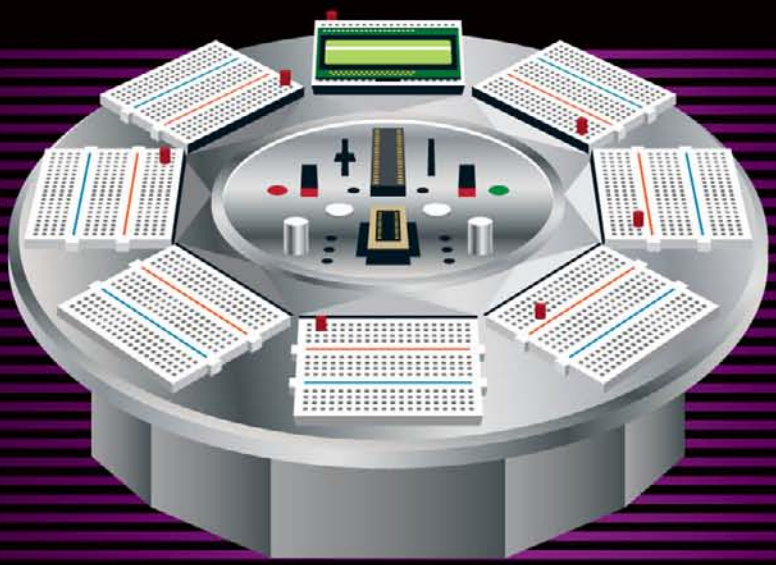
PICAXE® MICROCONTROLLER PROJECTS FOR THE EVIL GENIUS™



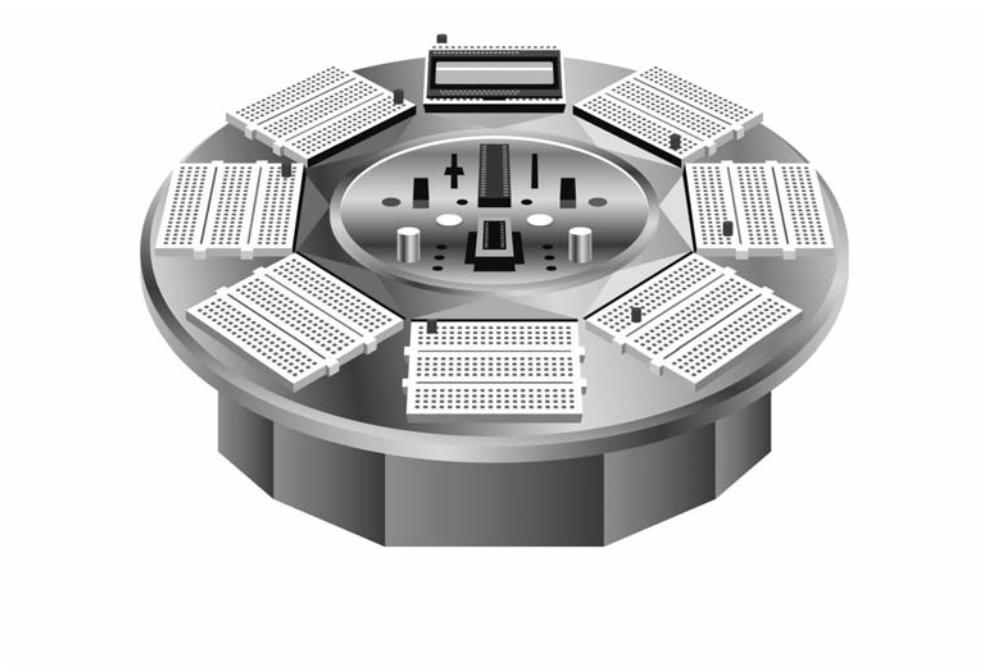
- Shows you how to program and build your own PICAXE microcontroller devices
- Filled with step-by-step instructions, illustrations, photos, and diagrams
- Projects lead up to a full-fledged robotics prototyping platform

TAB™ ELECTRONICS

RON HACKETT

Foreword by Clive Seager, Technical Director, Revolution Education Ltd.

PICAXE® Microcontroller Projects for the Evil Genius™



Evil Genius™ Series

Bike, Scooter, and Chopper Projects for the Evil Genius
Bionics for the Evil Genius: 25 Build-It-Yourself Projects
Electronic Circuits for the Evil Genius, Second Edition: 64 Lessons with Projects
Electronic Gadgets for the Evil Genius: 28 Build-It-Yourself Projects
Electronic Sensors for the Evil Genius: 54 Electrifying Projects
50 Awesome Auto Projects for the Evil Genius
50 Green Projects for the Evil Genius
50 Model Rocket Projects for the Evil Genius
51 High-Tech Practical Jokes for the Evil Genius
46 Science Fair Projects for the Evil Genius
Fuel Cell Projects for the Evil Genius
Holography Projects for the Evil Genius
Mechatronics for the Evil Genius: 25 Build-It-Yourself Projects
Mind Performance Projects for the Evil Genius: 19 Brain-Bending Bio Hacks
MORE Electronic Gadgets for the Evil Genius: 40 NEW Build-It-Yourself Projects
101 Outer Space Projects for the Evil Genius
101 Spy Gadgets for the Evil Genius
125 Physics Projects for the Evil Genius
123 PIC® Microcontroller Experiments for the Evil Genius
123 Robotics Experiments for the Evil Genius
PC Mods for the Evil Genius: 25 Custom Builds to Turbocharge Your Computer
PICAXE Microcontroller Projects for the Evil Genius
Programming Video Games for the Evil Genius
Recycling Projects for the Evil Genius
Solar Energy Projects for the Evil Genius
Telephone Projects for the Evil Genius
30 Arduino Projects for the Evil Genius
25 Home Automation Projects for the Evil Genius
22 Radio and Receiver Projects for the Evil Genius

