Contents

Chapter 1

Introduction

1.1 Emerging Robots

Piece by piece, robots are creeping into our lives.

Parts of robots can now be found almost everywhere. Nearly all automobiles manufactured today contain robot brains, highly integrated microcontrollers. These tiny computers have analog-to-digital converters, timers, and other features integrated in a single chip. In cars microcontrollers examine many aspects of engine operation, then command adjustments to optimize performance.

The eyes and ears of robots—photocells, infrared detectors, and other sensors—scan our homes for signs of intruders. Other such devices watch patiently for hands to appear under the faucet in an airport restroom—hands detected, the water goes on.

The muscles of robots—servo motors and solenoids—dispense soda from vending machines and move magnetic tape through the internal labyrinths of handheld video recorders.

Individually, these component parts of robots are commonplace and taken for granted. What remains uncommon is finding microcontrollers, sensors, and motors that have been assembled into a functioning autonomous robot. In everyday life we do not yet encounter machines moving about on their own, independently performing tasks that we consider important.

This situation is changing. Recent advances in the theory of robot programming, continuing progress in microelectronics, and the miniaturization of sensors have brought robots to a critical point. We are entering a time when robots will begin to leave the laboratory and

move into our offices and homes. Soon you may purchase luggage that can follow you through the terminal to your plane or you may choose an intelligent vacuum cleaner that operates when no one is home, keeping your floors clean with no effort on your part.

1.2 Declining Cost

A great deal of work remains to be done before these wonders can be fully realized. However, because of the advances mentioned above, the cost of learning about robots and conducting research in robotics is lower now than ever before. Just a few years ago, with startup costs in the tens of thousands of dollars, only a few well funded research institutions could contemplate doing experimental work with robots. Today, a serious program of robotics research or education can be begun for well under $1000. The door has been thrown open to participation in this vital and exciting field!

Rug Warrior and its accompanying text, *Mobile Robots: Inspiration to Implementation* (A K Peters, Ltd., 1993, ISBN 1-56881-011-3), were developed by researchers from the Artificial Intelligence Lab at the Massachusetts Institute of Technology. The purpose of both the robot and the book is to transfer newly developed technology from the MIT AI Lab and elsewhere to a wide audience. Toward this end, we designed Rug Warrior to be as inexpensive as possible without sacrificing technical sophistication.

1.3 Rug Warrior

Rug Warrior is offered in different packages to meet the needs of a diverse user community. The assembly of the Brains Kit, the Brawn Kit, and the integration of the Brains and Brawn components in the Expanded Kit are all described in this manual.

The Brains Kit provides the intelligence system builders need to construct highly capable mobile robots using their own motors and other mechanical components. The circuitry of the Brains Kit is described in great detail in *Mobile Robots: Inspiration to Implementation.*

In the Brawn Kit, builders have a rugged, uncomplicated platform to which they can add their own controlling circuitry. Such a control system might be composed of a hardwired, discrete component

circuit or a single board computer (plus required auxiliary devices) obtainable from any of a large number of vendors.

The Expanded Kit (the Brains plus Brawn Kits) provides a fully integrated robot, ready for use in education or research. If desired, builders may enhance the Expanded Kit with the addition of, say, a gripper, sonar sensor or other devices of their own design.

1.4 Guide Summary

The logic and interface circuitry of the Brains Kit are factory assembled which allows you to begin programming immediately. Read Chapter 2 to learn the details of how to operate the Rug Warrior circuit board and program it from your computer. Figure 1.1 shows the circuit board's layout of components and sockets.

Rug Warrior's sensor circuits are left to the builder to construct. Step-by-step instructions in Chapter 3 guide you through this process. Self-test software, provided with the Brains Kit, allows you to verify the functionality of each subsystem as you build it.

Interactive C (IC), a powerful, easy-to-learn language, is the recommended programming environment for Rug Warrior. IC is included with the Brains and Expanded Kits. Writing and debugging robot programs is greatly simplified by IC's interactive nature and built-in multitasking. The Rug Warrior self-test program, demonstration program, and other example programs provided with the robot are written in IC. You are, however, not restricted to programming your robot using this system. Full access is provided to the Motorola MC68HC11 microcontroller—it can be programmed using any compatible system.

Construction of the Brawn Kit and integration of the Brains and Brawn Kits are described in Chapter 4.

One of the most exciting aspects of robot building is implementing sensor and actuator systems of your own design. To accommodate this feature, Rug Warrior offers built-in expansion capabilities. Consult Chapter 5 to learn more about this option.

Chapter 6 offers a number of suggestions for diagnosing and correcting problems that sometimes crop up during or after construction.

The Interactive C Manual (Chapter 7) describes the IC programming system. The appendices explain the supplied set of Rug Warrior specific library routines and provide other useful information.

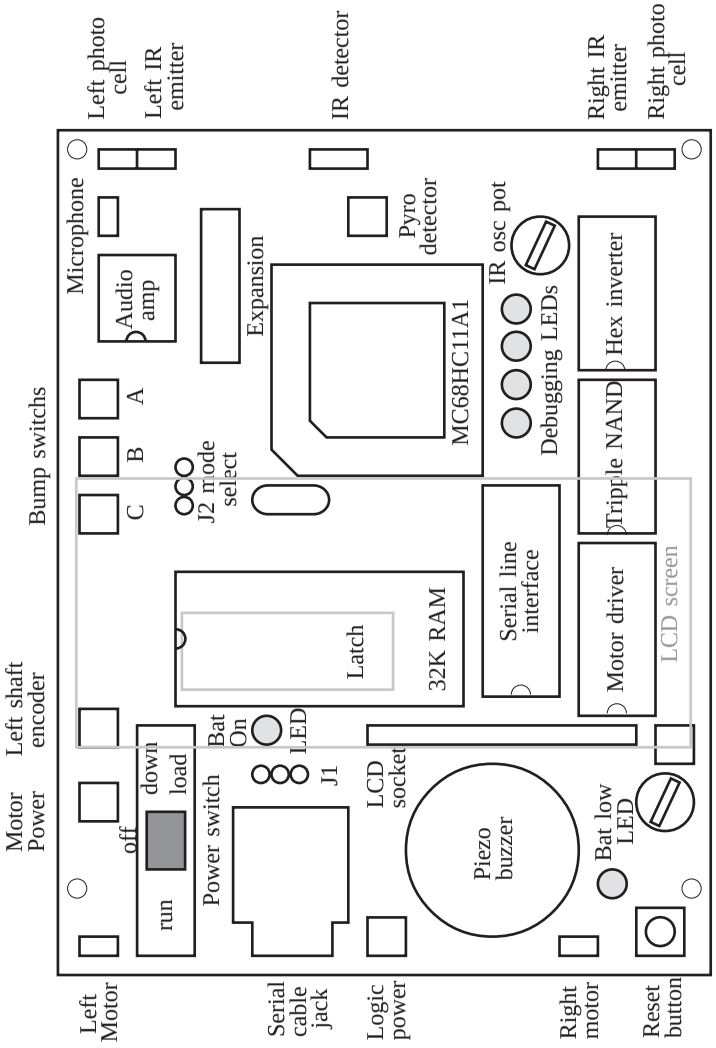

Figure 1.1: Layout of the Rug Warrior circuit board (Rev 3.1).

Chapter 2

Getting Started with the Brains Kit

We recommend that you verify the functionality of your board and familiarize yourself with the self test features before installing the remainder of the sensor circuitry or adding your own customizing touches. To accomplish this end you must complete the following steps:

1. Install the programming environment, Interactive C (IC), on your computer.
2. Connect battery power to the circuit board.
3. Connect a serial cable from your computer to the Rug Warrior circuit board.
4. Initialize the board by downloading the pcode (the operating system).
5. Run IC on your computer establishing communication with Rug Warrior.

Each step is explained in this chapter.

2.1 Software Installation

The recommended programming environment for Rug Warrior, Interactive C, is supplied on the software disk included with the kit. To get

started using IC, follow the directions below. Alternately, you may read and follow the directions in the `README.IC` file on the software disk.

2.1.1 Macintosh Instructions

Create a new folder called "IC" on your hard disk and drag the contents of the software disk into that folder.

2.1.2 DOS Instructions

Use `xcopy` to copy the files from the software disk to a directory called `ic` on your hard disk. If the software disk is in drive `a:` the following dialog is applicable:

```
C:> xcopy a:*.* c:\ic
Does IC specify a file name
or directory name on the target
(F = file, D = directory)? d
```

2.1.3 Windows Instructions

See the file `README.IC` file on the software disk.

2.2 Power and LCD Connection

Remove the circuit board from its protective antistatic envelope. Install the LCD screen into the raised 14-pin connector on the Rug Warrior circuit board. (In some cases this assembly step will have been done at the factory.) See Figure 1.1. One edge of the LCD screen may rest on the case of the 8.000 MHz crystal (the silver rectangular component at the upper left corner of the MC68HC11 microcontroller). Place a piece of electrical tape across the top of the crystal to prevent any possibility of shorting the traces on the underside of the LCD display against the crystal case.

Insert four AA alkaline batteries into the battery holder provided. See that the power switch on the board is in its center OFF position. Next, making certain that the circuit board does not rest on a conductive surface or conductive material (wires, solder, etc.), connect the cable from the battery holder to the 2-pin polarized plug on the left side of the Rug Warrior board. The plug is labeled "MOT PWR."

2.3 Serial Connection

The cable assembly supplied with the Brains Kit has a DB-25 female connector on one end and a 6-4 phone plug, RJ-11, connector on the other. Use this cable to make the connection between your computer's serial port and the Rug Warrior board. See Figure 2.1. The female DB-25 connector plugs directly into the serial connector on many IBM-PC compatible computers. If your computer has a 9-pin connector you must use either a modem cable or a 9 to 25 pin adaptor (Radio Shack part 26-209A) in order to connect to the Brains Kit cable assembly. If your computer is a Macintosh you must use a standard modem cable, DIN-8 to DB-25, to connect to the Brains Kit cable assembly.

Some computers have phone jacks associated with their internal modems. *Do not* attempt to connect Rug Warrior to your computer using this jack. Modems produce modulated tones suitable for use with phone lines, not the logic signals required by Rug Warrior.

Consult your computer owner's manual for information on how to configure your serial line. Notebook computers in particular often direct the output of their serial lines to an internal modem. If this is the case for your computer, you must change the computer's default settings so that "external modem" is selected. If you typically have some other device connected to your serial line, say a mouse or a local area network, you will most likely need to disable the associated software drivers. Otherwise, IC may not have access to the serial line.

2.4 Downloading Pcode

You can think of the pcode as the operating system of the robot. The pcode makes it possible for the robot to run programs written in IC. Normally, the pcode is resident in the on-board static RAM, maintained there by the battery backup system. The pcode will vanish, however, if batteries are disconnected from the robot for more than a few minutes. The pcode may occasionally become corrupted during program development or robot assembly. If this occurs the pcode must be reloaded into RAM.

To prepare the circuit board for downloading, connect the power and serial cables as described above and move the power switch to the right, toward the position labeled "DL" on the board. The green LED, labeled "BAT ON," should be illuminated. Next press the

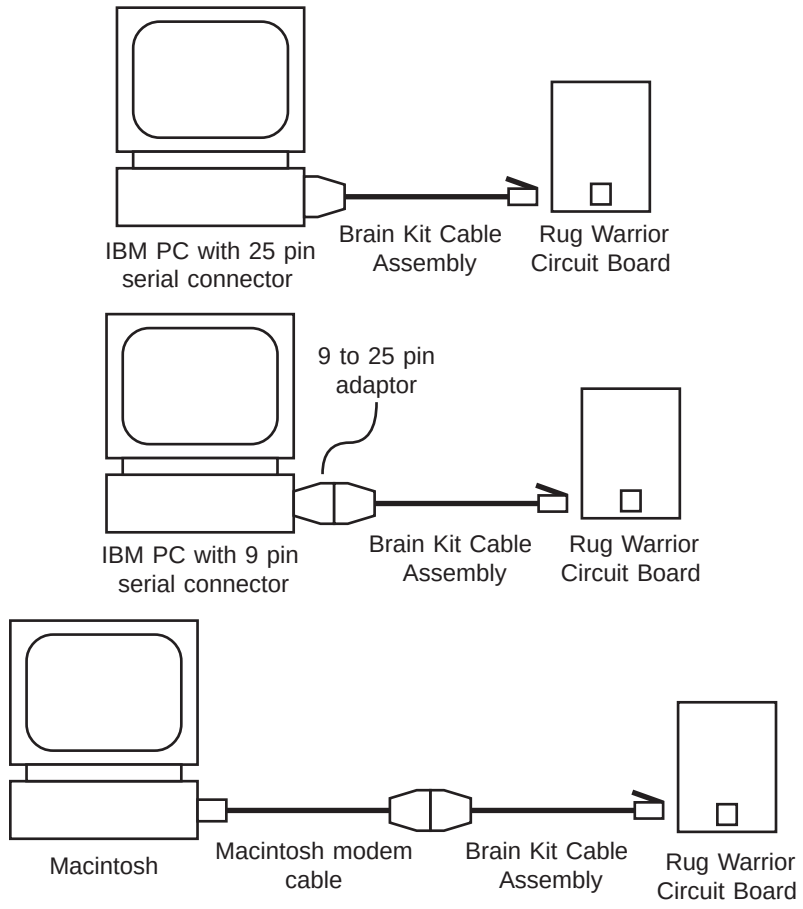

Figure 2.1: Options for connecting the host computer to the Rug Warrior circuit board.

RESET button. The red "BAT LOW" LED should remain lighted for as long as the RESET button is held down; it should go off when RESET is released.

To download pcode from a Macintosh computer, find the file labeled "Init Board RW 7.2 modem" and double click on its icon. (As an alternative, you may connect Rug Warrior to the printer port and double click on file "Init Board RW 7.2 printer.")

To download from a DOS computer, type: `dl pcoderwl.s19`. Options for downloading from computers configured in the standard way are described in the `README.IC` file.

The pcode downloading program running on your computer keeps you informed of its progress. In the first phase of downloading a special 256-byte loader program is sent to the microcontroller at 1200 baud. A dot (or dash) is typed to the screen each time your computer transmits a character. The purpose of this loader program is to tell the microcontroller how to load the pcode that comes next. When the last byte of the loader program has been received, the microcontroller automatically begins to execute the loader program.

