Board. By the end of this chapter, most of the circuitry depicted here will be explained.

A.3 The Motor Drivers

Motors are high-powered devices in the world of digital electronics. A typical digital output can supply about 10 to 20 milliamperes (mA) of current; a small permanent-magnet motor requires anywhere from 500 to 4000 mA of current. It should not come as a surprise that special circuitry is required to drive motors.

A.3.1 The H-Bridge Circuit

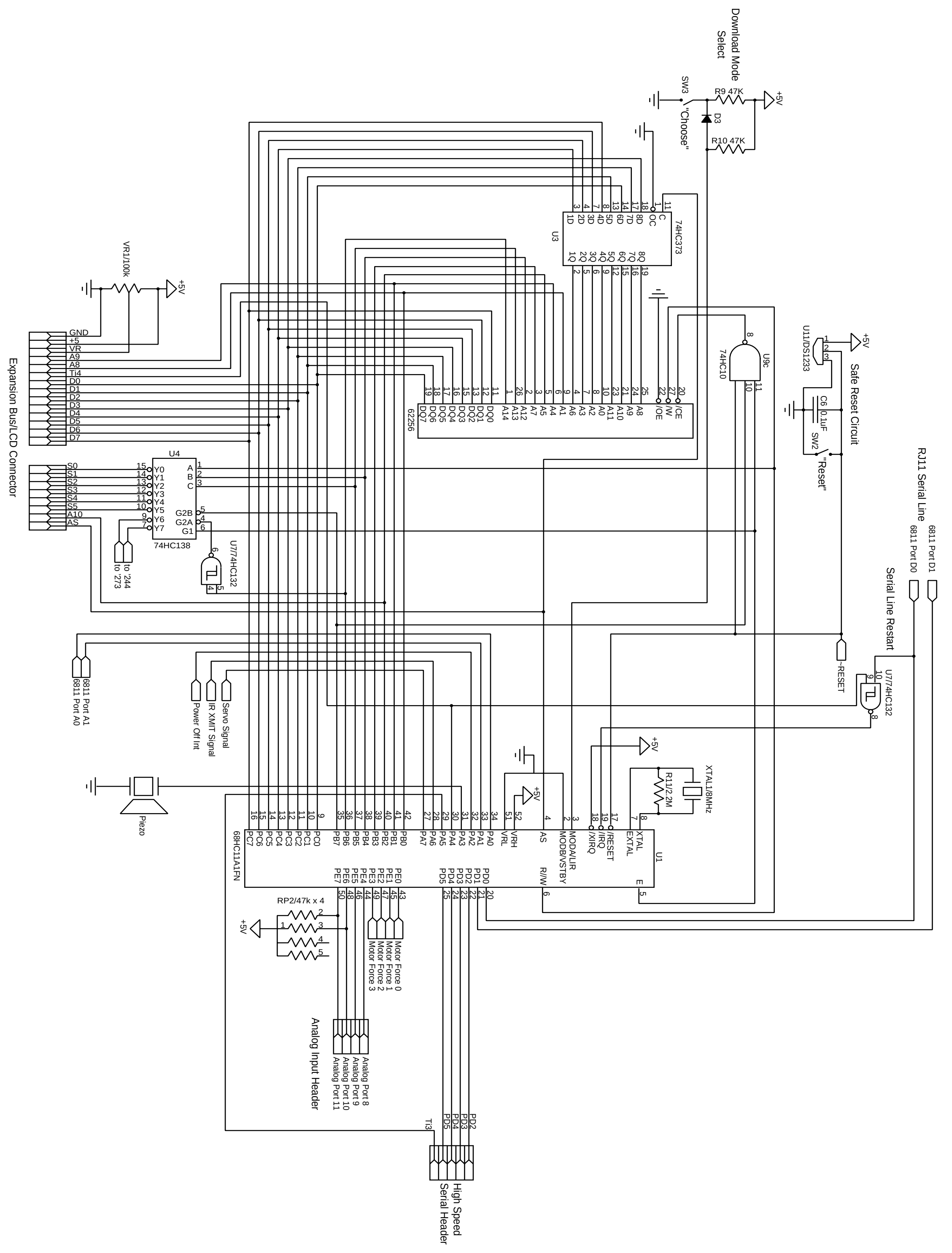


Figure A.8: 6811, Memory, Address Decoding and Miscellaneous Circuitry

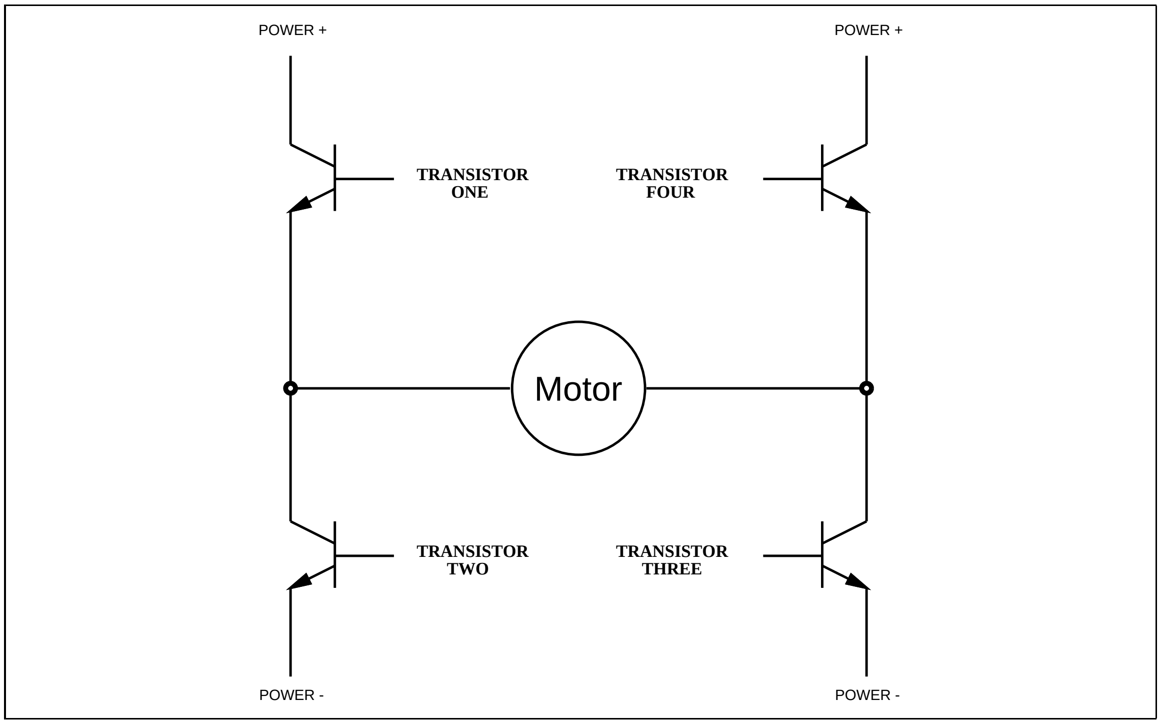


Figure A.9: The H-Bridge Circuit

A circuit known as the *H-bridge* (named for its topological similarity to the letter “H”) is commonly used to drive motors. In this circuit (depicted in Figure A.9), two of four transistors are selectively enabled to control current flow through a motor.

As shown in Figure A.10, an opposite pair of transistors (Transistor One and Transistor Three) is enabled, allowing current to flow through the motor. The other pair is disabled, and can be thought of as out of the circuit.

By determining which pair of transistors is enabled, current can be made to flow in either of the two directions through the motor. Because permanent-magnet motors reverse their direction of turn when the current flow is reversed, this circuit allows bidirectional control of the motor.

A.3.2 The H-Bridge with Enable Circuitry

It should be clear that one would never want to enable Transistors One and Two or Transistors Three and Four simultaneously. This would cause current to flow from Power+ to Power− through the transistors, and not the motors, at the maximum current-handling capacity of either the power supply or the transistors.

To facilitate control of the H-bridge circuit, enable circuitry as depicted in Figure A.11 is typically used.

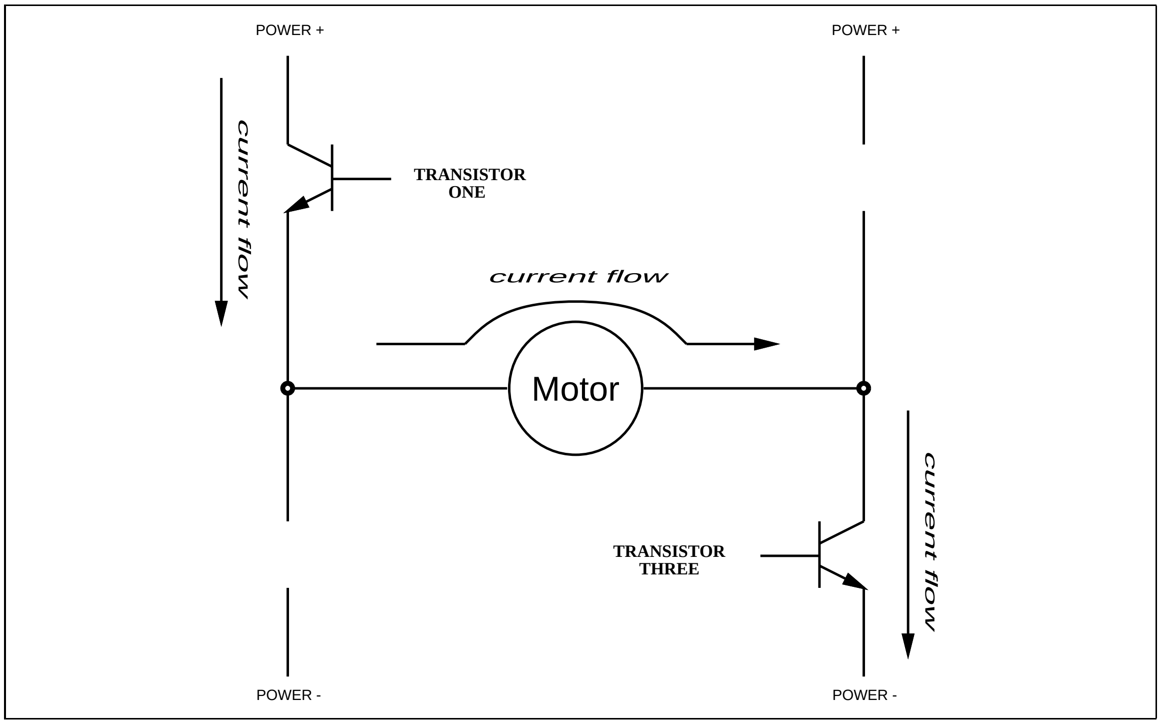


Figure A.10: The H-Bridge with Left-to-Right Current Flow

In this circuit, the inverters ensure that the vertical pairs of transistors are never enabled simultaneously. The *Enable* input determines whether or not the whole circuit is operational. If this input is false, then none of the transistors are enabled, and the motor is free to coast to a stop.

By turning on the *Enable* input and controlling the two *Direction* inputs, the motor can be made to turn in either direction.