PICAXE® Microcontroller Projects for the Evil Genius™

Ron Hackett

New York Chicago San Francisco Lisbon London Madrid
Mexico City Milan New Delhi San Juan Seoul
Singapore Sydney Toronto

ISBN: 978-0-07-170327-7

MHID: 0-07-170327-6

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-170326-0,
MHID: 0-07-170326-8.

This book is dedicated to my three beautiful grandchildren: Sasha, James, and Dima.

About the Author

Ron Hackett has had more than 30 years experience in the fields of education and psychology. He has taught mathematics, psychology, and computer science courses on both the high school and college levels, and in-service courses for teachers in the use of microcomputers in the classroom setting. Ron has published numerous PICAXE-related articles for *Nuts and Volts* and *SERVO* magazines. He also designed the "Brain-Alpha" PC board used in the popular SERVO TankBot robot. To assist his readers in obtaining PICAXE information and parts, Ron has also established a website (www.JRHackett.net) dedicated to "spreading the word" about the PICAXE line of microcontrollers.

When he isn't ensconced in his basement workspace, writing, or tinkering with Octavius, Ron enjoys grilling and barbecuing in the outdoor kitchen he built in his backyard. Currently, he's working on a PICAXE-based temperature controller for his ceramic smoker.

Contents at a Glance

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

I'VE JUST PUT DOWN THE TELEPHONE after a call from a lovely gentleman, who, at the age of 81, has decided to start electronics as a hobby. He phones every so often and always ends the conversation with a jovial "Thanks for your help. Got to get this project finished soon, as I don't know how long I've got left!" I hope it's a long time!

When we launched the PICAXE system over ten years ago, it was designed as a method of allowing schoolchildren to use all the power of Microchip PICs within their school projects without any of the technical difficulties of complicated hardware or complex programming languages. In the intervening years, the PICAXE system has been adopted by hundreds of thousands of other users—industrial, hobbyist, and educational—due to its ease of use. It's a joy for the team at Revolution Education to see all the wonderful projects created by users, both young and old, around the world.

In this book, Ron has worked hard to explain how the PICAXE system operates through simple examples, and I'm sure his easy-to-read style will help many people progress with their PICAXE projects. With the recent launch of the new M2 series of PICAXE chips described in this book, we hope you can achieve even more than before—and if in need of further help (or proud of a project to share!), why not join the ever-growing PICAXE community at www.picaxeforum.co.uk?

Enjoy!

Clive Seager
Technical Director
Revolution Education, Ltd.

This page intentionally left blank

Acknowledgments

For the past three years, I have had the opportunity of writing the "PICAXE Primer" column in *Nuts and Volts* magazine. I have learned a considerable amount about PICAXE programming during this time, and I want to thank *Nuts and Volts*, as well as my publisher at the magazine, Robin Lemieux, for the opportunity to do so. If you aren't already a subscriber to *Nuts and Volts*, I highly recommend it, along with its sister publication, *SERVO Magazine*. In addition to the "PICAXE Primer" column, *Nuts and Volts* regularly publishes other PICAXE-based articles, as well as a wide range of digital and analog projects. Some of the material in *PICAXE Microcontroller Projects for the Evil Genius*, especially in Part One, has been adapted from material previously published in the "PICAXE Primer" column and is presented here it its new form with the express permission of *Nuts and Volts*. Thank you, Robin!

This book would never have been possible without the initial enthusiasm, and continued support and encouragement of Roger Stewart, my Sponsoring Editor at McGraw-Hill. In addition, Joya Anthony, my Acquisitions Editor, enabled me to keep on track during the long months of writing the manuscript. Without Joya, I would probably still be playing around with yet another idea for inclusion in the book. Patty Wallenburg, my Project Manager, actually made the editing process seem like fun at times, and she was always available to answer my questions or clarify my confusion. Thanks also to the many other people at McGraw-Hill who have been involved in this project. Even though we may not have had the opportunity to communicate directly, I appreciate everyone's part in bringing the book to fruition.

I also want to thank Clive Seager (Technical Director of Revolution Education, Ltd.) for his willingness to read the manuscript and to respond to my many questions about the features of the new M2-class processors. His invaluable assistance has helped considerably in making the contents of this book as technically correct as possible. Any errors that may remain are clearly my responsibility.