Serial line speed is now switched to 9600 baud and the downloading program attempts to communicate with the loader program running in the microcontroller. If the loader program has been loaded successfully, synchronization between host computer and the Rug Warrior circuit board is achieved and the downloading program begins sending the actual pcode. As this happens more dots are printed to the screen. During the downloading process, the states (on or off) of the four LEDs below the MC68HC11 microcontroller have no special significance.

When downloading terminates, move the power switch all the way to the left, to the RUN position, then press the RESET button. The piezo buzzer will emit a brief "eeep" and the LCD screen will display the "Interactive C" wake-up message. You may need to adjust the LCD Pot (see Figure 1.1) to achieve the proper contrast.

2.5 Communicating with the Host

From a Macintosh computer, start the IC program by double clicking on the application "IC RW 2.xxx modem" or "IC RW 2.xxx printer" if you are using the printer port. (Here "xxx" is the three digit number of the IC version on your software disk.)

From a DOS computer, type `ic` to start IC running.

Your computer will connect with the Rug Warrior board and load the default library. You will now be able to execute C expressions from the keyboard and to load programs into your robot board. For example, if you type: `beep();` the piezo buzzer will emit a brief beep. Typing `2 + 2;` will produce a response similar to:

```
Downloaded 7 bytes (addresses C200-C206)
Returned <int> 4
```

2.6 Testing the Board

The first program you load should be `rw-test.lis`. At the IC prompt simply type: `load rw-test.lis`. When loading is complete, press RESET to select, in turn, the test for each subsystem . See Section 2.9 for a detailed explanation of what each test does. Note that initially only the LCD test, LED test, and buzzer test will produce meaningful results.

2.7 Rug Warrior Software

In addition to the IC programming environment, the software disk contains the following Rug Warrior specific code:

rw-test.lis Loader file that loads the self test functions.

selftest.c Routines for testing all of Rug Warrior's sensors and actuators.

cof.c Enables Rug Warrior to play a tune that tests the piezo buzzer.

shaft.lis Loader file for the shaft encoder utilities.

shaft.c User interface to the shaft encoders.

speed.asm Assembly file that implements the left shaft encoder click counter.

speed.icb Assembled version of `speed.asm`, loadable by IC (.asm files cannot be loaded).

regdefs.c Definitions for the MC68HC11's memory mapped registers.

lib_rw11.c Standard interface to Rug Warrior's sensors and actuators.

The software accompanying your robot board kit is supplied free of charge. You may copy and distribute this code as you choose, provided only that such distribution satisfies the restrictions described in `README.IC`. Software is provided as is; no warranty is stated or implied.

2.8 Assembly Language Routines

The current release of Rug Warrior software does not include the special MC68HC11 assembler (AS11) that simplifies construction of assembly language routines for inclusion in your code. This exclusion poses no problem for the majority of builders who will choose to program exclusively in IC. A description of the format of the processed assembly language files, .icb files, can be found in the IC Manual (Chapter 7). By deciphering this format, die-hard assembly programmers can create .icb files by hand.

A more convenient solution is available to builders who have access to the World Wide Web. Facilities provided by the creators of IC at the site `http://www.newtonlabs.com/ic` allow you to convert your assembly language code into .icb files. This site has additional information about IC including the availability of a supported, commercial version.

If you have FTP access to the Internet, you may want to check the site `cherupakha.media.mit.edu` periodically. The latest version of IC and related software for Rug Warrior and similar robot circuit boards can be found there.

2.9 Self Test

Rug Warrior is supplied with a set of routines that display the output of its sensors and activate its actuators. These routines can be used to verify the proper functioning of all subsystems as you construct and customize your robot.

When you load the self test routines, by typing `load rw-test.lis` to the IC prompt, a series of auxiliary files are automatically loaded.

These files include: `speed.icb`, `shaft.c`, `regdefs.c`, `cof.c`, and `selftest.c`.

Once this loading process completes, you can select the desired test by repeatedly pressing the RESET button. A particular test can also be selected by calling it directly from the keyboard. To do this, first type `kill_all` to stop the test currently running; then call the desired test from the list below. All tests run continuously but may be stopped by hitting carriage return.

`lcd_test()`

Displays, in turn, all characters defined by the LCD. Not all character codes are defined so some will show up as blank.

`led_test()`

Lights the four debugging LEDs. The 1s in the lowest 4 bits of the byte shown on the LCD indicate which of the LEDs should be on.

`piezo_test()`

Plays a tune on the piezo buzzer while displaying the frequency of each note.

`photo_test()`

Displays the light level sensed by the left and right photocells. Lower numbers indicate brighter light. The first line of the LCD shows the difference between left and right light levels; the arrow points toward the sensor exposed to the brighter light.

`bumper_test()`

Indicates which of the bumper switches are closed. Rug Warrior determines switch closure by decoding an analog value from a voltage adder. The voltage, mapped into the range 0 to 255 is displayed on the second line of the LCD.

`mic_test()`

Displays the instantaneous level of sound recorded by the microphone. This value is found by sampling the microphone as rapidly as possible. Sounds of very short duration, such as a hand clap, may be missed.

`ir_test()`

Alternately activates the left and right IR emitters and indicates whether a reflection has been detected. Note that the IR detector will function poorly if aimed directly at a fluorescent light.

`encoder_test()`

Displays the state of each encoder (high or low) using the two rightmost user LEDs. The LCD keeps a count of the encoder clicks (high to low transitions) detected. `encoder_test()` does *not* use the automatic velocity-checking feature from `speed.icb`.

`motor_test()`

Activates the left and right drive motors in a programmed sequence of velocities. (You must connect drive motors and motor power supply before the results of this test can be observed.) The number after the L or R in the display is, respectively, the left or right commanded velocity as a percentage of maximum velocity. The number before the letter is the actual velocity in units of encoder clicks per interval. For purposes of the test, the interval is 0.5 seconds.

`user_digital_test()`

Two digital input bits (PA1 and PA2) are available for user assignment. Their status is continuously displayed by `user_digital_test()`. Unless the lines (which are present on the expansion connector) are tied either high or low, the displayed value may fluctuate between 0 and 1.

`analog_test(6|7)`

Two analog lines (those associated with the microprocessor's PE6 and PE7 pins) are available for user input. The value of these or any of the six other analog lines may be displayed using `analog_test(n)`, where n is any number from 0 to 7.

`pyro_test(100,155)`

A pyroelectric sensor is an optional, user-supplied sensor. If you have installed a pyro sensor on Rug Warrior, `pyro_test` will display its state. The two arguments control the sensitivity and range of the thermometer display on the first line of the LCD. When the pyro is pointed toward an unchanging scene, a more or less constant value will be output. If a heat source (such as a person) moves across the pyro's field of view, the output of the pyro will first increase (or decrease) then decrease (or increase) as the heat edge passes. The sign of the change depends on the relative temperature of the source and background and on the orientation of the sensor.

2.10 Programming

See Chapter 7 for information on Interactive C. To create a robot program, use any editor capable of producing an ASCII text file. The editors Emacs, Microsoft Word, SimpleText, and a great many others will work. (You may have to experiment with the editor's file saving options in order to find a format compatible with IC.) After creating a program file, you can use the `load` command to load the program into Rug Warrior just as you did with the `rw-test.lis` program.

Before writing a program of your own, you may find it instructive to examine the self-test routines and the demo-one.c program. These files contain working examples of how to access sensors and run your robot.

Chapter 3

Electrical Assembly

3.1 Required Equipment

To complete the assembly of the Rug Warrior circuit board you will need the following tools and supplies:

- Soldering iron. The soldering iron should be a narrow-tipped, high quality instrument. It must deliver only the required heat as excessive heat can damage components.
- Small, sharp wire cutters.
- Needle-nose pliers.
- Wire stripper.
- Small gauge, rosin-core solder.

3.2 Optional Equipment

A multimeter capable of measuring voltage and resistance will enable you to identify short circuits and open circuits and to determine if proper voltage levels are present.

Along with being an invaluable aid in debugging high frequency circuits, an oscilloscope can provide important insights into the operation of the robot's sensors and other circuitry. Many wiring and other errors in the robot's circuits can be most easily diagnosed using an oscilloscope.

A circuit board holding/positioning vice (available from Radio Shack, Digi-Key, and elsewhere) makes circuit board assembly less awkward. If no such device is available it may be helpful to use masking tape to hold components in place while you turn the board over to apply solder.

3.3 Factory Pre-assembly

The Rug Warrior circuit board is partially assembled and tested at the factory. The microprocessor, memory, serial line interface, and certain other logic and auxiliary components are installed. This pre-assembly enables you to begin programming immediately and to use the microcontroller for debugging purposes as you build the rest of the circuitry.

If you have not already done so, please follow the self-test procedures described in the Chapter 2 *before* adding any other components to the board. Further, we recommend that the board be fully re-tested after each new system is installed. Doing so will allow you to more quickly isolate any errors you might make.

Finally, always make sure that battery power is disconnected before soldering new components to the board.

3.4 Building Rug Warrior

Compared to the circuit boards in most consumer electronics products, the Rug Warrior board is uncomplicated and straightforward. However, it will be necessary to pay very careful attention to detail when mounting parts and soldering components.

Before attempting Rug Warrior we recommend that the novice builder gain experience in the art of soldering and debugging with less demanding projects. One suggestion along this line: if you have previously constructed the TuteBot circuit on a bread board (as described in *Mobile Robots: Inspiration to Implementation*), try transferring the circuit to a PC board. Radio Shack supplies a PC board (part number 276-170) whose layout is identical to the bread board. Simply transfer the components and solder.

Most of Rug Warrior's integrated circuits use CMOS technology; CMOS chips can be damaged by electrostatic discharge. Manufacturers recommend that you work with a grounding strap attached to your wrist. Avoid handling the chips until you are ready to install

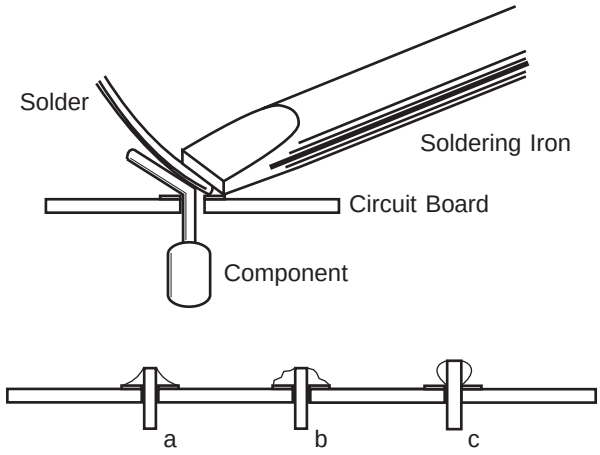

Figure 3.1: Soldering technique and results.

them and be careful not to grab a chip the first thing after walking across a carpeted floor—the spark that jumps from your finger to the chip may cause damage. Before touching a chip discharge yourself by touching something that is grounded.

After soldering, use wire cutters to clip the leads of components close to the bottom of the board.

Avoid putting the board down on anything conductive (wires, left-over clipped leads, solder, etc.). Even when the power is switched off some connections will still have voltage applied and may short out.

3.5 Soldering

The theory of soldering is simple: Heat the objects to be soldered, not just the solder itself. This action allows the solder to flow over, and securely bond, the parts to be joined. To actually achieve such happy results, however, requires a measure of skill.

Figure 3.1 is a side view of a component being soldered to a printed circuit board. Ideally, the tip of the soldering iron will touch the lead of the component, the pad to which the component will be attached, and the solder. To insure good heat transfer, keep the tip of your soldering iron properly tinned, coated with solder, at all times.

Parts *a*, *b*, and *c* of the figure show what can happen. In *a* a good

joint has been made, it appears smooth and shiny. *b* is an example of a "cold" solder joint. The surface appears dull and rough. Such joints often make intermittent contact. In *c*, heat was applied only to the component lead and not to the pad. The result is a joint that looks good upon casual inspection, but one that does not produce a reliable connection. Cases *b* and *c* can be corrected by reheating and, possibly, adding a little more solder.

Sometimes too much solder is applied to a joint or a component is soldered into the wrong hole. There are two helpful products that make it easy to remove the solder and try again. The first is solder wick, also known as desoldering braid. Solder wick is a copper braid that absorbs solder when heated. The second product, based on a spring-loaded piston, is the vacuum desoldering tool—popularly known as a "solder sucker." Both are available through Radio Shack, electronic hobbiest stores, and electronic mail order companies.

A helpful review of good soldering technique can be found in *The 6.270 Robot Builder's Guide* by Fred Martin. The guide is available online on the Internet from cherupakha.media.mit.edu, Internet address: 18.85.0.47.

3.6 Assembly Steps

This section guides you through the details of circuit board assembly. Please consult Figure 3.2 to identify components and their polarity. A number of tests will be recommended as construction proceeds.

3.6.1 Cables

All cables required by Rug Warrior can be made from the supplied 10-conductor cable and solder cup connectors. (See Figure 3.3.) As indicated, separate the cable into a group of four conductors (brown, red, orange, and yellow), a group of three conductors (green, blue, and violet), and a group of two conductors (white and black). Cut each group to the proper length as follows:

Shaft encoders: Two 4-inch lengths of 4-conductor cable.

Bump Switches: One 2.5-inch length, one 4.5-inch length, and one 5.5-inch length of the 3-conductor cable.

Motors: Two 7-inch lengths of the 2-conductor cable.