Note that if both direction inputs are the same state (either true or false) and the circuit is enabled, both terminals will be brought to the same voltage (Power+ or Power−, respectively). This operation will actively brake the motor, due to a property of motors known as *back emf*, in which a motor that is turning generates a voltage counter to its rotation. When both terminals of the motor are brought to the same electrical potential, the back emf causes resistance to the motor's rotation.

A.3.3 The SGS-Thomson Motor Driver Chip

A company named SGS-Thomson makes a series of chip called the L293 that incorporates two H-bridge motor-driving circuits into a single 16-pin DIP package. Figure A.12 shows a block diagram of this incredibly useful integrated circuit.

The schematic of the motor circuit (Figure A.13) shows how the L293 chips are

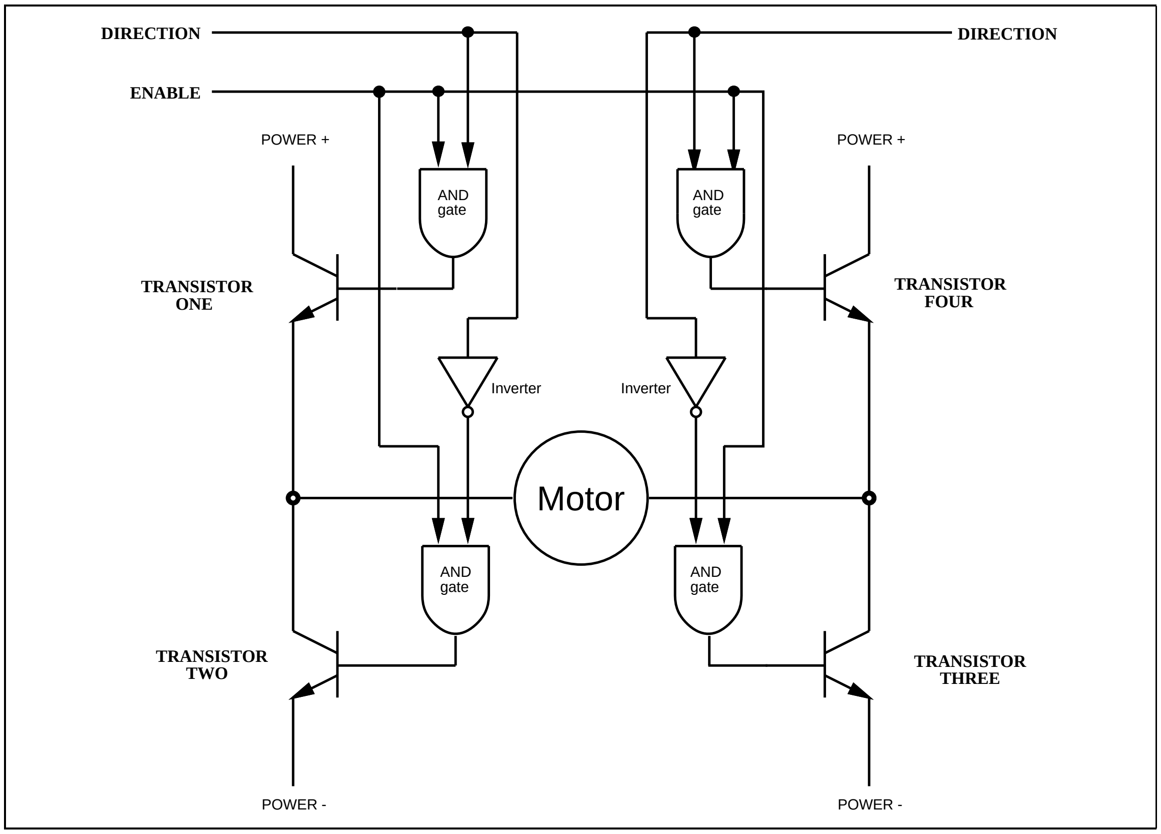


Figure A.11: The H-Bridge with Enable Circuitry

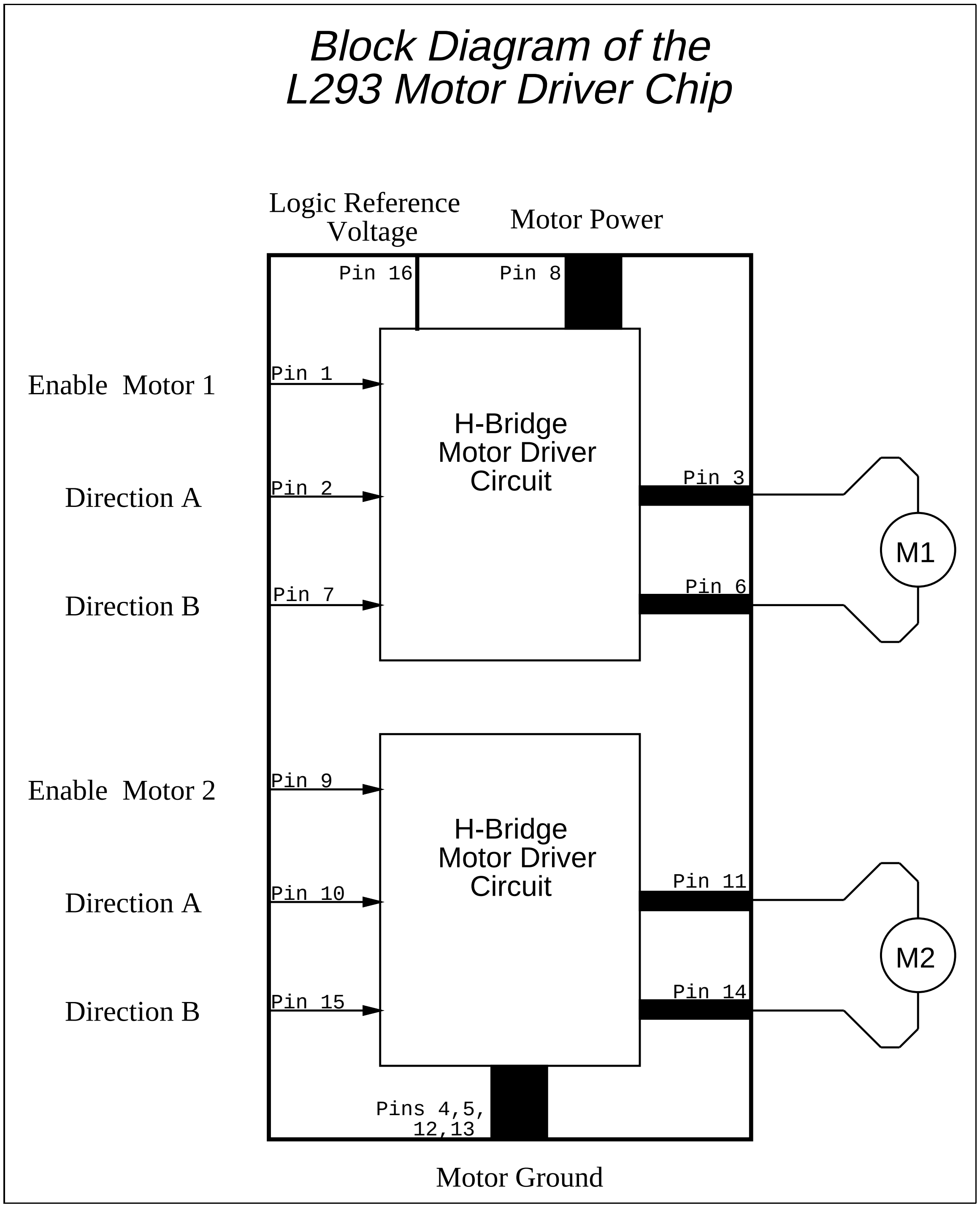


Figure A.12: The SGS-Thomson L293 Motor Driver IC

used in the 6.270 board design. Eight bits are used to control four motors. Four of the bits determine the direction of the motors (with the assistance of inverters) and four bits determine the when the motors are on or off.

Notice that braking a motor is not possible with this circuit configuration, because the inverters do not allow both direction inputs of a given motor to be the same state.

The speed of a motor may be controlled by pulsing its enable bit on and off. This technique, called *pulse width modulation*, is explained in the chapter on motors.

A.3.4 Power Considerations

Current Handling and Spike Protection

In the 6.270 circuit design, two L293 chips are used in parallel to control each motor. This is an unconventional circuit hack to add to the current-handling capacity of the motor drivers.

Two different L293 chips are used in this circuit. One chip, the L293D, has internal spike-protecting diodes on the motor outputs. These diodes protect the motor chip and the rest of the circuit from electrical noise generated by the motors. The other chip, the L293B, does not have these diodes, but has a greater current handling ability than the 'D chip.

The L293D can supply 600 mA of current per channel; the L293B, 1000 mA. Used in parallel, the circuit can supply 1600 mA per channel. Because of the spike-killing diodes contained in the 'D chip, the overall circuit is safe to use.

Power Supply Isolation

The electrical noise generated by motor can be hazardous to a microprocessor circuit even with the use of the diodes. For this reason, separate power supplies are used for the motors and the rest of the microprocessor electronics.

Figure A.14 shows the power-supply circuitry. Notice that *Logic Power*, for the microprocessor circuitry, is a configuration of four AA cells, while + *Motor*, power for the motors, is supplied through the J1 connector.

The motor ground and the logic ground must be kept at the same potential so that the control signals from the '273 chip shown in Figure A.13 can communicate with the L293 chips. These grounds are kept at the same potential by the inductor L1.

The inductor is used to provide reactance (frequency-dependent resistance) to trap spikes that might travel from the motors, through the L293 chips, and into the microprocessor circuit.