Finally, I want to express my love and appreciation to my wife Susan for her patience, understanding, and support throughout this entire project, not to mention the last 34 years! For the past several months, free weekends have been almost nonexistent and I have all but disappeared into my basement workspace. Now that the book is finished and summer has finally arrived, I definitely intend to mend my ways.

This page intentionally left blank

Prologue

The first microcomputer I ever purchased was the Synertek Systems SYM Model 1 computer (see Figure 1). I just couldn't resist the advertising blurb: "The [SYM-1] is one of the most versatile and sophisticated single-board computers available… It's an ideal introduction to the expanding world of microprocessor technology as well as a powerful development tool for design of microcomputer-based systems." In its day (1978), the SYM-1 was state-of-the-art technology: It ran at a blazing 1MHz and had 1KB of memory and two 8-bit I/O ports for expansion—all that for only $239.95! (According to the U.S. Bureau of Labor Statistics, that's more than $800 in today's money.)

Out of the box, the SYM-1 could only be programmed in machine language—a very tedious process, to say the least. However, it certainly provided me with a thorough introduction to the world of microcontrollers. Using it, I was able to develop a simple home control system that enabled me to use any telephone in the house to turn lights on and off, adjust the temperature, set timers and alarms, etc. I was having a grand time until the day the telephone repairman (who came to fix an unrelated problem) ordered me to cease and desist or he would confiscate my SYM-1. Back then, Ma Bell had the legal authority to do exactly that with anything it didn't want connected to the phone

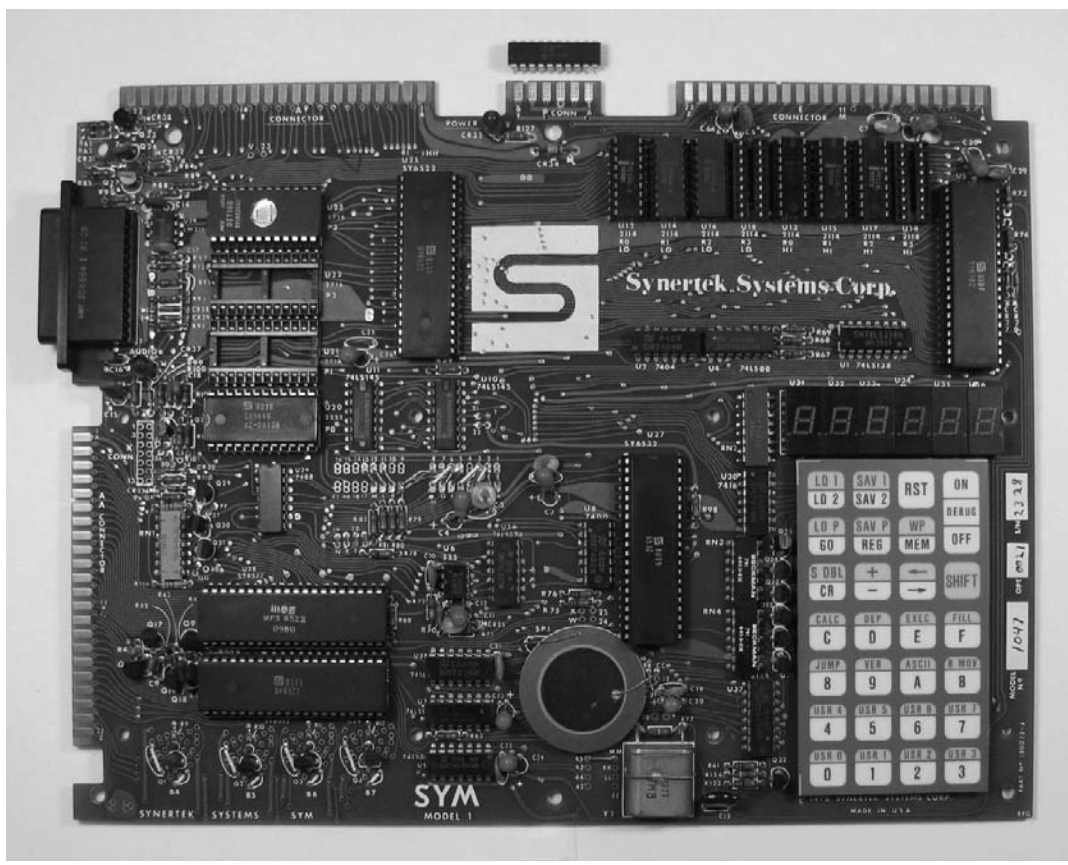


Figure 1 Synertek Systems SYM-1 computer

system, so of course I complied immediately—at least until after he left! It wasn't long before I moved on, first to a Commodore VIC-20 and then an Apple IIe, both of which were based on the same 6502 CPU as the SYM-1, so the transition was an easy one. When I began writing this book, I pulled my SYM-1 out of the closet, dusted it off, and powered it up. Amazingly, it booted up on the first try!