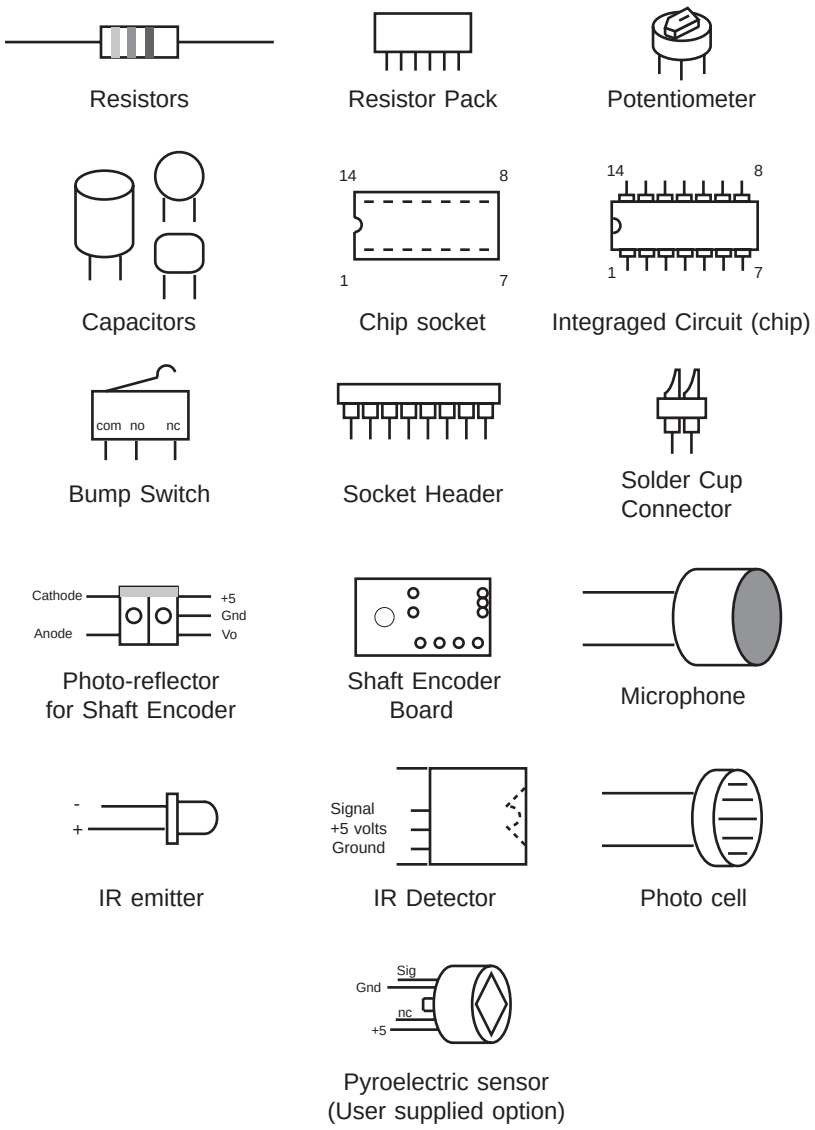

Figure 3.2: Component identification.

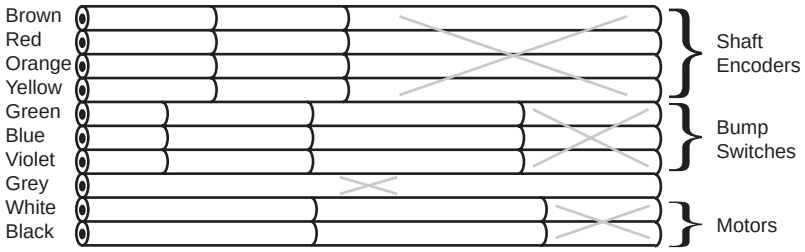

Figure 3.3: Separate the 10-conductor ribbon cable to make cables for shaft encoders, bump switches, and motors.

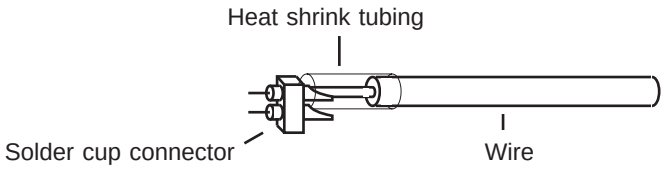

Figure 3.4: Detail of connector wiring. Strip insulation from the wire, tin with solder, solder the wire to the solder cup, and apply heat to heat shrink tubing to insulate the connection.

To make a good connection, first strip a bit of insulation from the end of a wire, say 1/4 inch in length. Then tin (coat with solder) the exposed wire. Next place a length of heat shrink tubing, sufficient to cover the connection, over the wire. Insert the tinned end into the solder cup end of a pin header and, using additional solder as required, solder in place. Next slide the heat shrink tubing over the connection and apply heat until the tubing shrinks around the connection. See Figure 3.4.

3.6.2 Microphone Circuit

Install and solder in place the following:

- ◯ 10 μF gain capacitor. Note the polarity of this capacitor. The positive terminal goes into the lower of the two holes, the one marked with a '+'.

○ 1000 μF capacitor. Note the capacitor's polarity. Since this component is large, it should be installed lying on its side on the board.

○ 2.2 K resistor (code: red, red, red) in the microphone circuit.

○ 0.001 μF disk capacitor in the microphone circuit. (The capacitor should be stamped with the number 102, among other numbers.)

○ Socket for the microphone (a 2-socket unit cut from the 32-socket terminal strip). Install it in the holes near the "MIC" label.

○ Socket for the LM386, the 8-pin IC socket.

Install:

○ The LM386 into its socket; pin 1 goes to the lower left.

○ The microphone into its socket. Note that the microphone is polarized. One pin of the microphone is connected to its case; this pin goes into the hole labeled GND on the board.

◇ Test the microphone. Install Rug Warrior's self-test code and run the microphone test, `mic_test`. In a quiet environment the thermometer display should show little or no activity. As you speak, whistle, or blow air on the microphone element the display should indicate a higher level of sound.

Construction tip: For certain applications, you might wish to reduce the motor noise picked up by the microphone by mounting the microphone remotely from the board. To do so use a length of coaxial cable. Connect the center conductor to the positive pin of the microphone and the other end to the left socket hole. Connect the outer braid to the ground on the board and to the pin that connects to the case on the microphone.

3.6.3 Sensors and Actuators

Solder in place the following:

○ Two motor connectors (2-socket terminal strips) at "L MOT" and "R MOT."

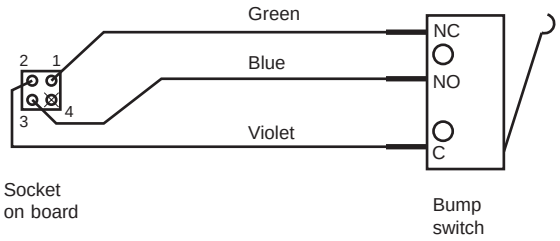

Figure 3.5: Schematic for bump switch cable.

○ L293D motor driver 16-pin socket (an SN754410 has been substituted for the L293D chip).

○ 74HC04 14-pin socket.

○ Sockets for the bump switches: six 2-socket units soldered in pairs. See Figure 1.1 for correct placement.

○ Two 1.2 K resistors (code: brown, red, red) for the bump circuit.

○ 2.2 K resistor (code: red, red, red) in bump circuit.

○ 47 K resistor pack (not polarized).

◇ Test the bump circuit. Connect SPDT switches to all connectors of the bump circuit. (See Figure 3.5.) Run `bumper_test()` described in Chapter 2 to verify that the bumpers have been wired correctly.

As an additional test of the bump circuit, you can measure the voltage at pin PE3 as you activate the switches. There should be evenly spaced, unique voltages for each switch and combination of switches. The possible combinations of switch closures and voltage at pin PE3 are:

Switch	Voltage
none	0.0
A	0.5
B	1.0
C	2.0
A & B	1.5
A & C	2.5
B & C	3.0
A & B & C	3.5

By monitoring the bump circuit voltage, the robot can determine collisions from six possible directions using only three switches.

Continue assembly by installing:

- ◯ Two shaft encoder sockets, using a total of four 2-socket terminal strips. Install sockets in the holes labeled "L SE" and "R SE" on the board.
- ◯ Two photocell sockets (2-socket terminal strip), at "L PH" and "R PH."
- ◯ Two IR emitter sockets (2-socket terminal strip), at "L IR" and "R IR."
- ◯ IR detector socket at "IR." Use pliers to bend the pins on a 3-socket header. Install it on the edge of the board as shown in Figure 3.6.

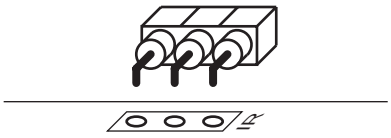

Figure 3.6: IR detector mounting detail. Use pliers to bend the pins at a right angle.

- ◯ Pyroelectric sensor socket (two 2-socket terminal strips) at label "PYRO." (This installation step is optional.)
- ◯ Two 680 Ω resistors (code: blue, gray, brown) for the shaft encoder circuit. (These resistors limit the current to the IR LEDs in the shaft encoder.)

○ Two 6.8 K resistors (code: blue, gray, red) for the shaft encoder circuit.

○ 5 K potentiometer for the IR oscillator at "5K POT."

○ 0.001 μF capacitor in the oscillator circuit. (The capacitor should be stamped with the number 102, among other numbers.)

○ 6.8 K resistor (code: blue, gray, red) in the oscillator circuit.

○ 100 K resistor (code: brown, black, yellow) in the oscillator circuit.

○ 50 K pot for LCD contrast adjustment at "CONTRAST."

○ Two 10 K resistors (code: brown, black, orange) for photocells.

○ Two 100 Ω resistors (code: brown, black, brown) for IR emitters.

○ The light sensors (photocells—not polarized) into their sockets; do not solder photocells. You may trim the leads so that the sensors mount close to the board.

◇ Run `photo_test` and observe that the arrows on the display point toward the photocell exposed to brighter light.

○ Install infrared LEDs (polarized). In both cases the negative terminal goes toward the center of the board. (The emitters will not be damaged if plugged in incorrectly.) You may trim the leads of the IR emitters so that they can be mounted closer to the board.

○ Solder a short piece of hookup wire directly from the ground terminal of the GP1U52X infrared detector to its case. This connection dramatically reduces the pickup of electrical noise by the detector.

○ Install the GP1U52X infrared detector in its socket (ground pin toward the bottom of the board). Bend the three mounting tabs on the detector's case out so that they do not contact the leads of other components already present on the board.

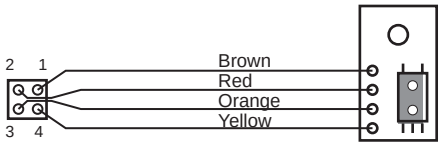

Figure 3.7: Wiring detail for shaft encoder board and cable. The photoreflector is mounted on top side of board; the bottom side has solder traces.

◯ Plug the xx74HC04x into its socket (labeled 74HC04). The 'x's in the prefix and suffix depend on the manufacturer of the chip. Pin 1 goes to the lower left.

◇ Test the IRs by running `ir_test`. Hold a thick white card in front of the IR emitters such that the radiation will reflect into the IR detector. Adjust the 40 kHz oscillator potentiometer until maximum range is achieved.

◯ Plug the motor driver chip, SN754410xx, into the socket labeled L293D. Pin 1 goes to the lower left.

Construction tip: If you are designing your own mechanical system, select motors that draw no more than 1 Amp when stalled. This amount is the maximum current that the motor driver chip can supply. The motor power supply can have a voltage in the range of 4.5 to 35 volts. The supply voltage should be 1 to 2 volts higher than the voltage rating of the motors because there is a voltage drop in the motor driver chip. See also the motors section of Chapter 6.

◯ Mount the photoreflectors on the shaft encoder boards. When soldering, use great care to avoid overheating and melting the photoreflector chip. Also, note that the mounting holes are very close together. After soldering, check carefully to see that no solder bridge has formed, shorting together adjacent mounting holes.

◯ The shaft encoders are mounted off board and thus require a cable. Wire the shaft encoder cables as indicated by the schematic, Figure 3.7. Install a 0.1 μF bypass capacitor (stamped with the number 104) in parallel with the power and ground lines on each shaft encoder connector as shown in Figure 3.8.

The motors and shaft encoders will be tested as a part of the final assembly in Chapter 4.

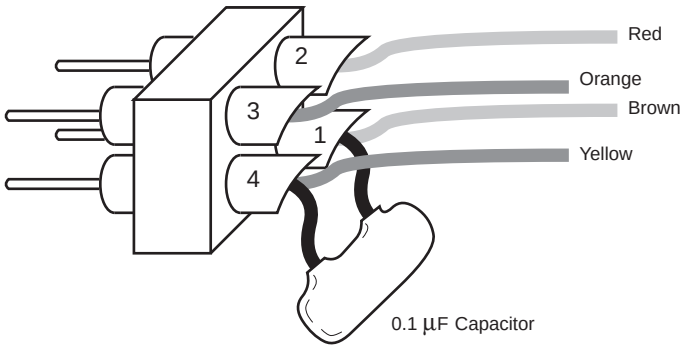

Figure 3.8: A 0.1 μF bypass capacitor is installed directly into the shaft encoder connectors.

3.6.4 Jumpers

If you are building the Expanded Kit it is not necessary to alter the factory jumper settings. You may, however, wish to change the default settings if you are designing your own base and power system or making certain other customizing changes.

Jumper J1 allows you to select either a single power supply or two separate power supplies for the logic and motor systems. The default setting requires you to connect a separate supply for the motors. This setup is recommended unless your motors are of the low noise, low current type. (Such motors are manufactured by Escap and Micro Mo, among others.) If you choose to use a single supply for both motor and logic power, you should cut the trace (on the bottom side of the board) connecting the center and top pin of the area labeled J1. Next, install a short wire on the top side between the center and bottom holes.

The setting of jumper J2 causes the MC68HC11 to run in *special test* mode. Special test mode was chosen because the LCD screen is operational only in this mode. If you wish to forego the LCD screen and operate in *expanded multiplexed* mode, cut the trace connecting the center and left holes of the J2 area and solder a wire between the center and right hole.

3.7 Summary

Rug Warrior's sensor circuitry is now complete—your circuit board should be fully functional. Please repeat the full set of tests in the `rw-test.lis` file to make sure that your assembly is free of errors.

If you discover a problem first consult Chapter 6 for trouble shooting suggestions. If the problem persists, you may want to contact the nearest Rug Warrior supporter on the enclosed list. Please remember that these individuals and groups have volunteered their time purely as a service to the amateur robotics community.

Chapter 4

Mechanical Assembly

This chapter describes the assembly of the mechanical components of the Brawn Kit and the integration of the Brains and Brawn Kits. If you have purchased the Brains Kit only, you may skip this chapter; however, you may find the examples instructive in designing your own robot. In addition to the equipment used for the electronic assembly you will need:

- Hobby/utility knife (e.g. an X-acto knife).
- Small slotted screwdriver.
- Small pliers.
- Scissors.