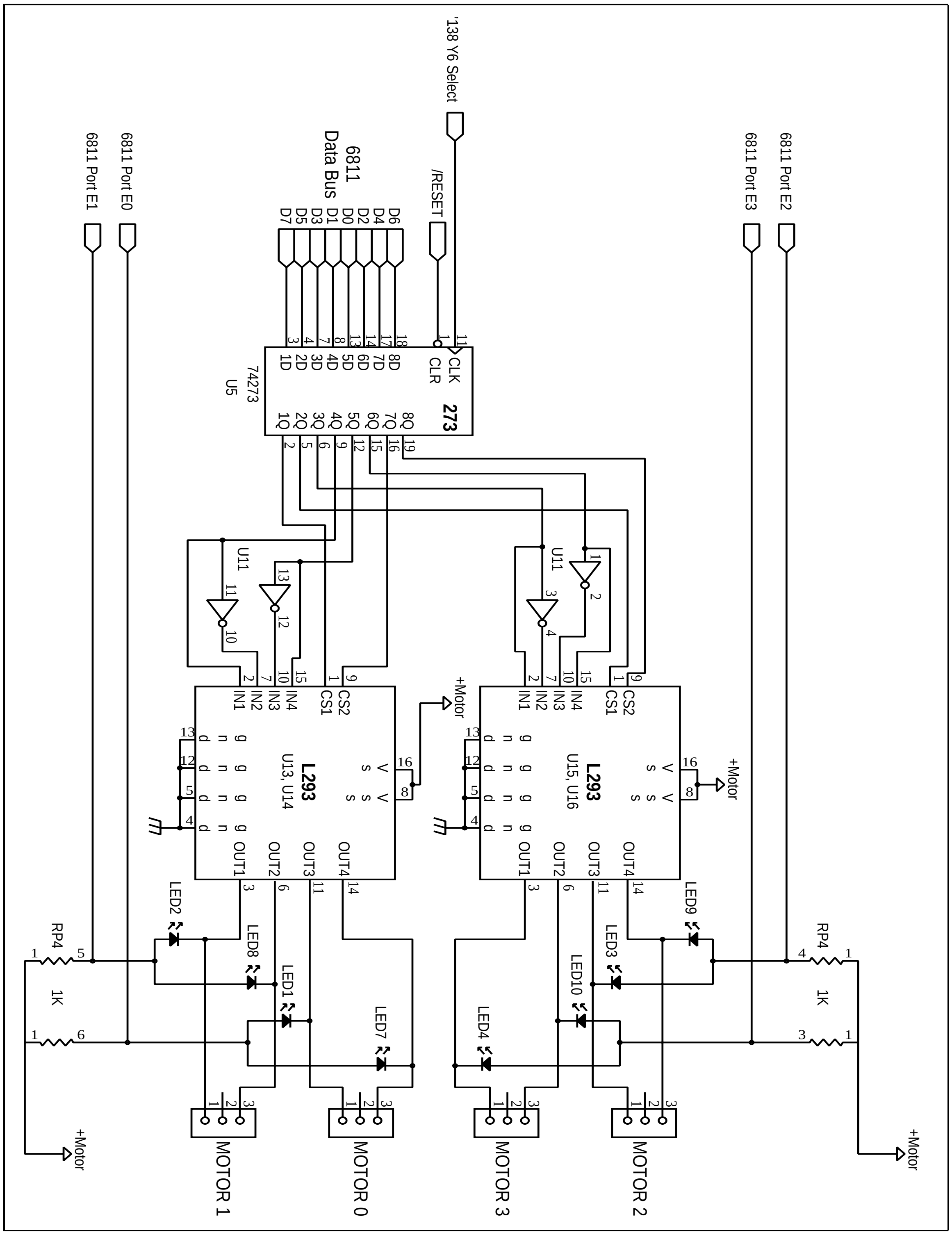


Figure A.13: Motor Driver Circuit

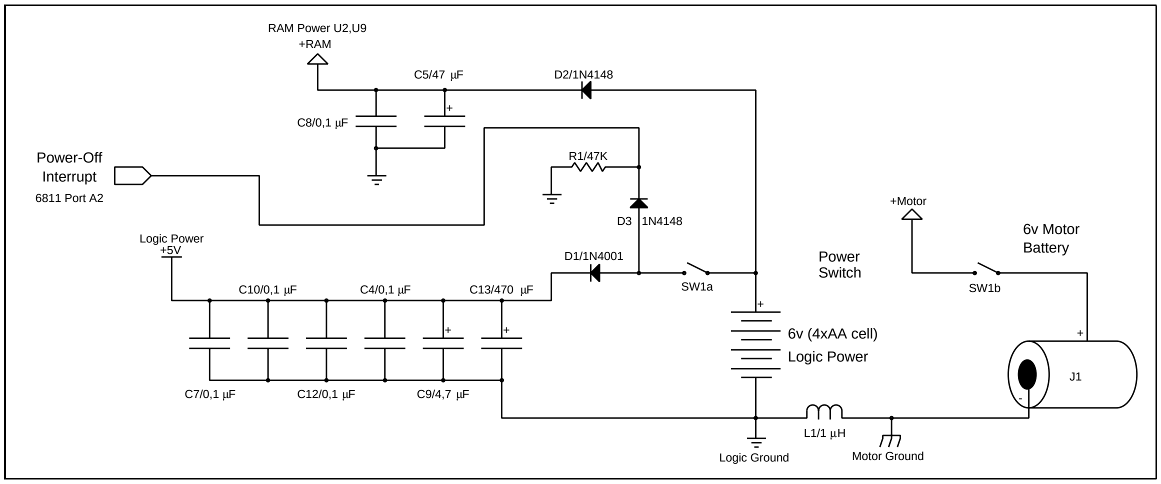


Figure A.14: Power Filtering and Switching Circuit

A.3.5 Expansion Board Motor and LED Circuitry

The 6.270 Expansion Board plugs into the Expansion Bus header depicted in Figure A.8. This header connects to the 6811 data bus and to the six '138 select signals that are not used on the main board.

Figure A.15 illustrates how a single L293D chip is used on the Expansion Board to provide outputs for two additional motors. Because six outputs of the '374 chip are wired to control all four direction inputs and the two enable inputs of the L293D, the motors can be braked if desired. Or, four unidirectional devices may be powered.

The remaining two bits of the '374 are connected to transistor drivers. These transistor circuits are well-suited for powering light-load devices, such as LEDs.

A.4 Analog Inputs

The 6811 has on-chip circuitry to perform an analog-to-digital signal conversion. In this operation, a voltage from 0 to 5 volts is linearly converted into an 8-bit number (a range of 0 to 255). This feature is one of the many that make the 6811 very well suited for control applications.

The 6811 has eight of these analog inputs. In the 6.270 board design, four of these pins are wired to a *motor current monitoring circuit*, and four of them are wired to input connectors.

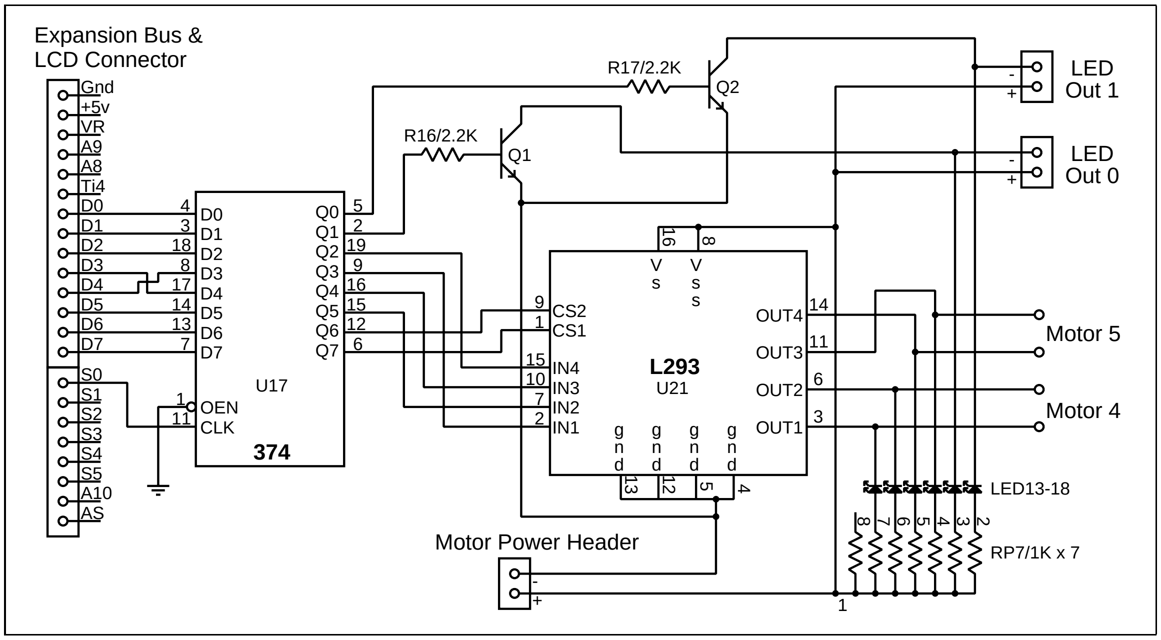


Figure A.15: Expansion Board Motor and LED Circuitry

A.4.1 Motor Current Monitoring Circuit

When the L293 chips drive a motor, there is a voltage drop across the transistors that form the H-bridge. The transistor connected to motor ground (0 volt potential) might drive the motor at some voltage between .2 and .8 volts; the transistor connected to the positive terminal of the battery (say it's at 6 volts) might drive the motor between 5.2 and 5.8 volts.

The amount of this voltage drop is proportional to the amount of current being supplied by the motor-driving transistor. When more current is being supplied, the transistor drops more voltage.

This undesirable property of the L293 transistors is exploited to give a crude measurement of the amount of current being driven through the motor. A fundamental property of motors is that as the amount of work they are performing increases, the amount of current they draw also increases. So the current measurement yields data on how hard the motor is working—if it is turning freely, if it is stalled, or if it is working somewhere in between.

As indicated in Figure A.13, the voltage feedback point is tapped from the indicator LEDs that are connected to the motor outputs. The voltage across the LEDs will decrease as a result of increased current draw of the motor (and the corresponding decreased performance of the L293's). This voltage is fed to a 6811 analog input and can be measured by the 6811 analog-to-digital conversion hardware.

Each of the four motor circuits is wired in this way to a 6811 analog input.

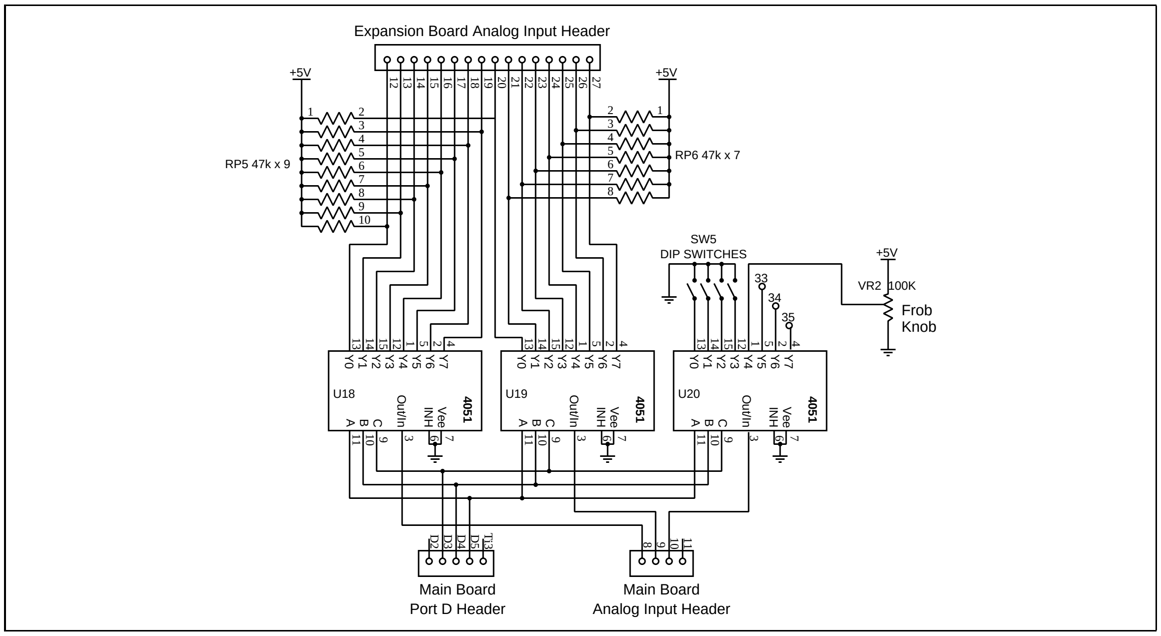


Figure A.16: Expansion Board Analog Input Circuitry

A.4.2 Analog Input Multiplexing on the Expansion Board

The Expansion Board has three *eight-to-one analog multiplexer* ICs. These chips (the 74HC4051) have eight inputs and one output; depending on the state of three selector inputs, one of the eight input lines is connected to the output.[2]

The outputs of the '4051 chips are wired into the 6811 analog inputs when the 6.270 Expansion Board plugs into the main board. Three signals from the 6811 are used to control the multiplexers and select which analog input is mapped to the 6811 analog input[3].