I can't resist comparing the SYM-1 to my most recent microcontroller purchase, Revolution Education's PICAXE-20X2 microcontroller. In Figure 1, you can see the 20X2 sitting just above the SYM-1 in the middle of the photo. The processing power of the 20X2 far exceeds that of the 30 or so chips on the SYM-1's motherboard. It can run at speeds up to 64MHz and has a built-in BASIC interpreter (*much* easier to program than machine language), 32KB of memory, two 8-bit I/O ports, and separate serial lines as well—all that in a single chip that sells for about $6! In the SYM-1's favor, it included a built-in keypad for input and an LED display for output, but both of these devices are relatively simple additions to a 20X2 processor. In fact, we will do exactly that in Chapters 11 and 12.

NOTE **PICAXE is a registered trademark licensed by Microchip Technology, Inc. The PICAXE® product line is developed and distributed by Revolution Education, Ltd. Revolution Education, Ltd. is not an agent or representative of Microchip and has no authority to bind Microchip in any way.**

The PICAXE-20X2 is only one of nine microcontrollers that are currently produced and distributed by the British firm of Revolution Education, Ltd. (www.picaxe.co.uk). Essentially, a PICAXE microcontroller is a Microchip PIC microcontroller with a preinstalled proprietary BASIC interpreter. To program any PICAXE processor, all you need is a simple serial or USB connection to a Mac or PC and Revolution Education's free Programming Editor or AXEpad software, with which you can edit, develop, and test your BASIC programs. In Chapter 1, we'll jump right into the details of using the Programming Editor software when we develop our first project ("Hello World!").

Revolution Education (RevEd) released its first microcontroller way back at the turn of the century. Figure 2 presents some of the main features of the current PICAXE lineup, which is divided into two subgroups: Educational and Advanced. The Educational line is primarily designed to provide an introductory experience in microcontroller programming for students in the lower and middle school grades, and is widely used throughout the British public school system today. However, precisely because the M2 chips are very inexpensive ($3 or $4 each) and simple to program, they also provide a great entry point for students and hobbyists of all ages.

When compared to the Educational line, the processors in the Advanced line have much greater capabilities in several areas; size of program and variable storage areas, number of I/O pins, and speed of operation are the easiest to spot. However, those features are only a part of the differences between the two categories. There are a significant number of additional features to be found in the Advanced processors that couldn't possibly be summarized in a brief table, but by the time you have finished working through all the projects in this book, you will have a good understanding of many of the advanced features to be found on PICAXE processors.

Overview

The central concept underlying the organization of this book is that of "multiprocessor" project design. In other words, we're going to use a PICAXE processor from the Advanced line (the 20X2) to implement a "master processor" project

	Educational Line						Advanced Line		
PICAXE Processor	**08M2**	**14M2**	**18M2**	**20M2**	**28X1**	**40X1**	**20X2**	**28X2**	**40X2**
Pins	8	14	18	20	28	40	20	28	40
I/O Pins	6	12	16	18	22	33	18	22	33
10-Bit ADC Input Pins	3	7	10	11	4	7	11	9	12
Touch Input Pins	3	7	10	11	-	-	-	-	-
PWM Output Pins	1	4	2	4	2	2	1	2	2
Hardware Interrupt Pins	-	-	-	-	-	-	2	3	3
Approx. Num. BASIC Lines	200	600	600	600	1000	1000	1000	4000*	4000*
General Purpose Vars. (Bytes)	28	28	28	28	28	28	56	56	56
Peek/Poke Storage Vars. (Bytes)	48	256	256	256	96	96	128	256	256
Get/Put Scratchpad Vars. (Bytes)	-	-	-	-	128	128	128	1024	1024
Data (EEPROM) Memory (Bytes)	256**	256	256	256	256	256	256	256	256
Maximum Speed (MHz)	32	32	32	32	20	20	64	40	40

* **Total for 4 Program Slots**

** **Data Memory Shared with Program Memory**

Figure 2 Features summary for current PICAXE processors

and then develop several intelligent peripherals based on PICAXE processors in the Educational line (especially the 08M2). For example, in Chapter 11 we're going to use the PICAXE-08M2 to develop a serially interfaced keypad that can be connected easily with our 20X2 master processor. Of course, the 20X2 (or any PICAXE processor, for that matter) could also be connected directly to a keypad without the help of a peripheral processor, but the approach we are going to take has two major advantages. First, once we have developed an "intelligent" peripheral (e.g., our keypad), it becomes a stand-alone device that can simply be connected to any future project we tackle. If we have dozens of projects, we don't need dozens of keypads; we can simply move the keypad from project to project as the need arises. Second, our multiprocessor approach greatly simplifies the process of software development as we move into more complex projects because many of the programming details of the I/O interfaces will already have been thoroughly developed and debugged before interfacing the peripheral device with the master processor. Essentially, we're using a "divide and conquer" approach to both the hardware and software development of complex projects.

In Part One of this book, we're going to start at the very beginning. Our first project is super-simple and doesn't require any previous knowledge of PICAXE programming. Throughout the remainder of Part One we'll focus primarily on the 08M2 and 20M2 processors, and cover some of the essential elements of PICAXE programming and I/O interfacing. By the time you have completed the projects in Part One, you will be ready to tackle Part Two, in which we will implement our master/peripheral paradigm.