Please identify the following components of the Brawn Kit illustrated in Figures 4.1 and 4.2:

- 1 Chassis plate.
- 1 Drive motor alignment channel.
- 1 4x1 AA battery holder. (The Brains Kit has a separate battery holder.)
- 1 Skirt.
- 1 Caster ball.
- 1 Caster ball shaft.

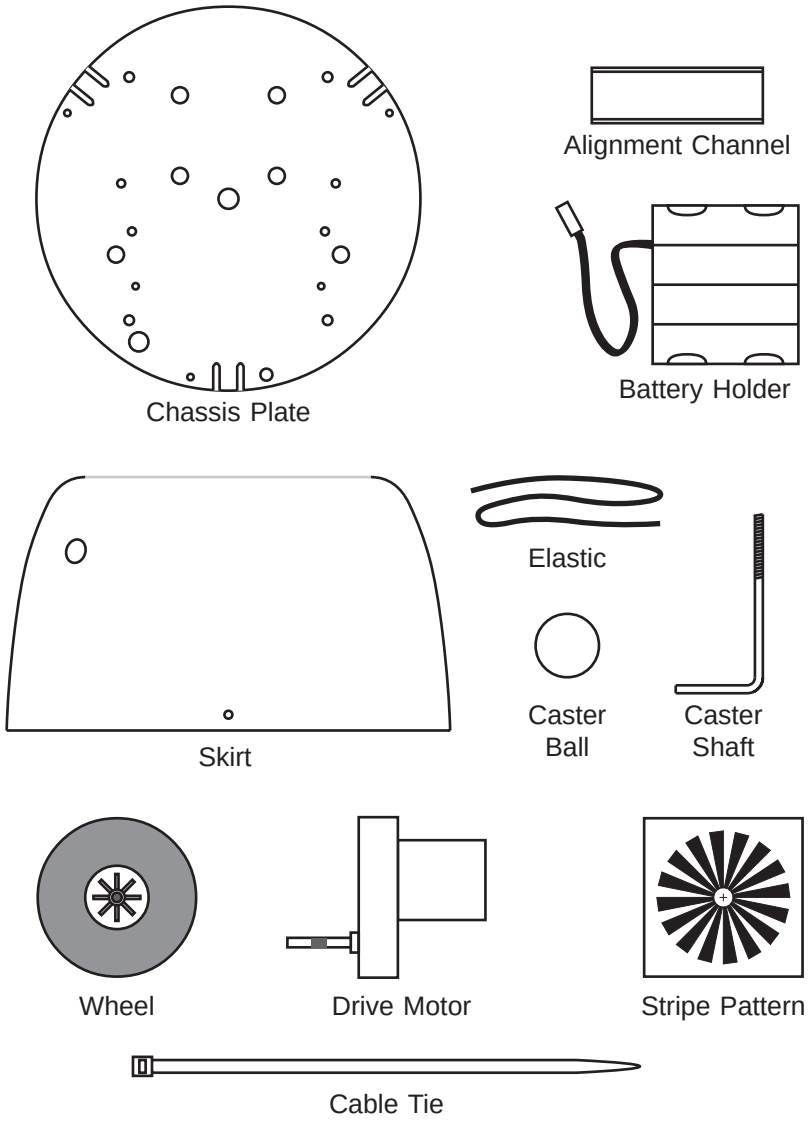

Figure 4.1: Brawn Kit mechanical components.

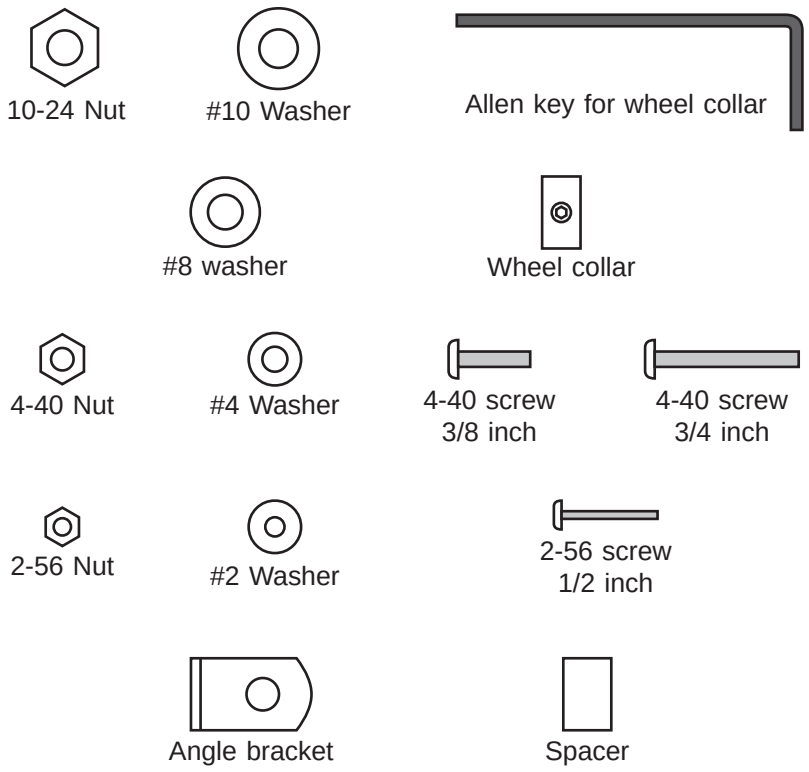

Figure 4.2: Brawn Kit mounting hardware.

- 1 Elastic strip.
- 2 Drive wheels.
- 2 Drive motors.
- 2 Adhesive-backed stripe patterns.
- 2 Cable ties.
- 2 10-24 nuts.
- 2 #10 washers.
- 1 Allen key for wheel collar.
- 2 #8 washers.
- 1 Wheel collar.
- 20 4-40 nuts.
- 22 #4 washers.
- 4 4-40, 3/4-inch screws.
- 16 4-40, 3/8-inch screws.
- 6 2-56 nuts.
- 6 #2 washers.
- 6 2-56 screws.
- 4 Motor/shaft encoder mounting brackets.
- 4 Nylon circuit board spacers.

4.1 Drive Motors

Use the black and white wires separated from the 10-conductor cable to make two 2-conductor cables for the motors.

○ Strip the insulation from both ends of the motor cable wires and slide short pieces of heat shrink tubing over each wire on each end leaving the stripped portion exposed. See Figure 4.3.

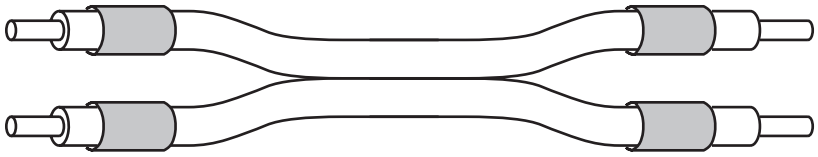

Figure 4.3: Prepare the motor cable as shown.

- ◯ Bend the two power connection tabs on the drive motor outward 90 degrees. Then solder the white wire of a motor cable to the + terminal and solder the black wire to the − terminal.
- ◯ Slide the heat shrink tubing down over the solder joints and apply heat to shrink the tubing in place. Repeat the steps above for the second motor/cable assembly.
- ◯ Solder the 2-pin solder cup headers to the free ends of the motor cables and secure the heat shrink tubing over each solder joint as was done for the motor terminals.

4.2 Drive Wheels

Striped patterns are installed on the inside of the drive wheels as a part of Rug Warrior's shaft encoder system. The output of a photoreflector pointed at the stripe patterns changes state as each stripe passes. The robot uses this information to determine how far each wheel has rotated. See Figure 4.4.

- ◯ With a utility knife roughen one lateral surface of each drive wheel. Hold the blade perpendicular to the wheel surface and scrape, being very careful not to cut the wheel.
- ◯ With scissors or a utility knife cut out the two patterns. Cut out the center of the pattern so that it is slightly smaller than the wheel hub (the hole should be about 0.9 inches in diameter). Cut slits in the pattern as shown. These slits will allow the patterns to conform to the contour of the wheel.
- ◯ Very carefully remove the pattern from the backing paper and apply the pattern to the roughened surface of the wheel. Press the inner portion of the pattern down so that it is captured by the hub.

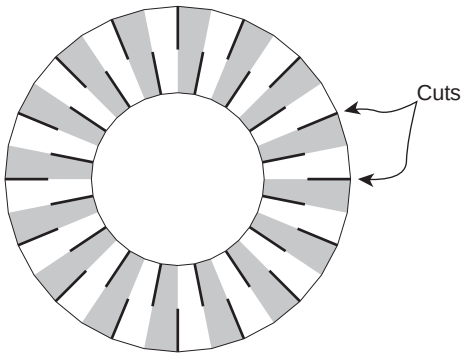

Figure 4.4: Stripe pattern cuts.

Construction tip: Sometimes, despite careful assembly, the outer part of the stripe pattern will pull away from the wheel. You can prevent this by laying down a thin bead of a cyanoacrylate gel adhesive or other strong adhesive along the outer edge of the pattern.

◯ Attach (but do not tighten) the motor mounting bracket to the motor as shown in Figure 4.5. Use a 4-40 screw, washer, and nut. The right motor assembly is the mirror image of the left assembly as shown in the figure.

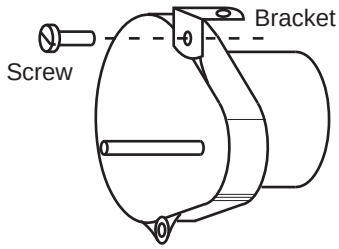

Figure 4.5: View, from the rear of the robot, of the left motor and motor mounting bracket.

◯ Place a #8 washer on the shaft of each motor. The washer maintains proper spacing.

◯ Mount the wheels on the pear-shaped gear assembly as follows. Hold the pear-shaped gear assembly flat against a table with the shaft pointing up. Let the motor (the cylindrical part) hang down over the edge of the table. Making sure that the stripe pattern faces downward, toward the gear assembly, press the wheel onto the shaft. This procedure insures that assembly forces are not transmitted to the gears inside the assembly.

Construction tip: If it ever becomes necessary to remove the wheel from the shaft, you should hold the wheel hub and press the shaft out rather than pull the wheel away from the gear assembly. This method will avoid damaging the gears.

4.3 Chassis

Be very careful to the orient the chassis plate as shown in Figure 4.6 when attaching components. The figure is a top view—motors and related components attach below the chassis plate; bump switches and the circuit board are attached above the plate.

◯ Place a piece of electrical tape across the back of each shaft encoder board. Put a 3/8-inch, 4-40 screw through the encoder board's mounting hole from the top. Place two #4 washers on the screw, then secure to the mounting bracket with a 4-40 nut. See Figure 4.7. The second assembly should be a mirror image of the first.

◯ Loosely attach the motor/bracket assemblies to the underside of the chassis plate using 3/8-inch-long 4-40 screws, 4-40 nuts, and #4 washers. See Figure 4.6 to identify the correct holes. Figure 4.7 shows motor and shaft encoder mounting details.

◯ Slide the motor alignment channel under the motors and press both motors into the channel to seat the motors.

◯ Push a cable tie through the motor attachment hole and wrap it around the motor and back up through the opposite attachment hole so that the motor is held against the aluminum channel. Tighten the cable tie and cut off the excess. Repeat the operation for the second motor.

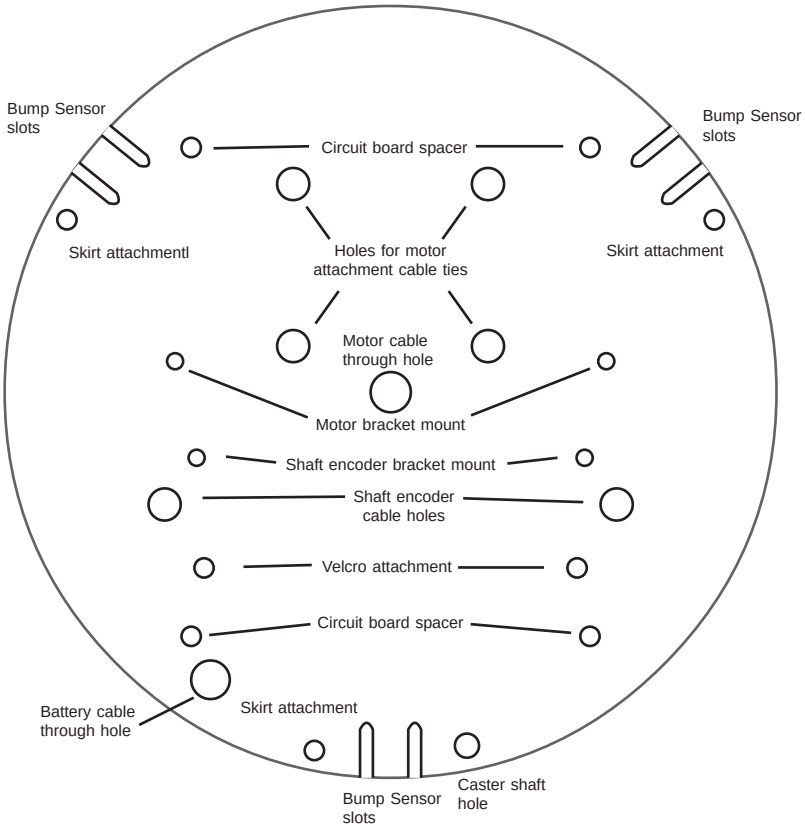

Figure 4.6: Function of the holes in the chassis plate (top view).

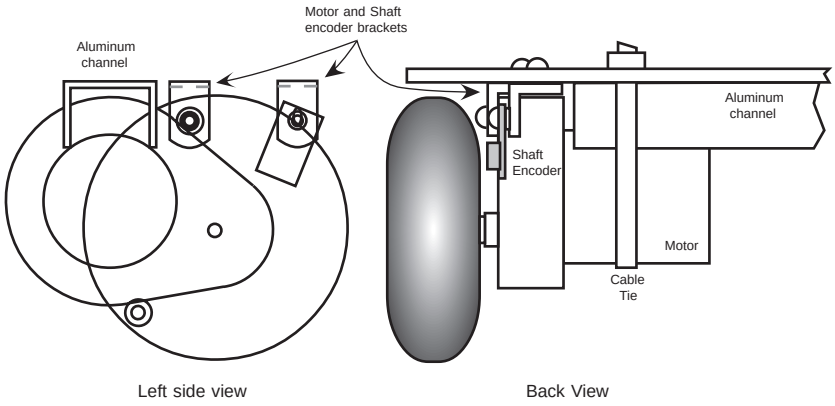

Figure 4.7: Detail of the motor/chassis attachment.