Figure A.16 is a schematic of the analog input circuitry on the 6.270 Expansion Board. It is easy to see how the use of the analog multiplexer chips greatly expands the analog input capability of the 6.270 hardware:

- Two of the '4051 chips have their inputs wired to a bank of sixteen open sensor inputs.
- The other chip is wired from the *Frob Knob*, a general-purpose analog input knob, and four DIP switches (for user configuration input).

[2]Actually, the chip's signals are bidirectional, but for the purpose of this discussion, it is convenient to think of the chip as having eight inputs and one output.

[3]These signals are taken from the 6811's *High Speed Serial Port*, a special sub-system of the 6811 that allows it to communicate at high speeds with other 6811's. In the 6.270 application, this functionality is not needed; instead, the signals are used as simple digital outputs.

- Three of the inputs to this third chip are open, as is one of the 6811's analog inputs.

A.5 The Serial Line Circuit

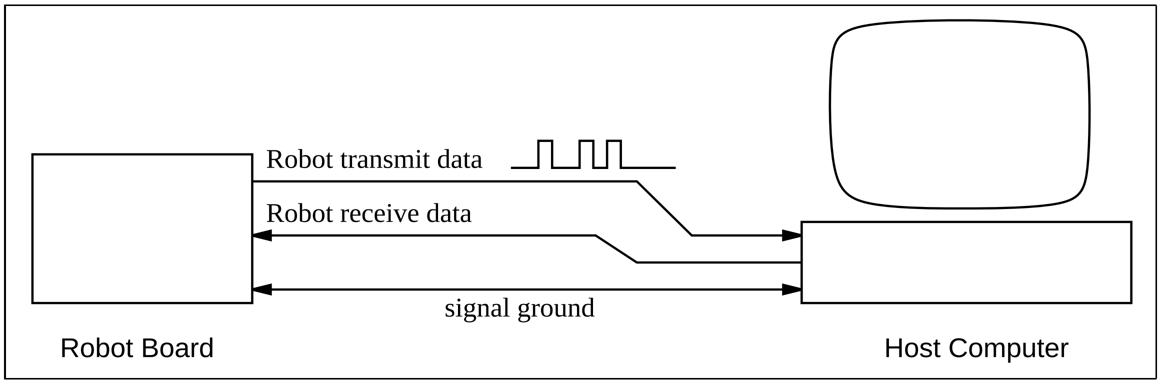


Figure A.17: Host and Board Communications over 3-Wire Serial Link

The 6.270 Board communicates with a host computer over an RS-232 serial line. "RS-232" refers to a standard protocol for communications over a three-wire system, as depicted in Figure A.17. Nearly all of today's computers have serial ports that conform to the RS-232 standard.[4]

In the RS-232 system, a "logic zero" is indicated by a $+15$ volt signal with respect to ground, and a "logic one" is indicated by a -15 volt signal. Note that this is different from standard digital logic levels in several ways. Negative voltages are used, higher voltages are used, and negative voltages connote a logic one value.

The 6811 chip includes circuitry to generate waveforms compatible with the RS-232 systems, but requires external circuitry to convert its own signals, which obey the digital logic norms, to RS-232 signals as described.

There exist off-the-shelf single-chip solutions to this problem (most notably, the MAX232 and MAX233 chips made by Maxim, Inc.), but these chips are typically expensive and consume a fair bit of power. The solution implemented on the 6.270 board requires a few more components, but is significantly cheaper and less power-hungry.

[4]The actual RS-232 standard involves quite a few more wires for conveying various status information, but the data itself is transmitted on two uni-directional wires.

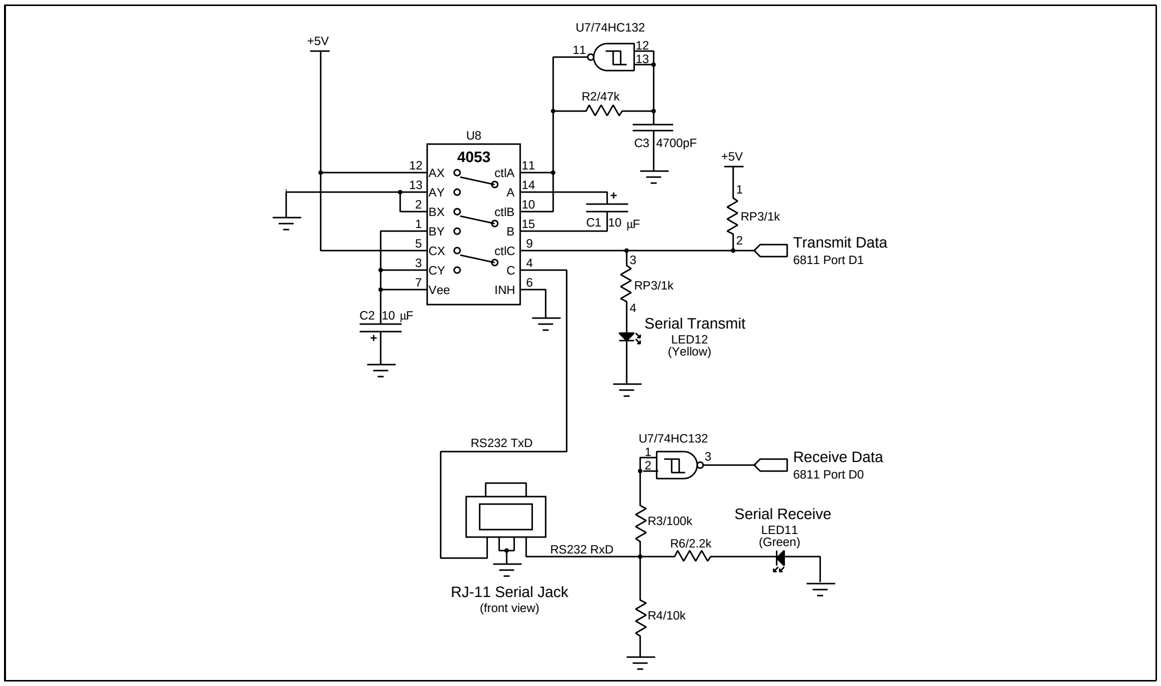


Figure A.18: Serial Line Circuit

A.5.1 Serial Output

One of the difficulties in generating RS-232 signals is obtaining the negative voltage required to transmit a logic one. However, it turns out that the specified -15 volts is not required: -5 volts will do for most applications.

A circuit called a *charge pump* is used to generate this negative voltage. A charge pump consists of two capacitors and a switch. One of the capacitors is charged to a positive voltage by the main power supply. Then the terminals of this capacitor are switched to the terminals of the second capacitor. The first capacitor discharges rapidly into the second, charging it negatively with respect to system ground. This process is switched rapidly, and a steady negative voltage supply is produced in the second capacitor.

The schematic for this circuit and the rest of the serial line circuitry is shown in Figure A.18. The heart of the circuit is a 74HC4053 chip, which is a triple analog SPDT switch that can be controlled digitally.

The charge pump is built from switches A and B of the '4053 chip. Capacitor C1 is charged from system voltage when the switches are in the X position (as is illustrated in the diagram). When the switches are flipped to the Y position, C1 discharges into capacitor C2, creating a negative voltage on C2 with respect to system ground.

The C switch is used to switch either the -5 volts from C2 or $+5$ volts from system power out over the serial line. This is done by wiring the 6811's logic-level "Transmit Data" signal to the control input of switch C.

Switches A and B are repeatedly alternated between the X and Y positions by an oscillator built from a schmitt-trigger NAND gate wired as an inverter (U7) and an RC delay (R2 and C3). This oscillator is tuned to about 10,000 Hertz, a frequency that has been experimentally determined to yield good results.

The commercially-available single-chip solutions mentioned earlier implement a similar circuit. In fact, they use two charge pumps. The first is used to double the system voltage of +5 volts to obtain a +10 volt supply that more closely matches the RS-232 standard. The second charge pump inverts this +10 volts to obtain a −10 volt supply.

A.5.2 Serial Input

A schmitt-trigger NAND gate is wired as an inverter to convert the negative-true RS-232 standard to the positive-true logic level serial standard. Resistor R3 limits the current that can flow into the gate when the serial line voltage is negative, preventing the possibility of damage from a high negative voltage.

The RS-232 standard dictates that a serial line should be in the logic true (negative voltage) state when it is not transmitting data. LED11, the serial receive indicator, is wired such that it will light in this state, being powered directly by the serial voltage generated by the host computer. This LED serves as an indicator that the 6.270 board is properly hooked up to the host.

A.6 Battery-Backing the Static RAM

The static RAM used in the 6.270 board is a special low power device, a relatively recent innovation in widely-available memory technology. This memory chip requires only an infinitesimal amount of current to store its contents when it is not being used.

The actual amount of current—less than one *micro*ampere—is so small that a standard alkaline battery does not notice it. That is, the battery will last as long as its shelf life, whether or not it is supplying one microamp to a circuit. (Alkaline batteries have a shelf life of several years.)

Having a battery-backed static memory greatly increases the usability of the 6.270 board. A robot can simply be turned on and operated immediately, without having to be connected to a computer first.

Unfortunately, implementing a battery-backed RAM can be complicated. The difficulty arises from unpredictabilities in microprocessor behavior when system power is either switched on or off. During these transition periods, the microprocessor is powered by illegal voltages, and its behavior is not defined. In order to make sure that the microprocessor does not corrupt the contents of the memory, orderly transitions from the powered-off to power-on states, and vice-versa, must be implemented.

A.6.1 Powering the Memory Chip

Figure A.14 illustrates how power is always provided to the memory (through diode D2) even when microprocessor and motor power is turned off. Capacitors C5 and C8 help to smooth the power supply of the memory, and also can provide power to the memory while batteries are being changed. Because the current draw is so small, capacitor C5 will actually keep the memory "alive" for periods of up to thirty minutes when the system is powered off and batteries are removed.

A.6.2 The Power-Off Interrupt

Diodes D1, D2, and D3 provide isolation amongst the three parts of the circuit:

- the memory's power supply
- the microprocessor's power supply
- the power-off interrupt circuit

This isolation is necessary to ensure clean transition of the power-off interrupt circuit when power is shut off. The power-smoothing capacitors (both for the memory and for the microprocessor circuit) retain charge for a brief period after power is switched off. The diodes prevent this charge from "flowing backward," and allowing one part of the circuit to power another.

When power is switched off, the power-off interrupt signal *immediately* goes low. However, system capacitors (mostly, C13) will keep the microprocessor powered up for a short while (about about one-tenth of a second).

The interrupt signal generates a hardware-level interrupt to the 6811. A special-purpose software driver is activated, which has the job of shutting down the 6811 in an orderly fashion before the capacitor power supply runs down.