In Part Two, we will begin by developing a master processor circuit based on the PICAXE-20X2 and then move on to the development of several stand-alone peripherals for our master processor, including:

- Infrared remote receiver
- Sixteen-character-by-two-line LCD
- 4 by 4 matrix keypad
- Four-digit LED display

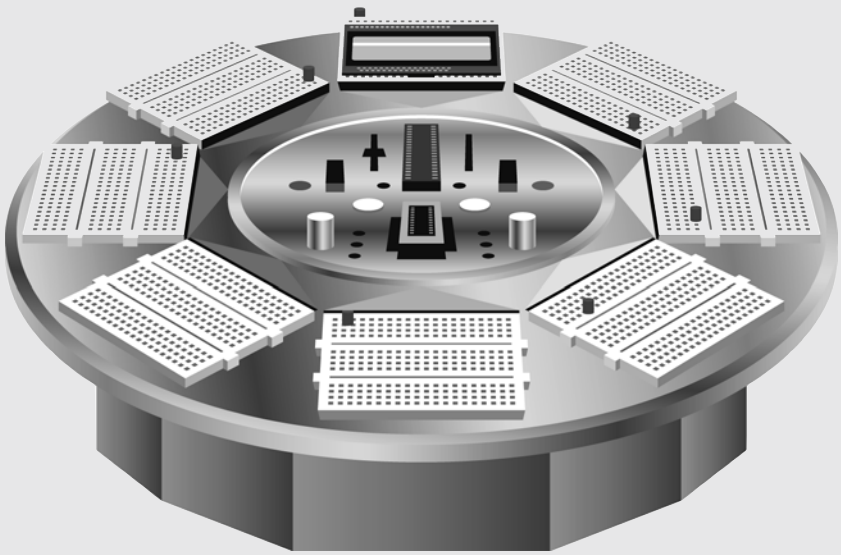
In Part Three, we'll apply our master/peripheral paradigm to the development of Octavius, a sophisticated robotics experimentation platform that includes a 40X2 master processor and eight breadboard stations, which span its perimeter and provide us with the necessary workspace to develop a range of sensory/motor peripherals to augment Octavius' functioning. Octavius also includes a unique time-slice arrangement that enables him to communicate with his numerous sensory/motor peripheral processors and greatly simplifies the software necessary for monitoring his various processes and successfully navigating his way through the environment.

A Brief but Important Note Concerning All the Projects in the Book

When I began writing this book, the new M2 class of PICAXE processors had not yet been announced. All the projects in the book that use a smaller chip were originally developed using the older M-class processors (08M, 14M, and 20M). When RevEd announced the new M2 processors, I decided to modify all the relevant projects so that the programs and information in the book would be as up to date as possible. However, since many people (including myself) still have the older M-class processors on hand, the relevant projects can also be implemented on the corresponding M-class processor with only minor modifications to the software presented in the book. When you visit my website (www.JRHackett.net) to download the software for the projects, you will see that I have included an "M" version and an "M2" version for every relevant program; simply download the version that will work with the processor you have available.

PART ONE

PICAXE Basics



This page intentionally left blank

CHAPTER 1

Introduction to PICAXE Programming and Projects

I HAVE NEVER BEEN A FAN of microcontroller project books that take three or four chapters to get to the first project, so we're going to tackle the "hands-on" part as soon as possible. However, there are a few things about the PICAXE programming system that we need do to cover before we jump into our first project. Essentially, the required information can be divided into four areas:

- Choosing a PICAXE processor
- Interfacing a project with your Mac or PC
- Using RevEd's free Programming Editor or AXEpad software
- Programming in PICAXE BASIC

Choosing a PICAXE Processor

For our first project, we want to get started as quickly as possible, so we're going to use the PICAXE-08M2. It's the smallest (and therefore the simplest) processor in the PICAXE lineup, but don't let that fool you—the 08M2 packs a surprising amount of computing power in its little eight-pin package, which is shown in Figure 1-1. The 08M2 can operate with a power supply anywhere between +1.8V and +5.0V, so the simplest way to power it is to use a two- or three-AA cell battery pack, which is what we'll do in our first project.

CAUTION **Never use a four-cell battery pack. Six volts can easily damage or destroy any PICAXE chip.**

The power connections for the 08M2 are made to pins 1 (+V) and 8 (0V or Ground). The Serial In and Serial Out pins are used for downloading programs from your Mac or PC. (We'll get to that shortly.) Once a program is downloaded to the 08M2, the Serial Out pin can also function as an output for your program, and the Serial In pin can function as an input, which gives the 08M2 a total of six I/O pins: output C.0, inputs C.3 and C.5, and I/O pins C.1, C.2, and C.4. The different numbering of the I/O pins and external pins can be