- ◯ Tighten the screw holding each motor to its motor mounting bracket. It will be necessary to compress the rubber wheel to get the blade of your screwdriver to the 4-40 screw.

- ◯ Tighten the screws holding the motor bracket assemblies to the chassis.

- ◯ Route the motor cables through the motor cable hole in the chassis.

- ◯ Mount each bracket/shaft encoder board assembly using a 4-40 screw, nut, and washer.

- ◯ Route the shaft encoder cables upward through the encoder cable holes in the chassis.

- ◯ Attach the three bump switches to the top of the chassis as shown in Figure 4.8, using the 2-56 screws and washers. Attach the bump switches only loosely as they will need to be adjusted when mounting the skirt.

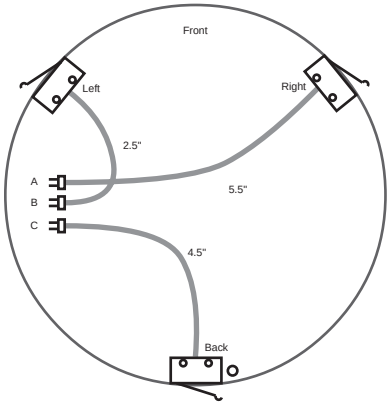

Figure 4.8: Position and orientation of bump sensors.

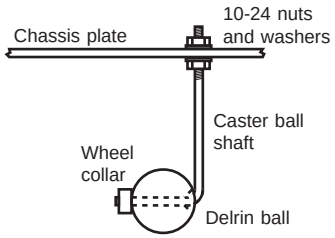

Figure 4.9: Caster wheel assembly.

4.4 Caster Wheel

◯ Place the 1-inch diameter delrin ball onto the caster shaft as shown in the drawing of Figure 4.9. Use the wheel collar to hold the ball in place. Tighten down the set screw in the wheel collar using the Allen key provided. Make sure that the ball can rotate freely on the shaft.

◯ Mount the caster wheel assembly on the chassis as shown. Use two each of the 8-32 nuts and washers. Adjust the nuts so that the chassis plate is level.

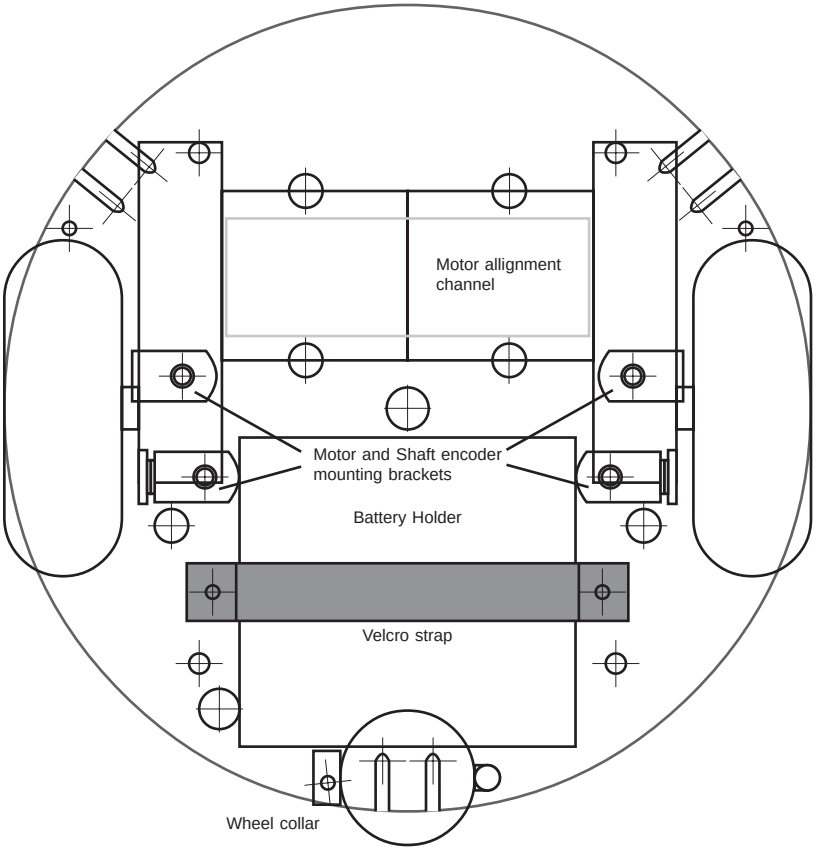

Figure 4.10: Vertical view of the assembly.

4.5 Battery Holders

○ Use a stylus or the blade of a utility knife to make a hole, centered about 1/4 inch from one end, in each piece of Velcro.

○ Attach the Velcro strips to the underside of the chassis with 4-40 screws, washers, and nuts. Orient the Velcro with one side up, one side down, so that the pieces can be stuck together to hold the battery holders in place. See Figure 4.10.

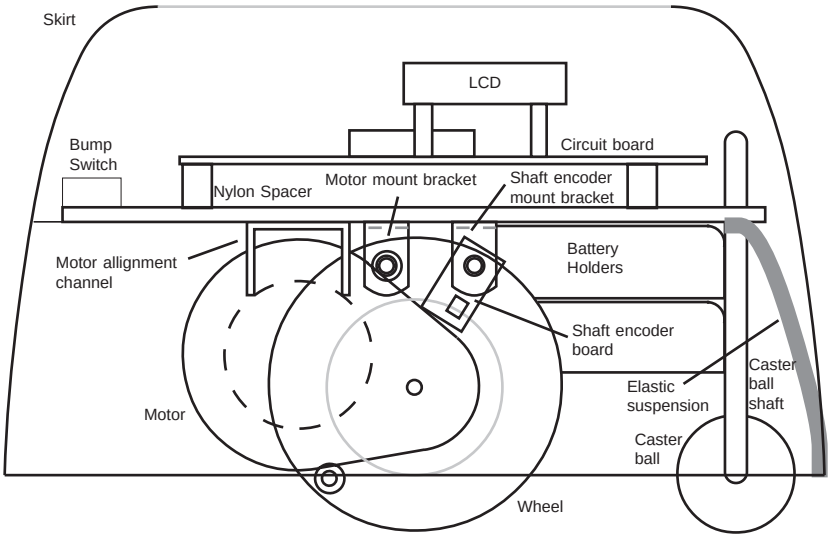

Figure 4.11: Side view of the mechanical assembly.

4.6 Skirt and Circuit Board

The force sensing skirt floats relative to the chassis, suspended on three lengths of elastic. The skirt is mounted such that collisions from any direction will cause one or more of the bump switches to be activated. Figure 4.11 shows how the elastic for the suspension is positioned.

◯ Cut the elastic into sections approximately 2.5 inches long.

◯ Use a stylus or pin to make a hole 1/4 inch from each end of each piece of elastic. Attach one end of each piece of elastic to the skirt attachment holes on the chassis. Use 3/8-inch-long 4-40 screws, 4-40 nuts, and #4 washers. The elastic should extend radially from the center.

◯ Attach the circuit board to the top of the chassis using four each of the nylon spaces, 3/4-inch-long 4-40 screws, 4-40 nuts, and #4 washers. Connect the bump cables, shaft encoder cables, and motor cables to their correct sockets on the board. Connect the motor cables so that, looking from the back, the white wire is on the left.

○ With the assembly resting wheels down on a table, slide the skirt over the top. Position the skirt so that the two larger holes (through which the IR emitters will point) are toward the front.

○ Attach the skirt to the elastic strips using the three mounting holes near the bottom of the skirt. The skirt should hang about 1/2 inch above the floor.

○ Carefully adjust the placement of the bump switches and the elastic/skirt attachment so that when the skirt is undisturbed none of the bump switches are depressed. Make sure that each switch is activated when the skirt is pressed from the proper direction. Try to avoid any blind spots, regions where the skirt can be pressed without activating any switches. When all is optimally adjusted, tighten the bump switch mounting screws.

○ Stack the two battery holders and attach them to the bottom of the chassis using the Velcro strips. Bring the cables up through the battery cable hole in the chassis. Plug the cables into the logic and motor supply connectors.

○ Position the battery holders as far back as possible so that the robot will balance properly. (If your robot has a tendency to pitch forward when rapidly reversing direction, it may be necessary to add weight at the rear.)

4.7 Final Adjustment

○ Hold or suspend the robot so that the wheels do not contact any surface. Type a command to move the motors forward, e.g. `drive(100,0);`. Both wheels should turn so as to make the robot move forward. If this is not the case switch the polarity of the cable (plug it in the other way) going to the motor turning in the wrong direction.

◇ Load the file `rw-test.lis` and run the `motor_test` routine. Verify that the robot operates as expected. During the initial motor break-in phase you may notice that the robot pulls to the left or right when it has been commanded to go straight. After 15 to 30 minutes of motor operation this condition should correct itself.

◇ Load up and run the shaft encoder test from the self-test routines supplied with the robot. As you move the wheels by hand, rotate and adjust each shaft encoder board until a reliable signal is obtained. Make sure that the shaft encoder can reliably detect every dark to light stripe transition (and that no false transitions are detected). Unless this procedure is done, distance measurements taken by the robot will be unreliable.

◯ Push heat shrink tubing, cut to approximately 1/2 inch, over each IR emitter and then carefully aim the emitters through the holes in the skirt.

◇ Test the IR obstacle detection system using the `ir_test` routine. If the emitters are not positioned properly, the plastic skirt can cause internal reflections that the detector will interpret as an obstacle. It may be necessary to cover the back of each emitter with electrical tape to prevent IR leakage; the detector is quite sensitive. If you need to block one emitter during the test, use black electrical tape rather covering the emitter with your thumb—your thumb and fingers are translucent to the IR radiation.

Rug Warrior is now complete. As a first exercise you may wish to run the demo-one.c program on the software disk. Demo-one has three modes of operation selected by pressing the RESET button (the same way that rw-test.lis selects different test routines). The LCD screen displays the current mode.

The first mode, "Seek light," causes the robot to move toward the brightest light. While it does so, Warrior monitors the IR obstacle detector and turns away if a collision seems imminent. The robot also watches the bump sensors. If a collision does occur, Rug Warrior will back up and attempt to escape from the obstacle. You will be able to make Rug Warrior follow you by pointing a flashlight at its photocells.

The second mode of operation, "Seek darkness," causes the robot to avoid the light. Rug Warrior monitors the IR detector and bump sensors as above while trying to find the darkest spot to hide.

In the third operating mode, "Wait for whistle," Rug Warrior remains still while monitoring the microphone circuit. When the robot detects a loud noise it responds by sounding the piezo buzzer.

You may use the ideas and example code in demo-one as a springboard for creating your own robot application programs.

Chapter 5

Expanding Rug Warrior

Rug Warrior has built-in connections for four additional sensor inputs and the signal to drive one more actuator is present on the board. With a small bit of extra circuitry, Rug Warrior can accomodate any number of new inputs or outputs.

5.1 Built-in Ports

Two digital inputs, lines PA1 and PA2, and two analog inputs, lines PE6 and PE7, are available on the expansion connector for user assignment. One digital output port, PA4, is present on the expansion connector as well. This port, however, is also used by the LCD screen and is available for user assignment only if the LCD software driver is disabled. (This deactivation may be accomplished by operating the microcontroller in expanded mode rather than special test mode.) The analog input, PE5, nominally assigned to the pyroelectric sensor, is also available if you have not installed a pyro.

Line PA3, which drives the piezo buzzer, can be shared to drive an RC servo or other actuator. To use PA3 you must install a socket (or solder a wire) to the board as shown in Figure 5.1.

Besides the lines mentioned, the expansion connector also supplies the non-multiplexed address lines of the MC68HC11's bus. The LCD connector brings out the multiplexed data/address lines. See Figure 5.2. Using the address lines, data lines, and other control sig-

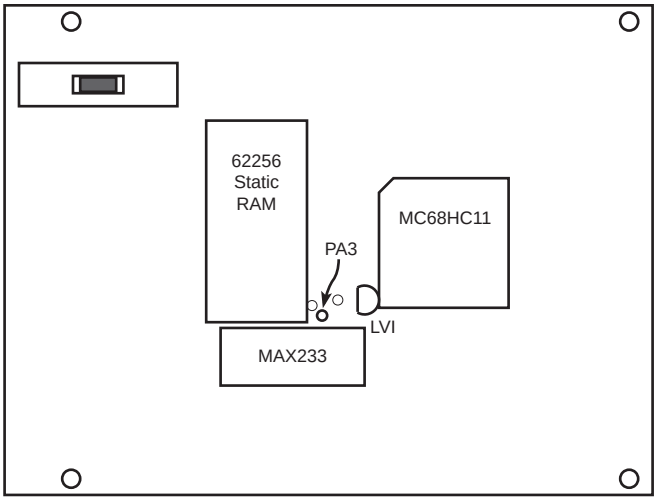

Figure 5.1: Line PA3 is available on the board at the point indicated.

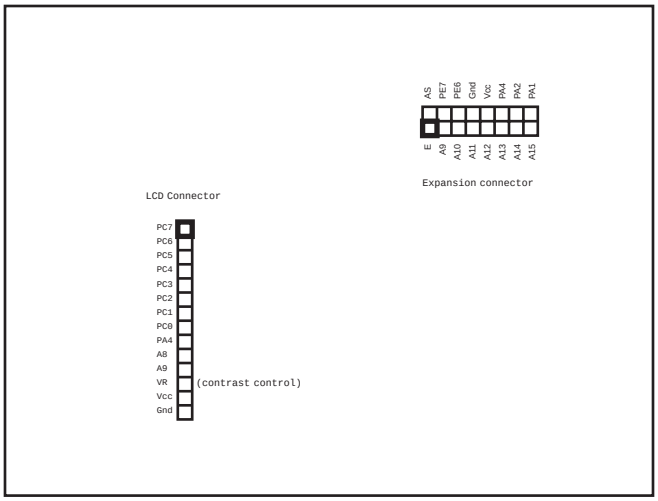

Figure 5.2: LCD connector and expansion connector.