Sometimes, a brief physical jolt to the microprocessor board will dislodge a battery momentarily, causing the interrupt to be triggered. It would be incorrect for the software to shut down the system in this case. So, the interrupt software waits for a short while to see if the interrupt line goes high (indicating that power has returned). If power does return, the interrupt exits without taking action.

If power does not return after about one-hundredth of a second, the software routine executes a machine-language HALT instruction, which shuts off the microprocessor. This sequence of actions implements an orderly shutdown sequence.

A.6.3 The Power-Up Delays

The Dallas Reset Chip (U11) holds the reset line low for 350 ms after the logic power reaches a level of at least 4.25 Volts. This prevents the 6811 from trying to operate with indeterminate voltages at its inputs, and it safeguards the SRAM while power levels settle.

Once power has normalized, the Dallas chip allows the reset line to rise. The MODA pin has reached a valid logic 1, and the microprocessor comes up in the "run" state. This circuit is shown in Figure A.19.

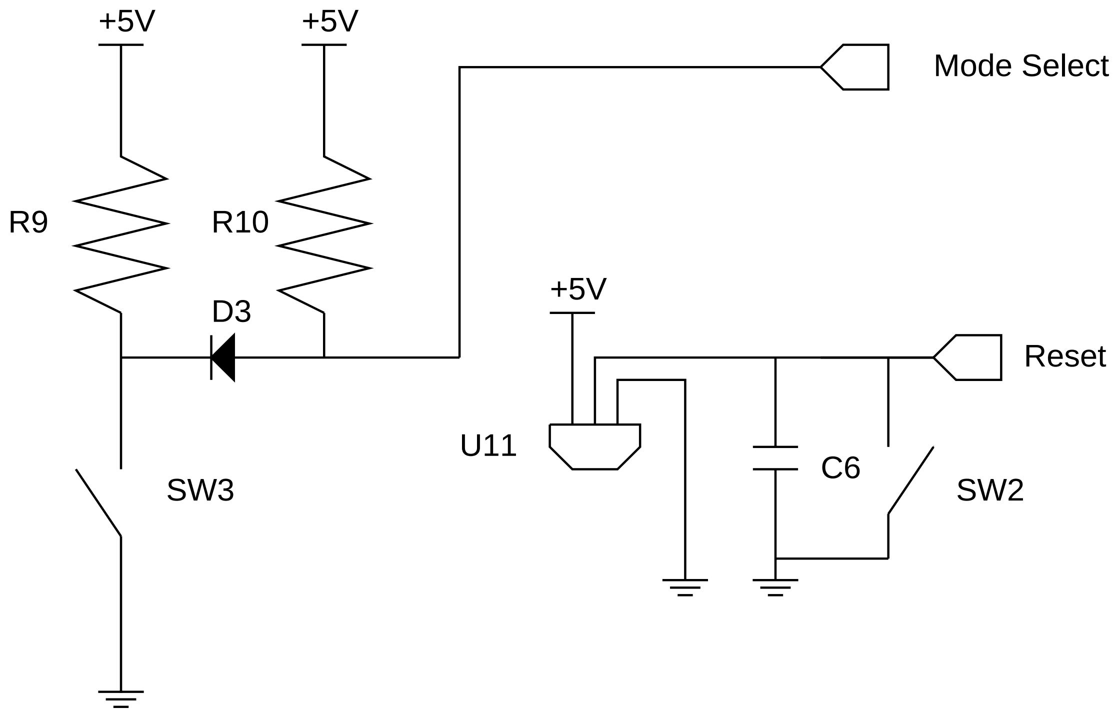


Figure A.19: Reset Circuitry

A second mode is used to download the operating system software to the microprocessor when initializing the board. To bring the processor up in this mode, the MODA must be a logic zero when the processor comes out of the reset state. This happens when the choose button is depressed while the reset occurs. This will put the 6811 into a bootstrap download mode in which a program is executed from internal ROM rather than external RAM.

Diode D3 allows the choose button to pull MODA low without forcing MODA to follow the state of the choose button at all times. Resistor R10 pulls MODA to high when nothing else is going on, so normal run mode is the default after a reset.

If the user presses reset without the choose button, the Dallas chip will pull the reset line low and MODA will remain high. The 6811 will go into the normal run mode, executing a program in external memory.

In order to ensure that the 6811 does not access external memory until the reset conditon is clear, the reset line is also an input to the RAM enable logic.

A.7 The Infrared Transmission Circuit

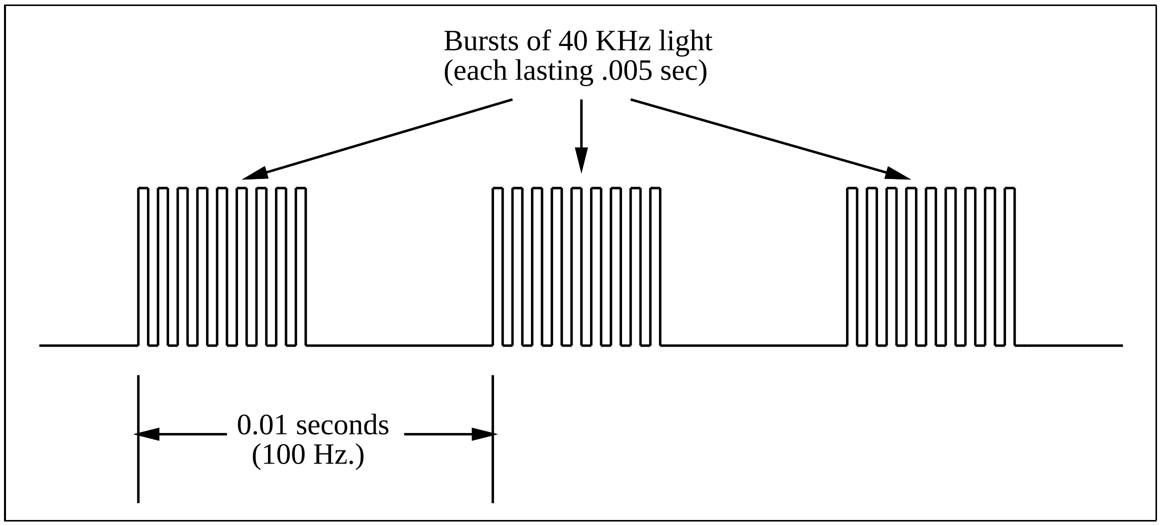


Figure A.20: Square Wave Consisting of Bursts of 40 Khz Signals

The Sharp GP1U52 sensor, and others like it commonly used in TVs, VCRs, and other devices controlled by infrared, is sensitive to *modulated* infrared light. It detects the presence of infrared light that is blinking on and off at a particular rate. The GP1U52 sensor is tuned to 40,000 Hertz (40 KHz).

In TV remote applications, a data stream is then generated around the 40 KHz carrier frequency. The signal consists of bursts and gaps of the 40 KHz transmissions.

For the 6.270 application, the 40 KHz carrier is used to transmit a square wave of relatively low frequency (100 or 125 Hz), as shown in Figure A.20. When the Sharp IR sensor decodes this signal, it removes the 40 KHz carrier, yielding a copy of the square wave that was originally transmitted (Figure A.21).

Software can continuously check the Sharp sensors for square waves of the specified frequency. It can lock on to the square wave when it is present and count the number of consecutive cycles that have been detected.

A special circuit is used to generate infrared emissions modulated at the 40 KHz frequency. A block diagram of this circuit is shown in Figure A.22.

The diagram shows that the '390 chip, wired in a divide-by-fifty configuration, is used to generate a 40 Khz signal from the 6811 E clock, a 2 Mhz signal. In actuality,

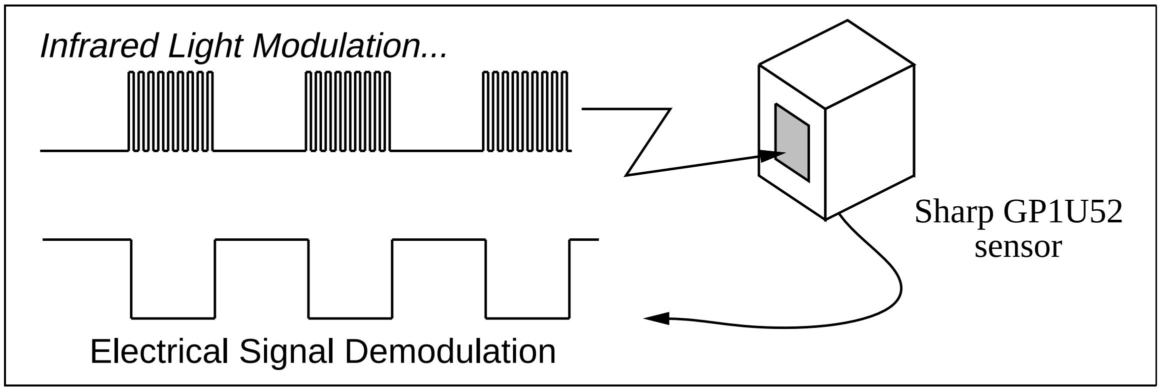


Figure A.21: Sharp IR Sensor Decoding IR-Encoded Square Wave

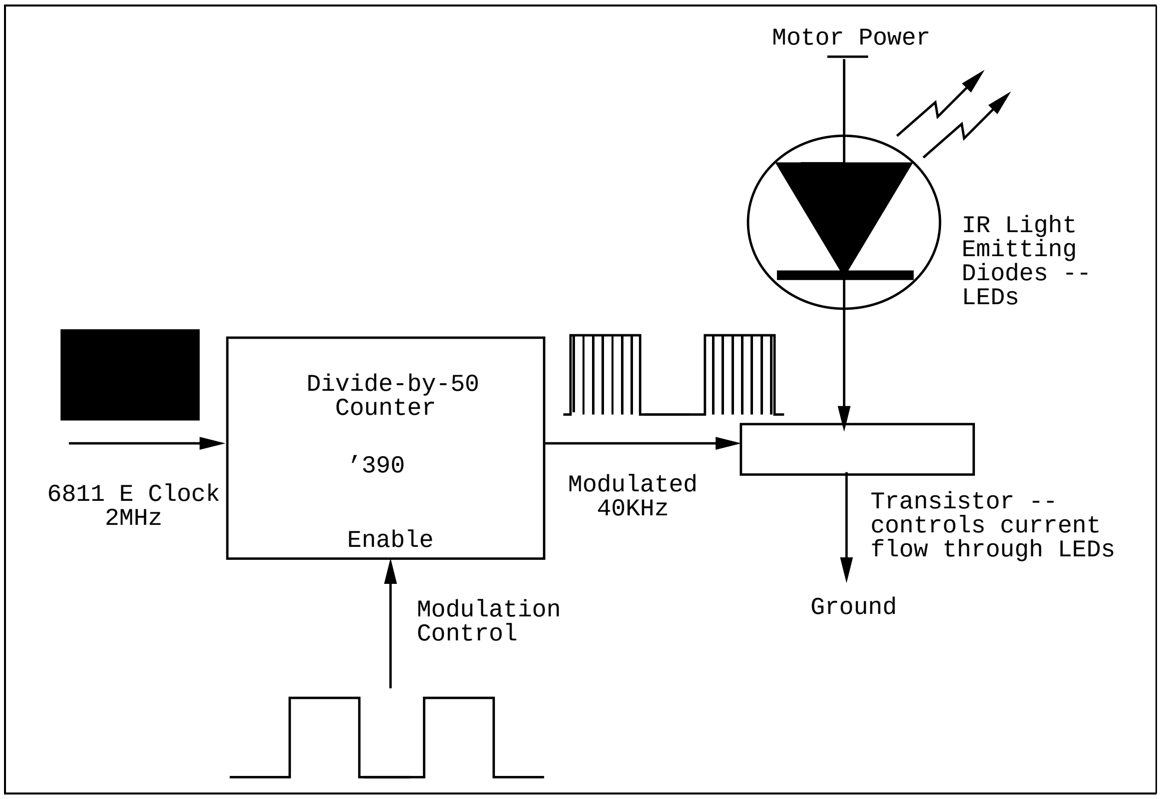


Figure A.22: Block Diagram of Infrared Circuitry

the '390 chip contains two *decade counters.* Each these consists of a separate divide-by-five counter and a flip-flop (a divide-by-two device). The '390 is wired in the divide-by-fifty function by ganging two of the divide-by-five counters and one of the flip-flops.