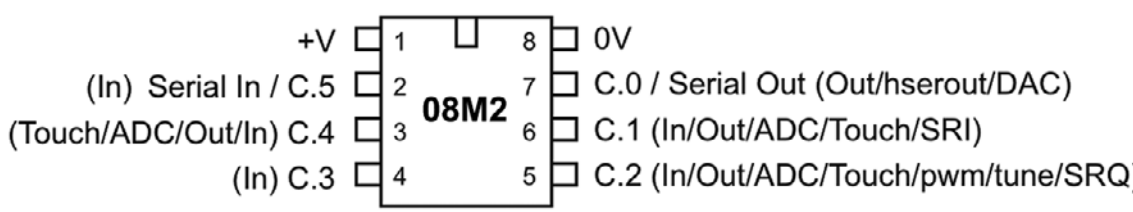


Figure 1-1 PICAXE-08M2 pin-out

a little confusing at first, but it's necessary because of the underlying structure of the Microchip PIC microprocessor on which the 08M2 is based. As you can see in Figure 1-1, many of the 08M2's I/O pins have multiple functions. We'll get into the details as the need arises; right now, I just want to mention that the function listed closest to each pin is always that pin's default function. In other words, when a chip is first powered up, each I/O pin is configured to implement the function that is adjacent to it, as shown in Figure 1-1. Any pin that can be either an input or an output always starts up as an input. This is a safety precaution that prevents the activation of any peripheral devices until your program is properly configured.

If you have any previous experience with microcontrollers, you will notice the absence of dedicated crystal or resonator pins in Figure 1-1. Part of the simplicity of the 08M2 is its internal 4MHz resonator (which can also be switched to 8, 16, or 32MHz by your program). The internal resonator is not as accurate as the external one found on many of the larger PICAXE processors, but it's accurate enough for the vast majority of 08M2 projects. If greater accuracy is needed, there's a BASIC command (*calibfreq*) that allows you to fine-tune the 08M2's operating frequency. Of course, you would need a frequency counter or oscilloscope to make the necessary adjustments.

Interfacing a Project with Your Mac or PC

Back when the first PICAXE chip was introduced, the majority of PCs had only parallel and serial I/O ports. USB ports had already been developed, but they were not yet widely available on new PCs. As a result, programs were originally downloaded from a PC to a PICAXE chip by means of a simple three-wire serial interface that's still in use today. However, virtually all new Macs and PCs no longer include serial connectors, so it won't be long before the serial port is totally obsolete. Fortunately, the PICAXE programming interface is essentially the same for serial or USB connections. Actually, there are two versions of the programming interface—basic and enhanced—both of which are presented in Figure 1-2. The basic version only includes the 10k and 22k resistors; the enhanced version adds two optional parts (the BAT85 diode and the 180Ω resistor) to improve the accuracy of the serial download circuit. For USB connections, the 10k and 22k resistors are all that's required, but the enhanced circuit will also function correctly in this situation as well.

If you still use a serial connection to your computer, my recommendation would be to try the basic interface first. If programs download reliably to your PICAXE processors, you're all set. If a download occasionally fails, try including the 180Ω resistor and the BAT85 diode. If you do include the diode, be sure to install it "backwards," that is, with its anode (rather than its cathode) connected to Ground, as shown in Figure 1-2.

If your computer only has USB ports, you will need a USB-to-serial adapter to program any PICAXE chip. Before you run out and buy one, however, you need to know that most currently available adapters simply don't work with PICAXE processors. Fortunately, RevEd produces the AXE027 USB programming cable, which is available at www.sparkfun.com (SKU: PGM-08312) and elsewhere. The AXE027 USB programming cable terminates in a 3.5-mm mini-stereo plug that provides the necessary serial in, serial out, and Ground connections for programming all PICAXE processors. Unfortunately, a mini-stereo plug is obviously not what you would call "breadboard-friendly." Since most of the projects in this book will be implemented on breadboards, we're going to need a way to adapt the mini-stereo plug for breadboard use. We will do exactly that in our "Hello World!" project later in this chapter.