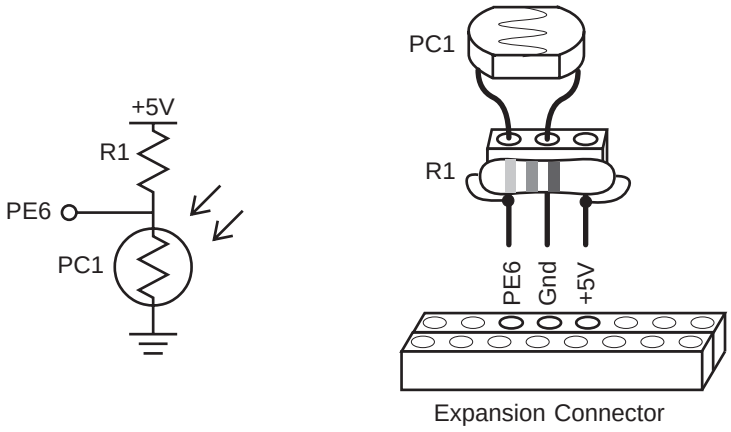

Figure 5.3: A photocell circuit built on a 3-socket header plugs directly into the expansion connector.

nals it is possible to construct any number of memory mapped I/O ports. To make full use of Rug Warrior's memory mapped expansion capabilities, the sophisticated builder should acquire a set of reference manuals for the Motorola MC68HC11A1. Contact the Motorola Literature Distribution office at (800) 544-9497. Outside the United States contact the nearest Motorola sales office or representative.

5.2 Expansion Example

Suppose we would like for Rug Warrior to determine when it has run under a table or chair. (Perhaps we're writing a program that has the robot hide in a dark place until it detects a loud sound, then rush out and surprise whomever has entered the room.) We can make this scheme possible by installing an upward-pointing photocell on Rug Warrior.

Normally, we would build expansion circuitry on a piece of protoboard (circuit board material with predrilled holes and possibly copper pads). However, a photocell circuit is so simple that it can be constructed entirely on a 3-socket header. See Figure 5.3.

The schematic diagram on the left of Figure 5.3 shows how the circuit works. A voltage divider is formed by resistor R1 and photocell PC1. We choose R1 such that its resistance equals the resistance of

PC1 when PC1 is exposed to an "average" level of illumination. If you build this circuit using the same type of photocell supplied with the Brains Kit, then R1 should be about 10 KΩs.

The right side of Figure 5.3 shows how the circuit is constructed. Resistor R1 is soldered to the first and third pins of the 3-socket header, PC1 is plugged into first and second pins of the header, and the header is plugged into the PE6, Gnd, and +5 V sockets of the expansion connector. In software we can access the new sensor using the function `analog(6)`.

Rug Warrior will support a great many other sorts of sensors and actuators. For a detailed example showing how to install an RC servo motor and sonar range finder on Rug Warrior, please consult "RugNav: a Scanning Sonar You Can Build," Parts 1 and 2. These articles appeared in the fall 1995 and winter 1996 editions of *The Robotics Practitioner*. If unavailable at your local library, contact *The Robotics Practitioner* via email at trp@footfalls.com, or mail at Footfalls, Ltd., 483 S. Kirkwood Road, Suite 130, Kirkwook, MO 63122.

Chapter 6

Trouble Shooting

This section contains some suggestions for solving problems sometimes encountered by robot builders.

Board doesn't work at all.

If not even the power-on (BAT ON) LED will light, check that the batteries have sufficient voltage (at least 5 volts). Also, check that the connector is plugged in correctly and that neither it nor the connecting wires have become damaged. If you have added components to the board, check that power and ground have not inadvertently become connected. Unplug the battery connector from the board and measure the resistance between power and ground. If the resistance is 0 or only a few ohms, inspect the board very carefully for solder bridges or other accidental power-to-ground connections.

If, while testing the board, you have unplugged and reinstalled any of the socketed integrated circuits, make sure that they have been inserted in the correct direction. Pin 1, marked by a dot or semicircle, (see Figure 1.1) should be on the left for all socketed chips, except the 28-pin RAM chip which has pin 1 on top. Chips can be damaged if installed in the reverse direction and may require replacement if this has occurred. Chips can also be damaged by an electric discharge.

Elevated temperature is one indication of a damaged chip. Only the motor driver chip (SN754410) should be warm to the touch under normal operation.

Cannot download pcode.

When attempting to download pcode (initialize the board) either the system times out or an error message is displayed stating that a char-

acter received is different from the one sent. Make sure that the download switch is in the DL (download) rather than RUN position. The battery voltage may be too low or there may be a fault in the connector. Make sure that the power-on LED is lighted and that the BAT LOW LED is not lighted. Check the voltage on the board. It should be no lower than about 5 volts.

The serial line in your computer may be configured or connected incorrectly. Check any system parameters in your computer that could affect the serial line; e.g., serial output could be directed to an internal modem. Check that the serial cable is plugged into the proper connector—modem vs. printer, or the correct com port.

The serial cable may have been miswired or a poor connection may exist. Miswiring may result if you are using a modular cable other than the one supplied in the kit. Recheck all connections and cables between the serial output of your computer and Rug Warrior's phone jack. Frequently, the transmit and receive lines are inadvertently switched when builders construct their own cable (this causes no damage). If this is the case, simply swap the lines and try downloading again.

If the problem persists you should do the following to isolate the error. Pull the phone plug out of the phone jack on the Rug Warrior board. Hold a thin wire so as to connect together the outermost contacts of the 4-wire plug, the transmit and receive lines. (This procedured may require an extra set of hands.) While doing so run the program to initialize the board. By connecting together the transmit and receive wires you have created a loopback—this guarantees that the initialize board program will receive the same characters it transmits.

If the initialize board program successfully completes loading the bootstrap loader (no timeout and no error message) it indicates that your serial line and cable are configured correctly and the problem may be in the Rug Warrior board.

If the initialize board program cannot load the bootstrap loader it means that there is a flaw in the serial cable, the connection, or the configuration of your serial line. In any case you may wish to contact a member of the Rug Warrior Experts Group for advice on how to proceed.

Board appears to initialize correctly but IC will not run.

After initializing the board move the download switch to the RUN position and press the RESET button. If you do not hear an "eeep"

from the piezo buzzer initialization was unsuccessful. Recheck the procedure and attempt initialization again. Also, make sure that the pcode code has not been corrupted. If this is a possibility, replace the file `pcoderwl.s19` on your hard disk with the copy from the software distribution disk.

Behavior of board is "flaky."

The two most common causes of odd behavior, e.g., board crashes and restarts, are 1) low battery voltage, and 2) voltage transients caused by the motors. If the LOW BAT LED flashes, it indicates that the low voltage inhibit chip has been activated, which halts, then restarts the processor. Check that the logic batteries are fresh. If the batteries are good, make sure that any newly added sensor or actuator circuitry is connected correctly and does not draw more current than the batteries are rated to supply.

If the board behaves strangely only when the motors operate, it indicates the presence of voltage transients. This condition should not occur with the motors in the Brawn Kit but is a possibility if you have designed your own mobility system. You should switch to separate power supplies for logic and motors. If supplies are already separate it may be that the motors produce excessive electrical noise (see below).

Processor crashes when the motors run.

If the noisy motors of your custom designed mobility system prevent your board from running properly, there are a few tricks to try before abandoning your motors in favor of higher quality (more expensive) ones. First, connect a disk capacitor (the value can be a fraction of a microfarad, say $0.1\,\mu$F) across the leads of each motor, at the motor. This action can reduce voltage spikes produced by the motor's brushes. Another trick is to attach a large **non-polarized** electrolytic capacitor across the motor; the larger the capacitance the better but try $10.0\,\mu$F or so to begin. (Don't use the more common polarized electrolytic capacitor for this purpose as it will be destroyed by the reverse voltage when you run the motors backward.) The non-polarized capacitors can reduce longer time scale voltage dips generated when the motor starts up or reverses direction.

Adding resistors of a few ohms in series with each lead of both motors can also help reduce voltage transients. This procedure is guaranteed to work for sufficiently large resistances but motor performance will degrade.

Finally, batteries capable of supplying a large surge current may help. Try using NiCd or lead-acid batteries in place of alkaline cells in the logic and/or motor supply.

Sensor does not pass self test.

Check the solder connections on board. Solder bridges (accidental connections between adjacent traces or pads) and poorly made solder joints can prevent circuits from working properly.

IR system does not work or range is too short.

Make sure that the IR emitters are correctly installed in their sockets. (No damage results if emitters are backward.) Make sure the 40 kHz oscillator is tuned correctly by adjusting the potentiometer (labeled 5 K POT on the board). Check for bad solder joints on the 78HC04 chip socket and in the 40 kHz oscillator.

Shaft encoders do not work.

The IR emitter-detector component of the shaft encoder system, the photoreflector, must be positioned so that it points at the radial black and white stripe pattern attached to the wheels. Performance is highly sensitive to the distance between the stripe pattern and the photoreflector. Ideally, separation should be about 3 millimeters.

The photoreflector output changes state as each stripe passes. If the output fails to change, it may be that there is insufficient contrast between reflective and non-reflective stripes. If the pattern was crinkled during installation, try smoothing it. If the black printed stripes have been worn off, try redrawing them with a marker.

I'm confused about how to plug in the LCD screen.

The LCD screen fits over the board as shown in Figure 1.1. Note that pin 14 on the LCD board corresponds to the pin labeled PC7 (with a bold square) on Rug Warrior's LCD connector.

LCD screen doesn't work.

When IC begins running, it prints a message to the LCD screen. If this message fails to appear, there are several possible causes. First, make sure that IC is in fact running. Move the power switch to the RUN position and press the RESET button. You should hear an "eeep" from the piezo buzzer and you should be able to communicate with the board. For example, if you type: $1 + 1$, you should see a response similar to:

```
Downloading 7 bytes (addresses C200-C206):   7 loaded
Returned <int> 2
```

on your computer screen. If you do not hear the "eeep" on reset or are unable to communicate with the board, then IC may not be running. Refer to the appropriate trouble shooting section.

If the LCD screen remains blank even though IC is running, try adjusting the LCD Pot (see Figure 1.1). Typically, at one extreme setting the screen will be blank; at the other, one line of the display will be dark. (This behavior should be observed even with the board in download mode.)

Make sure the LCD is correctly plugged into the 14-pin raised connector on the PC board. If the screen remains blank, check the voltage on the VR pin of the LCD connector (see Figure 4.1). Using the LCD Pot, you should be able to adjust the voltage on the VR pin from 0 to about 5 or 6 volts.

Getting help.

Whatever problem bedevils your robot, the likelihood is that someone has encountered it before. Don't be stymied when help is available. Tap into the wealth of experience and know-how at your fingertips by contacting a member of the Rug Warrior Experts Group.

Chapter 7

Interactive C Manual

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

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

- **Interpreted execution** that allows run-time error checking and prevents crashing. For example, IC does array bounds checking at run-time to protect against programming errors.
- **Ease of design.** Writing a compiler for a stack machine is significantly easier than writing one for a typical processor. Since IC's pcode is machine-independent, porting IC to another processor entails rewriting the pcode interpreter, rather than changing the compiler.
- **Small object code.** Stack machine code tends to be smaller than a native code representation.

- **multitasking.** Because the pseudocode is fully stack-based, a process' 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 pcode is interpreted, these advantages are taken at the expense of raw execution speed. Still, IC is no slouch.

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

7.1 Using IC

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 closed curly brace "`}`". The following example creates a local variable `i` and prints the sum `i+7` to the 6811's LCD screen:

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

7.1.1 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 command 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 redownloads remaining files.

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

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

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

- **Edit a file.** The command `edit` <*filename*> brings up a system editor to allow editing of a file. This command is most useful on single-tasking operating systems, like MS-DOS.

- **Run an inferior shell.** If IC is running on a MS-DOS system, this command opens a shell to execute MS-DOS functions.

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

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

7.1.2 Line Editing

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

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

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

Figure 7.1 shows the keystroke mappings understood by IC. IC does parenthesis-balance-highlighting as expressions are typed.

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

Figure 7.1: IC Command-line keystroke mappings.

7.1.3 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.

7.2 A Quick C Tutorial

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

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

All functions must have a return value; that is, a value that they return when they finish execution. **`main`** has a return value type of **`void`**, which is the "null" type. Other types include integers (**`int`**) and floating point numbers (**`float`**). This *function declaration* information must precede each function definition.

Immediately following the function declaration is the function's name (in this case, **`main`**). Next, in parentheses, are any arguments (or

inputs) to the function. `main` has none, but a empty set of parentheses is still required.

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

Next comes a series of C *statements*. Statements demand that some action be taken. Our demonstration program has a single statement, a `printf` (formatted print) which will print the message "`Hello, world!`" to the LCD display. The `\n` indicates the end of the 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 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 of incompatibility by *coercing* the integer sum `(square(a) + square(b))` into a float by preceding it with the desired type, in parentheses. Thus, the integer sum is made into a floating point number and passed along to `sqrt`.

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

This concludes the brief C tutorial.

7.3 Data Types, Operations, and Expressions

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

7.3.1 Variable Names

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

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

7.3.2 Data Types

IC supports the following data types:

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

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

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

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

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

7.3.3 Local and Global Variables

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

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

Variable Initialization

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

```
int foo()
{
int x;        /* create local var x with initial value 0 */
int y= 7;     /* create local var y with initial value 7 */
...
}
float z=3.0;  /* create global var z with initial value 3.0 */
```

Local variables are initialized whenever the function containing them runs.

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

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

Persistent Global Variables

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

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

```
persistent int i;
```

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

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

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

Persistent variables were created with two applications in mind:

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

7.3.4 Constants

Integers

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

Long Integers

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

Floating Point Numbers

Floating point numbers may use exponential notation (e.g., "`10e3`" or "`10E3`") or must contain the decimal period. For example, the floating point zero can be given as "`0.`", "`0.0`", or "`0E1`", but not as just "`0`".