The *IR control* signal is wired to the clear input of the '390 chip; when this signal is low, the counters will reset and will be prevented from counting. By modulating this signal, the 6811 can generate the low-frequency square wave that ends up being transmitted to the Sharp sensor.

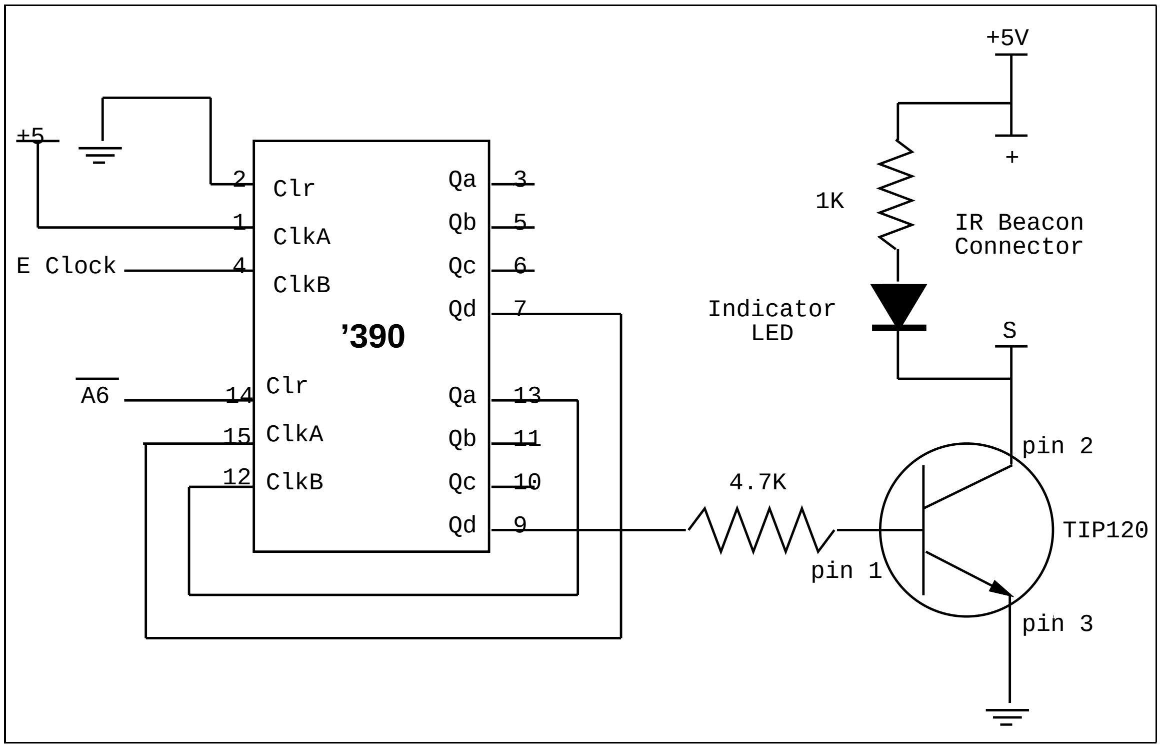


Figure A.23: Infrared Transmission Circuit

Figure A.23 shows the full circuit schematic for the IR subsystem.

The TIP120, a power transistor, is used to drive the infrared LEDs. The output of the '390 chip is driven into the base of the TIP120. When this signal is high there is a positive differential between the base and the emitter of the TIP120 and current is allowed to flow from the collector to the emitter, thus driving current through the IR emitter. When this signal is low, the base and emitter are at the same differential and no current flows. This causes no current to be allowed to flow from the collector to the emitter and the IR LEDs have no current flowing through them so they turn off.

A.7.1 The IR Beacon

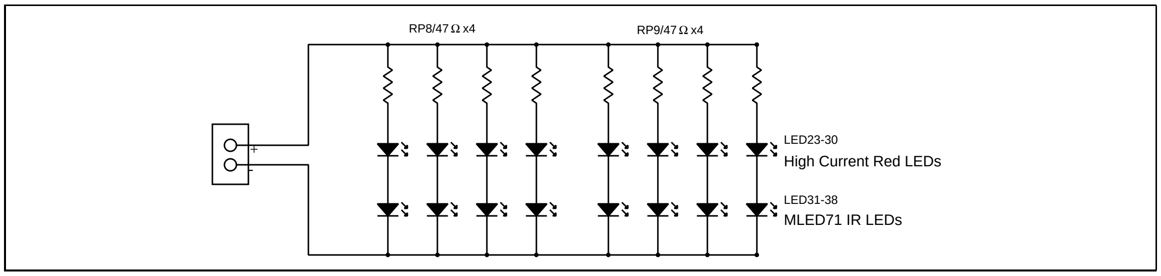


Figure A.24: Infrared Beacon Circuit

Figure A.24 shows the schematic for the IR beacon. Each infrared LED has a visible LED in series with it so it should be easy to ascertain that the device is transmitting infrared light properly. The resistors act as current-limiters, limiting the amount of current that can travel through any branch of the circuit to between 10 to 20 mA.

A.8 The LCD Display

The first fourteen pins of the 6.270 Board's Expansion Bus are designed to be compatible with a 14-pin standard LCD bus. A variety of character-based LCD devices with different screen sizes use this standard bus.

The LCD bus standard is fairly simple, consisting of the following signals:

- an 8-bit data bidirectional bus
- two mode select input signals
- a clock line
- a voltage reference for contrast adjustment
- +5 volt logic power
- signal ground

In fact, reading and writing data to an LCD is much like reading and writing data to latches or to memory. There is one problem, however: LCDs only work at data transfer rates up to 1 MHz. The 6811 in the 6.270 board operates at 2 MHz—too fast for most LCDs.

One straight-forward solution to the speed problem would be to use a '374-type latch between the 6811 and the LCD. The '374 could be written to at the full bus rate of the 6811; its outputs would drive the data bus of the LCD. A separate signal could be used to toggle the LCDs clock line, causing it to latch the data that had been written to the '374[5].

An unconventional, zero-additional-hardware solution has been implemented in the 6.270 system, which takes advantage of an obscure feature of the 6811 microprocessor.

The 6811 has two main operating modes, known as *single chip mode* and *expanded multiplexed mode*. The discussion of memory read and write cycles that has been presented in this chapter has been based on the expanded multiplexed mode, which is the 6811 mode that is used when external memory is part of the 6811 circuit.

When the 6811 is operated in single-chip mode, the upper-eight-bit address bus and multiplexed address/data bus become general purpose inputs and outputs of the 6811, controllable by system software. Thus, in single-chip mode, the 6811 could communicate with the LCD with a software driver, rather than the too-fast hardware communication.

There is a problem with this, however: when the 6811 is placed into single-chip mode, it can no longer execute a program from its external RAM. In fact, as far as the 6811 is concerned, there *is no* external memory anymore.

Fortunately, the 6811 has 256 bytes of internal RAM, from which it can execute a program when in single-chip mode. Thus, a software driver could execute out of internal RAM, perform a transaction with the LCD, and then switch back to expanded-multiplexed mode and return control to the main program in external memory.

The obscure feature mentioned is not the fact that the 6811 has both of these modes, but the idea of dynamically switching between them. Here is the solution that has been implemented:

1. Start by copying a software driver from external system memory into the 256 bytes of internal 6811 memory.

2. Begin execution of the driver program located in internal memory:

   - Place the 6811 into single-chip mode; external memory disappears.
   - Execute a low-speed transaction with the LCD by directly controlling the data bus via software.
   - Place the 6811 into expanded-multiplexed mode.
   - Return to the main program in external memory.

3. Continue normal program execution.

[5]This solution assumes that one does not need to read status data back from the LCD.

The actual LCD driver routine buffers characters to be printed to the LCD; one thousand times per second, an interrupt routine calls the internal memory driver as described, writing a single character to the LCD. The whole process operates transparently to the 6.270 system user.

A.9 The Low-Battery Indicator

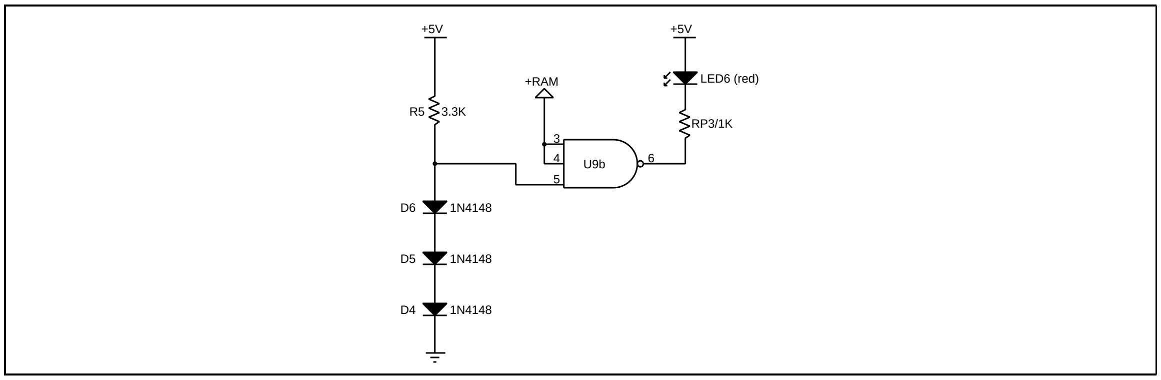


Figure A.25: Low Battery Indicator Circuit

A spare gate on U9 has been used to implement a low-battery indicator. The schematic is shown in Figure A.25.

The transition point for determining if a digital input is logic one or logic zero is normally one-half of the supply voltage. Assuming a 5 volt supply, signals greater than 2.5 volts will be interpreted as logic ones, and signals less than 2.5 volts will be interpreted as logic zeros.

Diodes have the interesting property that they drop exactly 0.6 volts when current travels through them. Thus the input voltage to the gate U9b will be about 1.8 volts, over a wide range of system supply voltages.