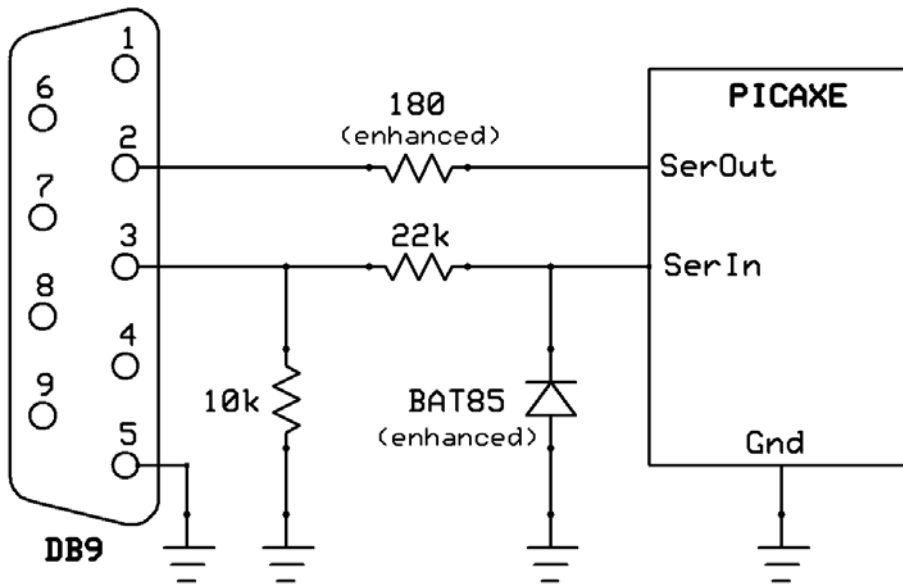


Figure 1-2 PICAXE programming circuit

Before we move on to the Programming Editor software, I should mention that RevEd also makes a serial programming cable (the AXE026), which is also available at SparkFun (SKU: PGM-08313). The AXE026 serial cable terminates in the same mini-stereo plug as the AXE027 USB cable. If you will be using a serial programming connection, you may want to purchase an AXE026 cable because it connects to the same adapters that we will be using throughout this book. Also, it probably won't be long before you will be forced to upgrade to USB anyway; having the same connector on both cables will ensure that any adapters you construct or purchase will still be functional when you switch to a USB connection.

Using RevEd's Free Programming Editor or AXEpad Software

Historically, PICAXE programming was limited to users of the Windows operating system. Mac users had to run Virtual PC or some other emulation software in order to run the free PICAXE Programming Editor. (There's a bit of irony in a Porsche pretending to be a Fiat, but we won't get into that here!) However, early in 2009 RevEd released their new AXEpad software with versions that run on Windows, Mac OS X, and Linux systems, so now everyone can join the party.

AXEpad was specifically designed to run on the new "budget" netbooks, which have considerably less memory and processing power than standard laptop and desktop PCs. As a result, AXEpad isn't as full-featured as the Programming Editor (ProgEdit). For example, it lacks three major features that are included in the Programming Editor: flow charts, program simulation mode, and automatic BASIC-to-assembly language conversion. However, AXEpad does support the majority of ProgEdit's standard development functions, and it's certainly capable of handling all the projects we will be implementing throughout this book. So if you are a Mac or Linux user, you should give AXEpad a try. If you're comfortable with Apple's Boot Camp, VMware's Fusion, or Parallel's Desktop, you can also install Windows on your Mac and run the Programming Editor in a virtual machine. Of course, if you are running Windows on a PC, you can use whichever program you prefer.

ProgEdit and AXEpad are both available on the RevEd website (www.picaxe.co.uk). Just click the Software tab near the top of the page and scroll down until you find the link for downloading the latest version of the software you want to use. While you are at the RevEd site, you can also download the drivers for the AXE027 programming cable. In addition, be sure to download all three sections of the PICAXE Manual (Part I: "Getting Started," Part II: "BASIC Commands," and Part III: "Interfacing Circuits"); they contain a wealth of helpful information. The three parts of the manual are also available under the Help menu of either software package, but it's handy to be able to view them on your computer without having to run a separate piece of software. Finally, at the RevEd website check out the PICAXE Forum—just click the Forum tab near the top of the home page. The forum is the primary meeting place for more than 50,000 PICAXE enthusiasts, and it's a resource that's well worth joining. A quick search of the Forum archives will usually provide helpful answers to any question you may have. In those rare instances when you can't find what you need, just post your question to the forum with all the relevant details. You're bound to get helpful information and advice from the membership.

When you have downloaded your choice of PICAXE programming environments, simply double-click the installer icon and follow the onscreen directions for installation on your computer. When we get to our first project, we'll discuss the basics of using the software to develop and download a program to a PICAXE processor. We'll focus on ProgEdit throughout the book, but the principles are similar for the AXEpad software as well.

Programming in PICAXE BASIC

The PICAXE BASIC language is similar to (but much more powerful than) many other variations of the language. In addition to the usual BASIC commands (branch, do…loop, for…next, gosub, goto, if…then…else, select case, etc.), there are many specialized commands to accomplish a variety of useful I/O tasks in a simple manner. To whet your appetite, here's a brief list of some of the advanced functions that can be implemented with the built-in commands in PICAXE BASIC:

- Analog-to-digital conversion
- i2c I/O
- Infrared I/O
- Interrupt processing
- "One-wire" I/O
- PC keyboard input
- Pulse-length measurement and production
- Pulse-width modulation (PWM) for DC motor control
- Serial I/O
- Servo motor control
- Sound and music output
- Serial Peripheral Interface (SPI) I/O
- Table lookup and lookdown
- Temperature measurement

The documentation for all the PICAXE BASIC commands is contained in Part II of the PICAXE Manual. We'll explore many of these commands throughout the pages of this book, but you may also want to spend some time browsing the documentation in Part II of the manual to get a sense of the scope of PICAXE BASIC.

Breadboards, Stripboards, and PC Boards

In this book, we will be focusing on three different circuit construction techniques: breadboards, stripboards, and PC boards. Each one of these approaches to circuit construction has advantages in different situations, and we'll capitalize on these advantages.