Characters and Character Strings

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

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

7.3.5 Operators

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

Integers

The following operations are supported on integers:

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

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

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

Long Integers

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

Floating Point Numbers

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

The following operations are supported on floating point numbers:

- **Arithmetic.** addition `+`, subtraction `-`, multiplication `*`, division `/`.
- **Comparison.** greater-than `>`, less-than `<`, equality `==`, greater-than-equal `>=`, less-than-equal `<=`.

- **Built-in Math Functions.** A set of trigonometric, logarithmic, and exponential functions is supported, as discussed in Section 7.8 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).

7.3.6 Assignment Operators and Expressions

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

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

The abbreviated form

```
a += 2;
```

could also be used to perform the same operation.

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

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

7.3.7 Increment and Decrement Operators

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

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

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

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

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

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

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

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

7.3.8 Precedence and Order of Evaluation

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

Operator	Associativity
`() []`	left to right
`! ~ ++ -- -` (*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

7.4 Control Flow

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

7.4.1 Statements and Blocks

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

There is never a semicolon after a right brace that ends a block.

7.4.2 If-Else

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

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

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

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

7.4.3 While

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

`while (` *expression* `)`
  statement

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

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

`while (1)`
  statement

7.4.4 For

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

`for (` *expr-1* `;` *expr-2* `;` *expr-3* `)`
  statement

This is equivalent to the following construct using `while`:

expr-1 `;`
`while (` *expr-2* `) {`

statement
expr-3 ;
}

Typically, *expr-1* is an assignment, *expr-2* is a relational expression, and *expr-3* is an increment or decrement of some manner. For example, the following code counts from 0 to 99, printing each number along the way:

```
int i;
for (i= 0; i < 100; i++)
printf("%d\n", i);
```

7.4.5 Break

Use of the `break` provides an early exit from a `while` or a `for` loop.

7.5 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 procedure is best illustrated by some examples.

7.5.1 Printing Examples

Example 1: Printing a message. The following statement prints a text string to the screen.

```
printf("Hello, world!\n");
```

In this example, the format string is simply printed to the screen.

The character "`\n`" at the end of the string signifies *end-of-line*. When an end-of-line character is printed, the LCD screen will be cleared when a subsequent character is printed. Thus, most `printf` statements are terminated by a `\n`.

Example 2: Printing a number. The following statement prints the value of the integer variable `x` with a brief message.

```
printf("Value is %d\n", x);
```

The special form `%d` is used to format the printing of an integer in decimal format.

Example 3: Printing a number in binary. The following statement prints the value of the integer variable `x` as a binary number.

```
printf("Value is %b\n", x);
```

The special form `%b` is used to format the printing of an integer in binary format. Only the *low byte* of the number is printed.

Example 4: Printing a floating point number. The following statement prints the value of the floating point variable `n` as a floating point number.

```
printf("Value is %f\n", n);
```

The special form `%f` is used to format the printing of floating point number.

Example 5: Printing two numbers in hexadecimal format.

```
printf("A=%x  B=%x\n", a, b);
```

The form `%x` formats an integer to print in hexadecimal.

7.5.2 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)

7.5.3 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 halted.
- Characters that would be printed beyond the final character position are truncated.
- When using a two-line display, the `printf()` command treats the display as a single longer line.
- Printing of long integers is not presently supported.

7.6 Arrays and Pointers

IC supports one-dimensional arrays of characters, integers, long integers, and floating point numbers. Pointers to data items and arrays are supported.

7.6.1 Declaring and Initializing Arrays

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

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

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

Arrays are initialized by default to contain all zero values; arrays may also be initialized at declaration by specifying the array elements, separated by commas, within curly braces. Using this syntax, the size of the array would not be specified within the square braces; it is determined by the number of elements given in the declaration. For example,

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

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

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

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

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

7.6.2 Passing Arrays as Arguments

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

In normal C, there are two ways of declaring an array argument: as an array or as a pointer. IC only allows declaring array arguments as arrays.

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

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

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

When passing an array variable to a function, use of the square brackets is not needed:

```
{
int array[10];
retrieve_element(3, array);
}
```

7.6.3 Declaring Pointer Variables

Pointers can be passed to functions which then go on to modify the value of the variable being pointed to. This situation is useful because the same function can be called to modify different variables, just by giving it a different pointer.

Pointers are declared with the use of the asterisk (`*`). In the example

```
int *foo;
float *bar;
```

`foo` is declared as a pointer to an integer, and `bar` is declared as a pointer to a floating point number.

To make a pointer variable point at some other variable, the ampersand operator is used. The ampersand operator returns the *address* of a variable's value; that is, the place in memory where the variable's value is stored. Thus:

```
int *foo;
int x= 5;
foo= &x;
```

makes the pointer `foo` "point at" the value of `x` (which happens to be 5).

This pointer can now be used to retrieve the value of `x` using the asterisk operator. This process is called *de-referencing*. The pointer, or reference to a value, is used to fetch the value being pointed at. Thus:

```
int y;
y= *foo;
```

sets `y` equal to the value pointed at by `foo`. In the previous example, `foo` was set to point at `x`, which had the value 5. Thus, the result of de-referencing `foo` yields 5, and `y` will be set to 5.

7.6.4 Passing Pointers as Arguments

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

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

In the code, the function argument is specified as a pointer using the asterisk:

```
void average_sensor(int port, int *result)
{
int sum= 0;
int i;
for (i= 0; i< 10; i++) sum += analog(port);
*result=  sum/10;
}
```

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

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

7.7 Multitasking

7.7.1 Overview

One of the most powerful features of IC is its multitasking 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 monitored in a *process table*; each time through the table, each process runs once (for an amount of time equal to its number of ticks).

Each process has its own *program stack.* The stack is used to pass arguments for function calls, store local variables, and store return addresses from function calls. The size of this stack is defined at the time a process is created. The default size of a process stack is 256 bytes.

Processes that make extensive use of recursion or use large local arrays will probably require a stack size larger than the default. Each function call requires two stack bytes (for the return address) plus the number of argument bytes; if the function that is called creates local variables, then they also use up stack space. In addition, C expressions create intermediate values that are stored on the stack.

It is up to the programmer to determine if a particular process requires a stack size larger than the default. A process may also be created with a stack size *smaller* than the default, in order to save stack memory space, if it is known that the process will not require the full default amount.

When a process is created, it is assigned a unique *process identification number* or *pid.* This number can be used to kill a process.

7.7.2 Creating New Processes

The function `start_process` creates a new process. `start_process` takes one mandatory argument—the function call to be started as a process. There are two optional arguments: the process' 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 outcome 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 millisecond per invocation and has a stack size of 50 bytes (for the given definition of `check_sensor`, a small stack space would be sufficient).

7.7.3 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 1 second later:

```
void main()
{
int pid;
pid= start_process(check_sensor(2));
sleep(1.0);
kill_process(pid);
}
```

7.7.4 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 the name of the function that is currently executing.

7.7.5 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 it 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, and is useful if a process knows that it will not need to do any work until the next time around the scheduler loop. `defer()` is implemented as a C built-in function.

7.8 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; the result is in radians.

`float cos(float angle)`
Returns cosine of `angle`. Angle is specified in radians; the result is in radians.

`float tan(float angle)`
Returns tangent of `angle`. Angle is specified in radians; the result is in radians.

`float atan(float angle)`
Returns arc tangent of `angle`. Angle is specified in radians; the result is in radians.

`float sqrt(float num)`
Returns square root of `num`.

`float log10(float num)`
Returns logarithm of `num` to the base 10.

`float log(float num)`
Returns natural logarithm of `num`.

`float exp10(float num)`
Returns 10 to the `num` power.

`float exp(float num)`
Returns e to the `num` power.

`(float) a ^ (float) b`
Returns `a` to the `b` power.

7.9 Memory Access Functions

IC has primitives for directly examining and modifying memory contents. These primitives should be used with care as it would be easy to corrupt memory and crash the system using these functions.

There should be little need to use these functions. Most interaction with system memory should be done with arrays and/or globals.

`int peek(int loc)`
Returns the byte located at address `loc`.

`int peekword(int loc)`
Returns the 16-bit value located at address `loc` and `loc`+1. `loc` has the most significant byte, as per the 6811 16-bit addressing standard.

`void poke(int loc, int byte)`
Stores the 8-bit value `byte` at memory address `loc`.

`void pokeword(int loc, int word)`
Stores the 16-bit value `word` at memory addresses `loc` and `loc`+1.

`void bit_set(int loc, int mask)`
Sets bits that are set in `mask` at memory address `loc`.

`void bit_clear(int loc, int mask)`
Clears bits that are set in `mask` at memory address `loc`.

7.10 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.

7.10.1 Compile-Time Errors

When compiler errors occur, an error message is printed to the screen. All compile-time errors must be fixed before a file can be downloaded to the board.

7.10.2 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

7.11 Binary Programs

With the use of a customized 6811 assembler program, IC allows the use of machine language programs within the C environment. There are two ways that machine language programs may be incorporated:

1. Programs may be called from C as if they were C functions.
2. Programs may install themselves into the interrupt structure of the 6811, running repetitiously or when invoked due to a hardware or software interrupt.

When operating as a function, the interface between C and a binary program is limited: a binary program must be given one integer as an argument, and will return an integer as its return value. However, programs in a binary file can declare any number of global integer variables in the C environment. Also, the binary program can use its argument as a pointer to a C data structure.

7.11.1 The Binary Source File

Special keywords in the source assembly language file (or module) are used to establish the following features of the binary program:

Entry point. The entry point for calls to each program defined in the binary file.

Initialization entry point. Each file may have one routine that is called automatically upon a reset condition. (The reset conditions are explained in Section 7.3.3, which discusses global variable initialization.) This initialization routine is particularly useful for programs which will function as interrupt routines.

C variable definitions. Any number of 2-byte C integer variables may be declared within a binary file. When the module is loaded into IC, these variables become defined as globals in C.

To explain how these features work, let's look at a sample IC binary source program, listed in Figure 7.2.

```
/* Sample icb file */
/* origin for module and variables */
ORG     MAIN_START
/* program to return twice the argument passed to us*/
subroutine_double:
ASLD
RTS
/* declaration for the variable "foo" */
variable_foo:
FDB     55
/* program to set the C variable "foo" */
subroutine_set_foo:
STD     variable_foo
RTS
/* program to retrieve the variable "foo" */
subroutine_get_foo:
LDD     variable_foo
RTS
/* code that runs on reset conditions */
subroutine_initialize_module:
LDD     #69
STD     variable_foo
RTS
```

Figure 7.2: Sample IC binary source file: `testicb.asm`.

The first statement of the file ("`ORG MAIN_START`") declares the start of the binary programs. This line must precede the code itself.

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 program 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 two; 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 statement is an assembler directive that creates a 2-byte value (which is the initial value of the variable).

Variables used by binary programs must be declared in the binary file. These variables then become C globals when the binary file is loaded into C.

The next binary program in the file is named "`set_foo`." It performs the action of setting the value of the variable `foo`, which is defined later in the file. It sets the value 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 action overwrites the 55 which would otherwise be the default value for that variable.

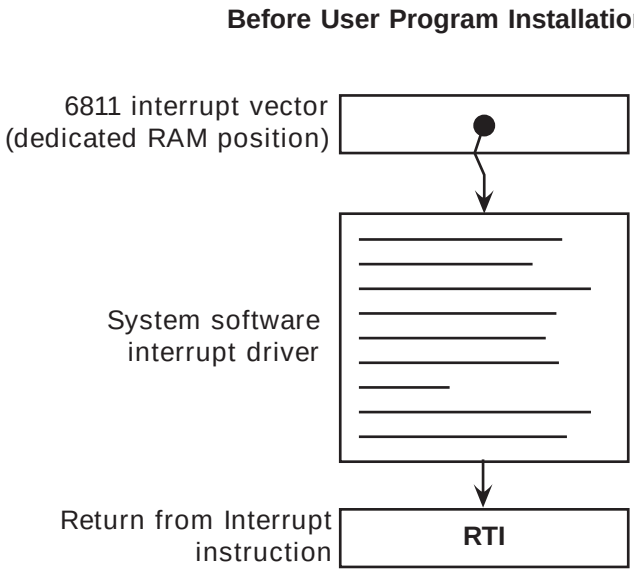

Figure 7.3: Interrupt structure before user program installation.

Initialization of global 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.

7.11.2 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 system 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 result is accomplished by having the user program "intercept" the original 6811 interrupt vector that points to system interrupt code. This vector is made to point at the user program. When the user program finishes, it jumps to the start of the system interrupt code.

Figure 7.3 depicts the interrupt structure before user program installation. The 6811 vector location points to system software code, which terminates in a "return from interrupt" instruction.

Figure 7.4 illustrates the result after the user program is installed. The 6811 vector points to the user program, which exits by jumping to the system software driver. This driver exits as before, with the RTI instruction.

Multiple-user programs could be installed in this fashion. Each would install itself ahead of the previous one.

Figure 7.5 shows an example program that installs itself into the System Interrupt. This program toggles the signal line controlling the piezo beeper every time it is run; since the System Interrupt runs at 1000 Hz, this program will create a continuous tone of 500 Hz.

The first line after the comment header includes a file named "`6811regs.asm`". This file contains equates for all 6811 registers and interrupt vectors; most binary programs will need at least a few of these equates. It is simplest to keep them all in one file that can be easily included.

The `subroutine_initialize_module` declaration begins the initialization portion of the program. The file "`ldxibase.asm`" is then included. This file contains a few lines of 6811 assembler code that perform the function of determining the base pointer to the 6811 interrupt vector area, and loading this pointer into the 6811 X register.

The following four lines of code install the interrupt program (beginning with the label `interrupt_code_start`) according to the method that was illustrated in Figure 7.4.

First, the existing interrupt pointer is fetched. As indicated by the comment, the 6811's TOC4 timer is used to implement the System Interrupt. The vector is poked into the JMP instruction that will conclude the interrupt code itself.

Next, the 6811 interrupt pointer is replaced with a pointer to the new code. These two steps complete the initialization sequence.