Assuming a 5 volt supply, this input would to be interpreted as logic zero. U9b is wired as an inverter, so it will output a logic one. Since the LED is wired from supply voltage, it will be off in this state.

Suppose supply voltage falls to 3.5 volts. Now the transition point is around 1.75 volts. The input to the gate is 1.8 volts, so it becomes a logic one. U9b inverts this to obtain a logic zero, and drives zero volts on its output, lighting the LED.

The actual transition point in the circuit is closer to 4 volts, because the diodes tend to drop a bit more than 0.6 volts that are usually specified. Surprisingly, nearly all of the 6.270 electronics, including the 6811 microprocessor, work fine at voltages as low as 4 volts. One notably exception is the Sharp GP1U52 sensor: its performance decreases sharply at supply voltages less than 4.5 volts.

A.10 Fun Hacks

Many people have created clever hardware hacks to the 6.270 board. Hopefully this section will grow as more and more features are found to further develop the board.

A.10.1 Adding a Loudspeaker

There are two rather simple ways to add an external loudspeaker to the 6.270 board. Teams have done this to play music loudly. However, it requires some minor hardware modifications, in particular, the loss of at least one unidirectional motor port on the Expansion Board. The first version of this hack uses both sides of Motor 5. The second version uses only the right unidirectional side of Motor 5.

Speaker Hack, v1.0

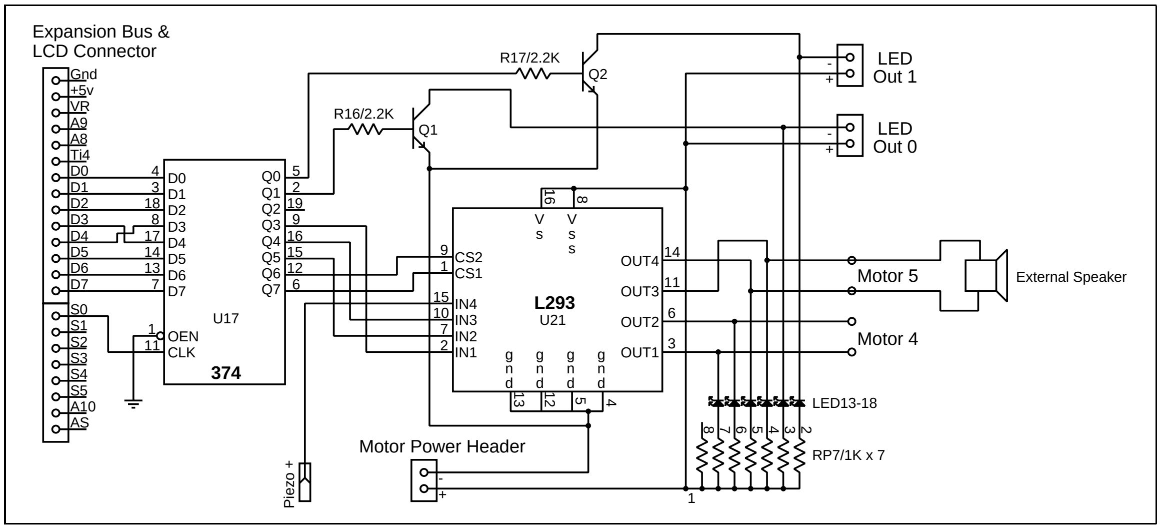


Figure A.26: Speaker Hack, v1.0

Steps for Speaker Hack v1.0:

- ◯ Find Component Side of Expansion Board.
- ◯ Find trace under the 74HC374 socket, from pin 19 of the '374 to pin 15 of the L293D. This trace extends underneath the '374 socket left toward the VR2 pot, then up through the false LCD Connector. There is an unused hole, immediately left and slightly above pin 20 of the '374 socket that this trace runs through.
- ◯ Cut the trace between the unused hole and the '374 socket with a sharp knife.

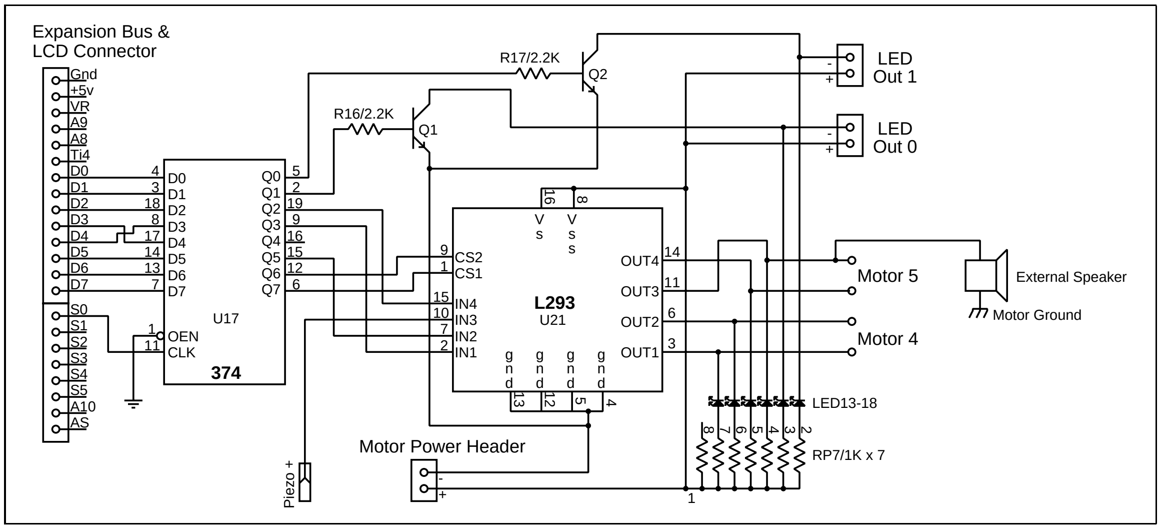


Figure A.27: Speaker Hack, v2.0

◯ Place a single Female Header pin into the unused hole. This pin should now connect with pin 15 of the L293D.

◯ Run a wire from the + signal of the Piezo on the Controller Board into the Female Header you placed on the expansion board. There are several unused holes on the Controller Board for the piezo that you can solder to. Use Male Header on this signal wire so you can disconnect the Expansion Board from the Controller Board.

◯ Plug your 8Ω external speaker into the bidirectional Motor 5 port. You may wish to put a 100Ω-200Ω pot in series with the 8Ω speaker as a volume control.

◯ In IC, use the command:

```
fd(5);
```

to turn on the speaker. Use the command:

```
off(5);
```

to turn off the speaker.

Speaker Hack, v2.0

Steps for Speaker Hack v2.0

- ◯ Find Component Side of Expansion Board.
- ◯ Cut the trace from pin 16 of the '374 to pin 10 of the L293D. This trace should be cut immediately above the '374 socket at pin 16.
- ◯ Find the column of holes on the bottom right corner of the prototyping area, immediately to the left of the U19 label. Look one column left of that, directly above pin 10 of the L293D. Solder a single Female Header pin there.
- ◯ Make a solder bridge between the Female Header pin you just placed and pin 10 of the L293D immediately below it.
- ◯ Run a wire from the + signal of the Piezo on the Controller Board into the Female Header you placed on the expansion board. There are several unused holes on the Controller Board for the piezo that you can solder to. Use Male Header on this signal wire so you can disconnect the Expansion Board from the Controller Board.
- ◯ Plug your 8Ω external speaker into the bidirectional Motor 5 port. You may wish to put a 100Ω-200Ω pot in series with the 8Ω speaker as a volume control.

  The right unidirectional motor port of Motor 5 is now wired to drive the speaker. The left unidirectional motor port of Motor 5 may still be used as normal.
- ◯ In IC, use the command:

  ```
  motor5_right(1);
  ```

  to turn on the speaker. Use the command:

  ```
  motor_right(0);
  ```

  to turn off the speaker.

Appendix B

Printed Circuit Layouts

This section has the printed circuit board artwork patterns for the 6.270 Rev. 2.21 boards:

- the Microprocessor Board
- the Expansion Board
- the Battery Charger Board
- the Motor Switch Board
- the Infrared Beacon Board

The board artworks are provided to facilitate debugging; they are not intended to serve as master artworks for fabricating new printed circuit boards. The layouts are reproduced at actual size given the limits of reproduction technology.

B.1 Microprocessor Board

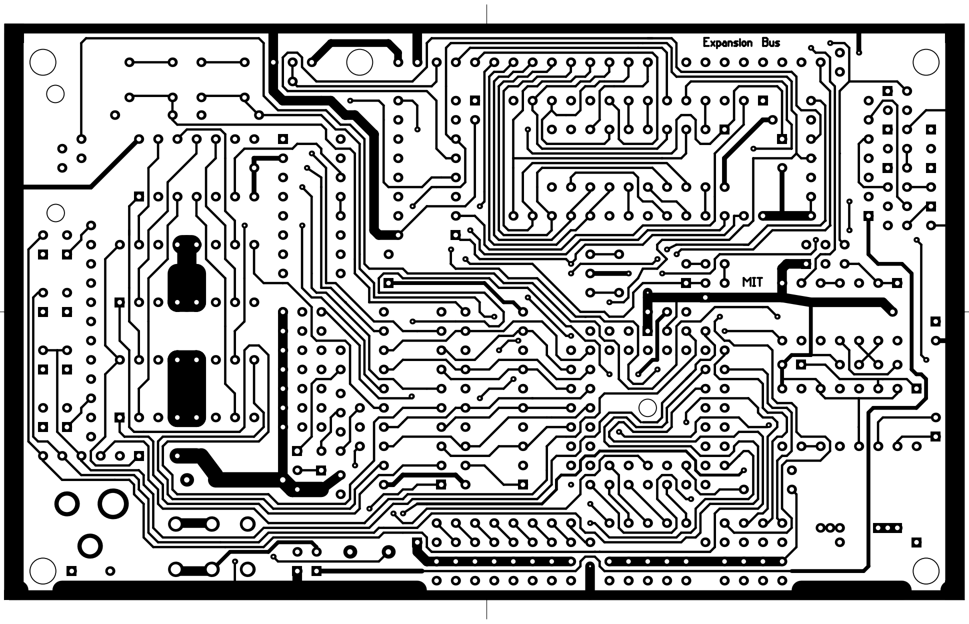


Figure B.1: Microprocessor Board, Component Side

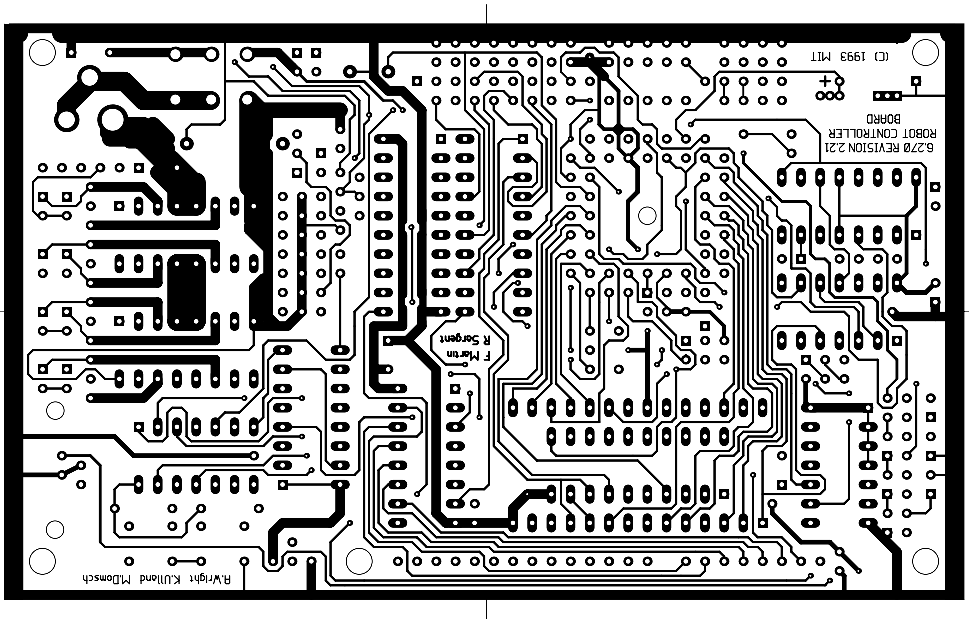


Figure B.2: Microprocessor Board, Solder Side

B.2 Expansion Board

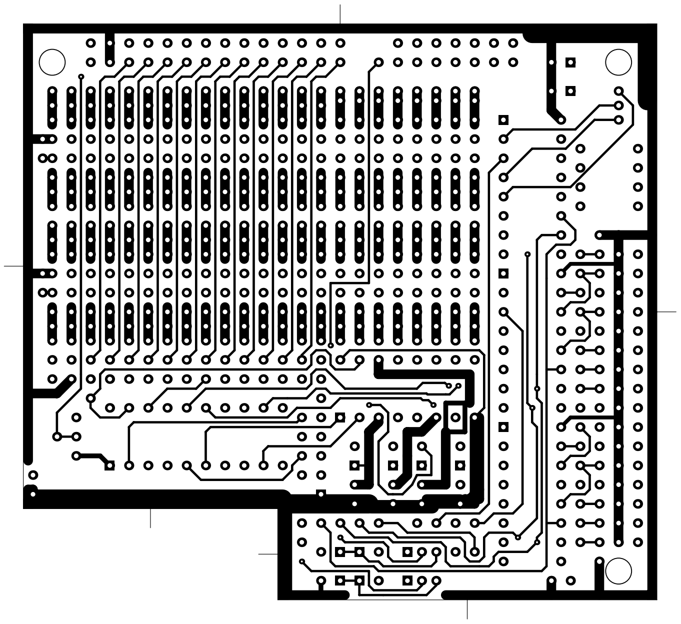


Figure B.3: Expansion Board, Component Side

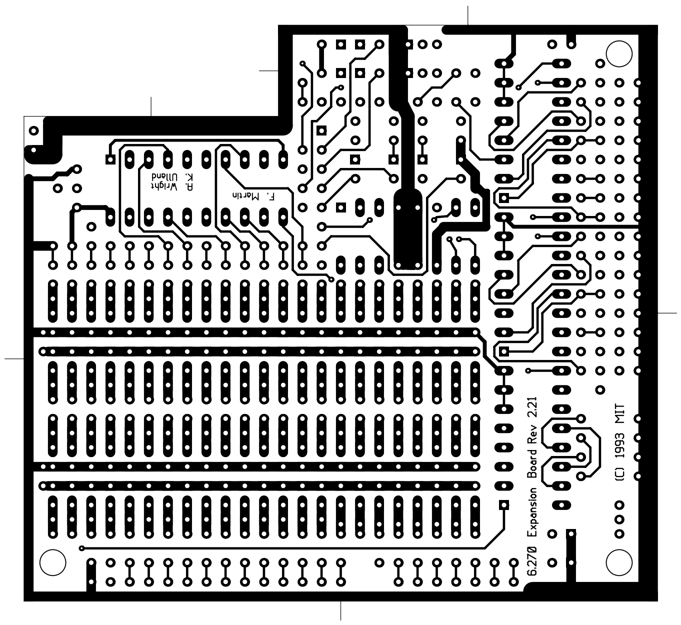


Figure B.4: Expansion Board, Solder Side

B.3 Battery Charger Board

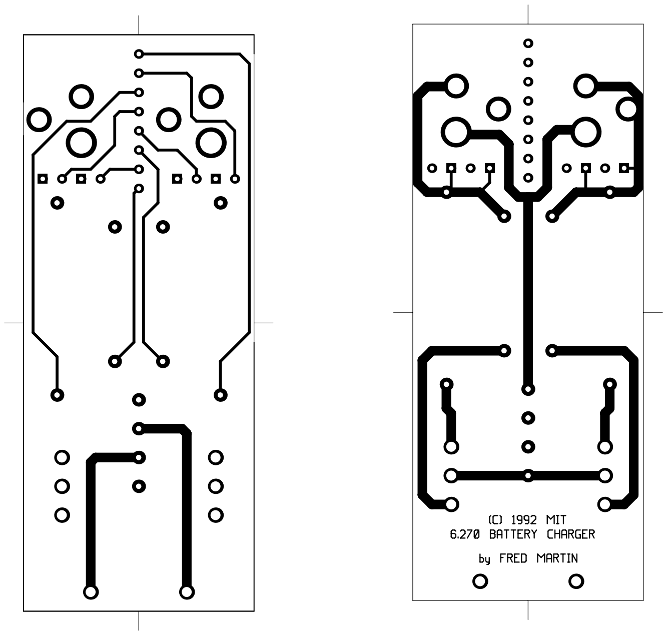


Figure B.5: Battery Charger Board, Component and Solder Sides

B.4 Motor Switch Board

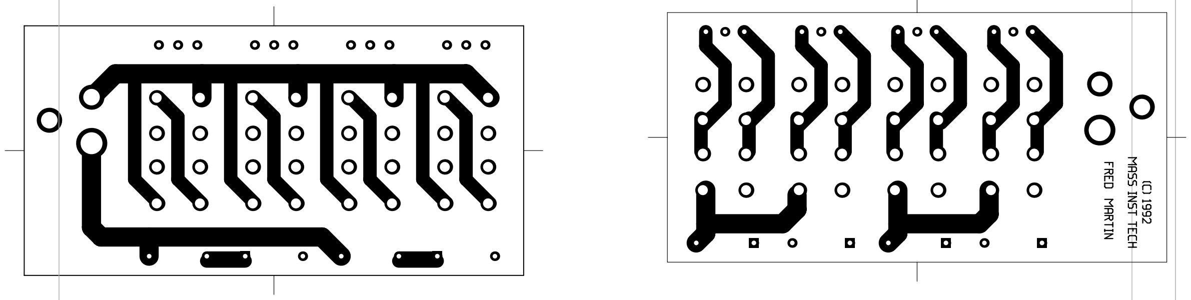


Figure B.6: Motor Switch Board, Component and Solder Sides

B.5 Infrared Beacon Board

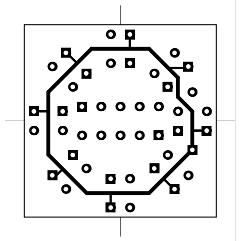
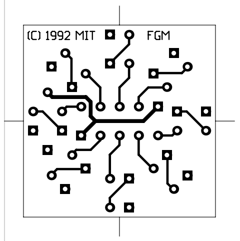



Figure B.7: Infrared Beacon Board, Component and Solder Sides

Appendix C

Batteries

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

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

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

C.1 Cell Characteristics

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

C.1.1 Voltage

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

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

C.1.2 Capacity

In general, the larger a cell is, the more electricity it can supply. This *cell capacity* is measured in *ampere-hours*, which are the number of hours that the cell can supply a certain amount of current before its voltage drops below a predetermined threshold value.

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

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

C.1.3 Power Density

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

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

C.1.4 Discharge Curve

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

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

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

C.1.5 Internal Resistance

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

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

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

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

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

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

C.1.6 Rechargeability

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

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

The Memory Effect

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

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

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

C.1.7 Cost

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

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

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

Figure C.1 summarizes the characteristics of commonly available cell technologies.

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

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

Figure C.1: Table of Cell Characteristics

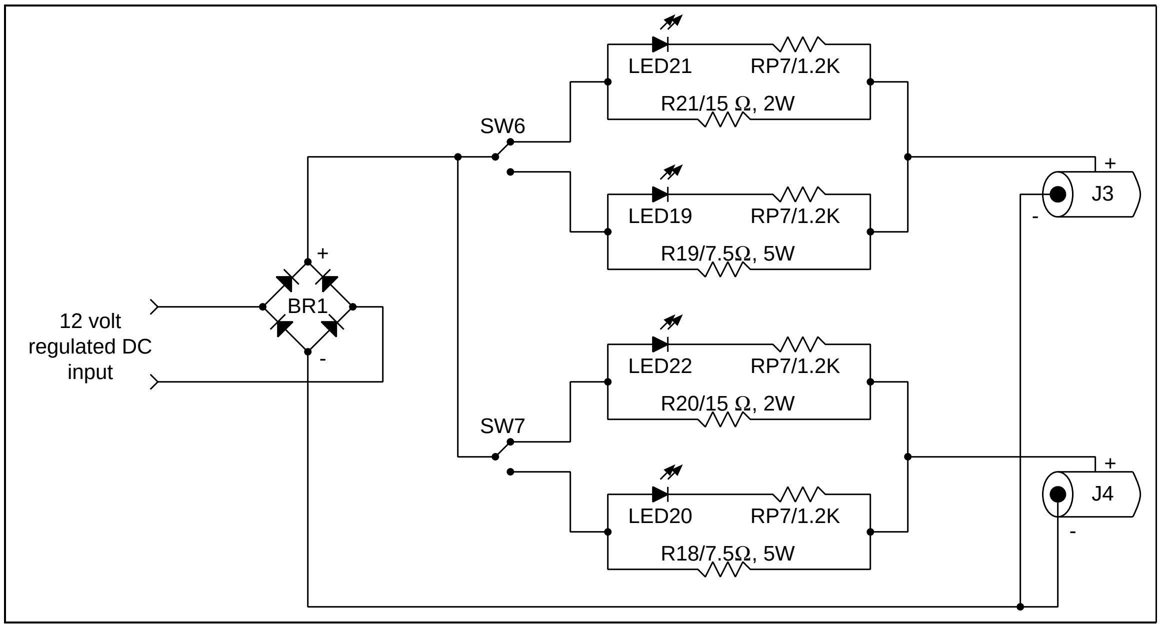


Figure C.2: Battery Charger Schematic Diagram

C.2 Battery Packs

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

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

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

C.3 6.270 Battery Charger

The rule of thumb for charging batteries is to charge them at a rate equal to one-tenth of the amp-hour capacity of the battery. For example, if a battery is rated for 2.5

amp-hours (as are the Hawker cells included in the 6.270 kit), then it would normally be charged at a rate of 250 milliamps.

Figure C.2 shows the schematic diagram of the battery recharger. The essence of the charger is simply a resistor in series with the battery hooked up to a regulated voltage power supply.

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

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

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

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

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

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

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