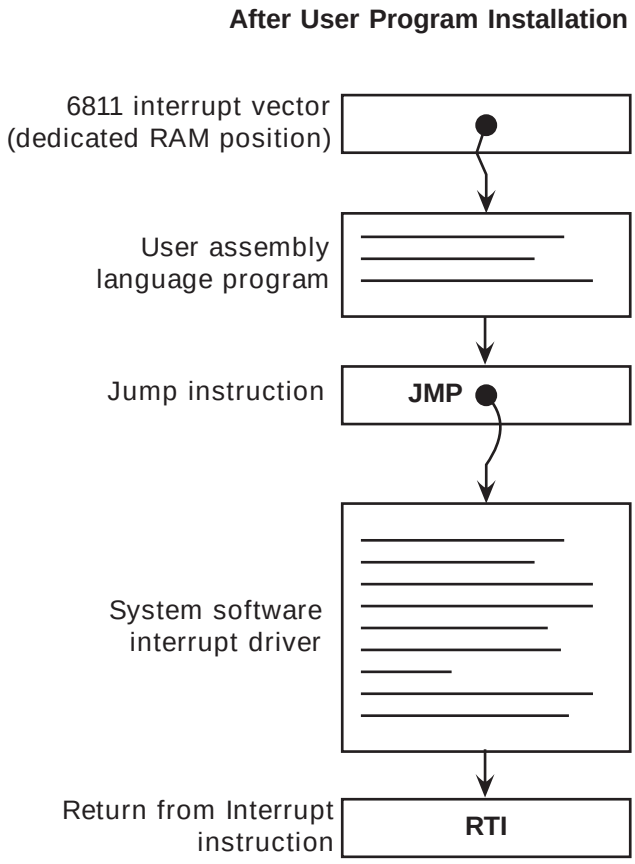

Figure 7.4: Interrupt structure after user program installation.

```
* 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 7.5: `sysibeep.asm`: Binary program that installs into System Interrupt.

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.

7.11.3 The Binary Object File

The source file for a binary program must be named with the `.asm` suffix. Once the `.asm` file is created, a special version of the 6811 assembler program is used to construct the binary object code. This program creates a file containing the assembled machine code plus label definitions of entry points and C variables.

```
S116802005390037FD802239FC802239CC0045FD8022393C
S9030000FC
S116872B05390037FD872D39FC872D39CC0045FD872D39F4
S9030000FC
6811 assembler version 2.1  10-Aug-91
   please send bugs to Randy Sargent (rsargent@athena.mit.edu)
   original program by Motorola.
subroutine_double 872b *0007
subroutine_get_foo 8733 *0021
subroutine_initialize_module 8737  *0026
subroutine_set_foo 872f  *0016
variable_foo 872d *0012 0017 0022 0028
```

Figure 7.6: Sample IC binary object file: `testicb.icb`.

The program `as11_ic` is used to assemble the source code and create a binary object file. It is given the file name of the source file as an argument. The resulting object file is automatically given the suffix `.icb` (for IC Binary). Figure 7.6 shows the binary object file that is created from the `testicb.asm` example file.

7.11.4 Loading an `icb` File

Once the `.icb` file is created, it can be loaded into IC just like any other C file. If there are C functions that are to be used in conjunction with the binary programs, it is customary to put them into a file with the same name as the `.icb` file, and then use a `.lis` file to load the two files together.

7.11.5 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 2-byte length value followed by the array contents.

7.12 IC File Formats and Management

This section explains how IC deals with multiple source files.

7.12.1 C Programs

All files containing C code must be named with the "`.c`" suffix.

Loading functions from more than one C file can be done by issuing commands at the IC prompt to load each of the files. For example, to load the C files named `foo.c` and `bar.c`:

```
C> load foo.c
C> load bar.c
```

Alternatively, the files could be loaded with a single command:

```
C> load foo.c bar.c
```

If the files to be loaded contain dependencies (for example, if one file has a function that references a variable or function defined in the other file), then the second method (multiple file names to one load command) or the following approach must be used.

7.12.2 List Files

If the program is separated into multiple files that are always loaded together, a "list file" may be created. This file tells IC to load a set of named files. Continuing the previous example, a file called `gnu.lis` can be created:

Listing of `gnu.lis`:

```
foo.c
bar.c
```

Then typing the command `load gnu.lis` from the C prompt would cause both `foo.c` and `bar.c` to be loaded.

7.12.3 File and Function Management

Unloading Files

When files are loaded into IC, they stay loaded until they are explicitly unloaded, which 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 reloaded 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 redownload `foo.c` and any other files that are presently loaded.

The solution is to first unload the file containing the `main` that is not desired, and then load the file that contains the new `main`:

```
C> unload foo.c
C> load bar.c
```

7.13 Configuring IC

IC has a multitude of command-line switches that allow control of a number of things. Explanations for these switches can be obtained by issuing the command "`ic -help`."

IC stores the search path for and name of the library files internally; these may be changed by executing the command "`ic -config`."

When this command is run, IC will prompt for a new path and library file name, and will create a new executable copy of itself with these changes.

Appendix

A.1 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 sensor values.

IC automatically loads the library file every time it is invoked. IC may be configured to load different library files as its default.

Functions in this appendix apply to software distribution disk 2.0 and later.

A.1.1 Time Commands

System code keeps track of time passage in milliseconds. The time variables are implemented using the long integer data type. Standard functions allow use of floating point variables when using the timing functions.

`void reset_system_time()`
Resets the count of system time to 0 milliseconds.

`long mseconds()`
Returns the count of system time in milliseconds. Time count is reset by hardware reset (i.e., pressing the 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 1 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. For 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);
```

A.1.2 Tone Functions

Several commands control the production of tones using the piezo beeper.

`void beep()`
Produces a tone of 500 Hertz for a period of 0.3 seconds.

`void tone(float frequency, float length)`
Produces a tone at pitch `frequency` Hertz for `length` seconds. Both `frequency` and `length` are floats.

`void set_beeper_pitch(float frequency)`
Sets the beeper tone to be `frequency` Hz. The `beeper_on()` function is then used to turn on the beeper. Change beeper pitch while the beeper is on to produce warbling tones.

`void beeper_on()`
Turns on the beeper at the frequency `set_beeper_pitch` last selected.

`void beeper_off()`
Turns off the beeper.

A.1.3 Sensor Input

`int digital(int nth)`
Returns the state of the nth input bit of digital I/O Port A. Bits 1 and 2 are unassigned inputs. Bits 0 and 7 are inputs connected to the left and right shaft encoders, respectively. Bits 3, 4, 5, and 6 are

outputs. For example, calling `digital(0)` should return the same value as `left_shaft()`.

`int analog(int p)`
Returns the value of sensor port numbered `p`. The result is an integer between 0 and 255.

Analog ports are mapped to the 6811's Port E pins. Analog(0) corresponds to PE0, analog(1) to PE1, and so on. Lines PE6 and PE7 are available on the expansion connector for user customization. If the optional pyro sensor has not been installed, line PE5 (available on the pyro connector) may also be used.

`int analog(photo_right | photo_left)`
Returns the value of the right or left photo cell, respectively. Lower numbers indicate brighter light.

`int analog(microphone)`
Returns the instantaneous value of the A/D input connected to the microphone. When no sound is present `analog(microphone)` returns a value of approximately 128. When sound is detected by the microphone the value may be more or less than 128. To determine a meaningful value for sound level the microphone must be sampled frequently and averaged.

`int analog(pyro)`
Returns the value of the optional pyro sensor. This value is more or less constant with time unless a heat edge passes the pyro sensor.

`int bumper()`
Returns a 3-bit value corresponding to the closed bumper switches. If the bump switches are labeled A, B, and C, then bit 0 set corresponds to A closed, bit 1 set corresponds to switch B closed, and bit 2 set corresponds to switch C closed. For example, if `bumper()` returns '3' it means that switches A and B are closed.

`int ir_detect()`
Returns a 2-bit value with the following meanings: 0 – no obstacles detected, 1 – obstacle detected on right side, 2 – obstacle detected on left side, and 3 – obstacle detected on both sides. This test requires at least 2 milliseconds to complete.

`int leds(int n)`
Set the debugging LEDs to the binary value of the four least

significant bits of `n`. If the LEDs are labeled from 0 to 3 starting on the right of the board then bit 0 of n corresponds to LED 0, bit 1 to LED 1, and so on. Note that the debugging LEDs are connected in parallel with the motor direction bits and the IR emitters. Thus the LEDs cannot be activated without also activating these other functions and vice versa.

A.1.4 Motor Functions

Rug Warrior's drive motors are velocity controlled. This feature means that it is possible to specify the speed at which each motor is to run, rather than just whether the motor is on or off. The primitives described below provide only open loop control. However, using these primitives it is possible to construct a closed loop velocity control system.

`int init_motors()`

This function initializes several registers that allow velocity control of the motors. It is called automatically on reset but may need to be called again if the MC68HC11 registers OC1M, TCTL1, TOC1, or DDRD are altered.

`void motor(int index, int speed)`

This function is a primitive for controlling motor velocity. Index = 0 accesses the left motor; index = 1 accesses the right motor. Speed is an integer number between -100 and +100. Speed represents the percentage of full speed at which the motor operates. *Before software distribution 2.0* `speed` *was implemented as a float number.*

`void drive(float trans_vel, float rot_vel)`

This function is used to control robot velocity and direction. `Trans_vel` is an integer number between -100 and 100 specifying robot velocity as a percentage of maximum velocity (negative numbers cause the robot to back up). `rot_vel` is a number between -100 and 100 specifying the robot's rotational velocity as a percentage of maximum velocity. Positive numbers correspond to counterclockwise rotation. Drive specifies only the open loop velocities. *Before software distribution 2.0* `trans_vel` *and* `rot_vel` *were implemented as float numbers.*

A.1.5 Shaft Encoders

Unlike other library routines, most functions that monitor the shaft encoders are not loaded automatically. These functions must be included explicitly by loading `shaft.lis`. This file loads `speed.icb` and `shaft.c`. The former file enables an interrupt routine that monitors the shaft encoder connected to line PA0. The latter file provides a low level user interface in the form of functions `get_left_clicks()` and `get_right_clicks()`. Each time one of these routines is called it returns the number of shaft encoder clicks since the last time it was called. This number is proportional to the distance the wheel has turned during that interval.

To measure robot velocity, arrange to have the `get_clicks` functions called at regular intervals. For example, this arrangement could be accomplished by setting up a process that calls a `get_clicks` function, writes the value to a variable, and then sleeps for a certain amount of time. How frequently the process should call `get_clicks` depends on the number of black to white stripe transitions (16 for the Brawn Kit) on your wheel and the velocity resolution that you require. Note that the shaft encoders ignore the direction of rotation—the `get_clicks` functions count up whether the wheel is turning clockwise or counterclockwise.

`void init_velocity()`

Initializes the routines that enable velocity monitoring. This function must be called before the results returned by `get_left_clicks()`, and `get_right_clicks()` become meaningful.

`int get_left_clicks()`

Accesses a variable maintained by a lower level interrupt routine (defined in `speed.asm`). The interrupt routine is associated with 6811 pin PA0. The value returned is the number of clicks since the previous call to `get_left_clicks`.

`int get_right_clicks()`

Accesses the pulse counter register associated with 6811 pin PA7. The value returned is the number of clicks since the previous call to `get_right_clicks`.

For programming convenience the following two functions *are* present in the Rug Warrior library.

`left_shaft()`

Returns the current state of the left shaft encoder; 1 if a white

stripe is sensed, 0 otherwise.

`right_shaft()`
Returns the current state of the right shaft encoder; 1 if a white stripe is sensed, 0 otherwise.

A.2 Serial Connection

Figure A.7 is provided for informational purposes. The figure will be helpful if you choose to construct your own cable to make a direct connection to Rug Warrior from either a Macintosh DIN-8 connector or an IBM PC 9-pin connector.

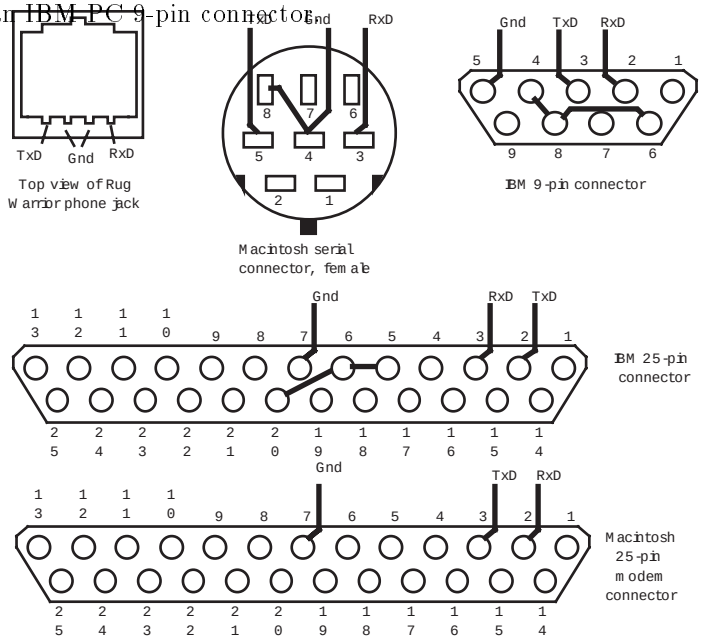

Figure A.7: The Rug Warrior connector requires only three signals, Ground, Transmit, and Receive (Gnd, TxD, and RxD respectively).

The drawing shows several common host computer connectors indicating the salient pins. Wire your cable in such a way that Gnd, TxD, and RxD on your host are connected to the respective signals on the Rug Warrior phone jack. Depending on your serial card, it may be necessary to wire together pins 5, 6, and 20 of the IBM 25-

pin connector or pins 4, 6, and 8 of the IBM 9-pin connector. Pins 4 and 8 of the Macintosh serial connector must be connected.

A.3 Selected Rug Warrior Specifications

The following are typical specifications for Rug Warrior robots constructed from the Expanded Kit.

Component	Typical	Min	Max	Units
Logic supply	5.0	4.6	7.0	Volts
Motor supply	5.0	4.0	9.0	Volts
Microprocessor clock freq.	2.0			MHz
Serial line speed	9600			baud
IR osc. freq.	40	38	42	KHz
Obstacle detection range	15			inches
Robot speed	0.67			ft/sec
Encoder clicks per rotation	16			Units
Robot weight (no batteries)	27			Oz
Motor current (each motor)			1.0	Amp
Wheel diameter	2.5			inches
Robot diameter	7.3			inches
Robot height	4.75			inches

IR obstacle detection range assumes a flat white object and optimal oscillator tuning.