- **Breadboards** are by far the most flexible approach to use in the early stages of project development. They are inexpensive, quick to set up, and easy to modify when changes are needed. When working with breadboards, it's important to remember that neatness counts! There's nothing more frustrating than trying to debug a breadboard circuit that's a mass of tangled jumper wires. That's why I strongly recommend that you always use the preformed jumper wires that you will see in our "Hello World!" project later on.
- **Stripboards** are simple protoboards with holes that are evenly spaced on a 0.1-inch (2.54-mm) grid and connected by rows of copper traces on the bottom of the board (see Figure 1-3). They can be helpful in two different situations: adapting components that are not very breadboard-friendly for use on a breadboard, and constructing small circuits that can be easily moved from breadboard to breadboard as the need arises. As part of our "Hello World!" project, we'll construct a simple stripboard circuit to adapt the standard PICAXE mini-stereo connector for use in our breadboard circuits. In Chapter 2, we'll get into the details of stripboard construction and make a complete programming adapter for use with our projects.
- **PC boards** are by far the most reliable method of circuit construction, but they tend to be expensive to manufacture in small quantities as well as difficult and messy (and possibly toxic) to produce at home. However, the size or complexity of some circuits requires their use. For example, I doubt that our "Octavius" project in Part Three could be implemented without the use of a PC board. In spite of that fact, breadboard circuits will also play a central (perhaps I should say "peripheral") role in Octavius' construction.

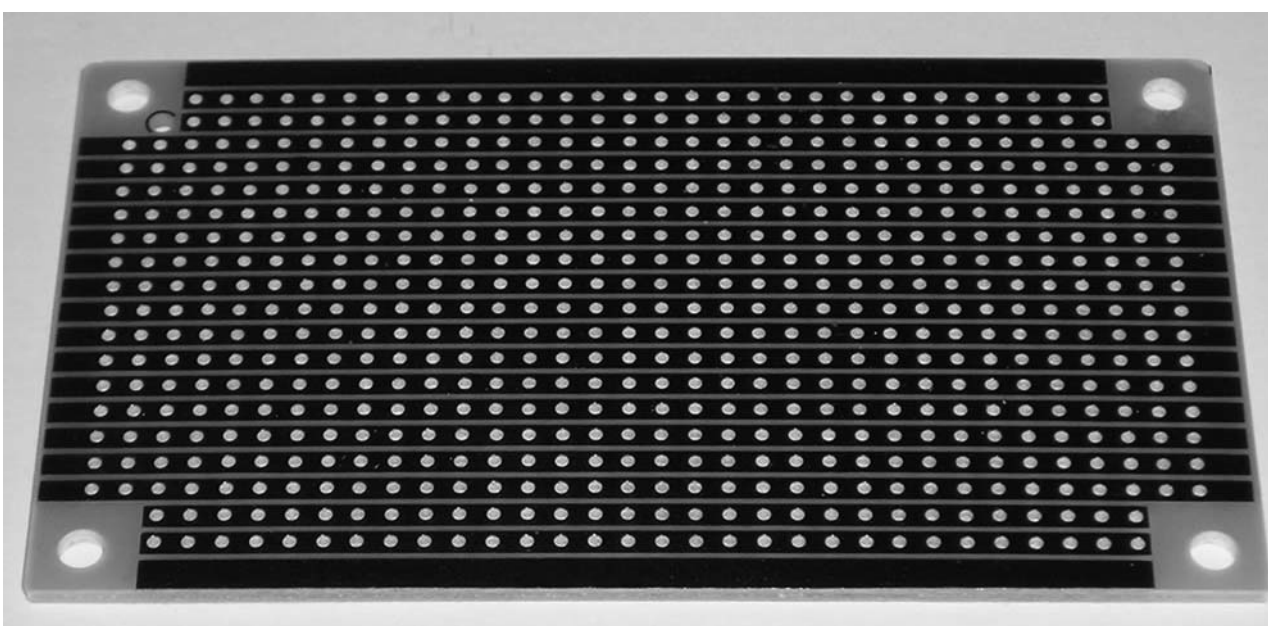


Figure 1-3 Bottom view of a typical stripboard

Project 1
"Hello World"

Traditionally, introductions to microcomputer programming always begin with a basic "Hello World!" project. For computers that include output to a TV or monitor, such as the Commodore VIC-20 I used many years ago, this program simply prints the phrase "Hello World!" (or something similar) on the output screen as a demonstration that the system is functioning correctly and that it has been programmed properly by the user. For microcontrollers without a character-based output device, such as simple PICAXE systems, the corresponding "Hello World!" program usually involves blinking a light-emitting diode (LED) on one of the processor's outputs as proof that everything is functioning properly. In Chapter 10 we'll develop a character-based liquid crystal display (LCD) as an output device for our PICAXE projects, but for our first project, we'll stick with the traditional "blinking LED" approach to demonstrate that we're on the right track.

The complete parts list for our "Hello World!" project is presented in the Parts Bin. The listed sources are only one possible suggestion; most of the parts we will be using are readily available from a variety of suppliers.

PARTS BIN

PICAXE AXE027 USB programming cable (sparkfun.com, SKU: PGM-08312)
Breadboard, 400 points (pololu.com, item #351)
Jumper wires, pre-formed (pololu.com, item #347 or 354)
3-AA battery holder, enclosed, with switch (pololu.com, item #1152)
3 AA alkaline batteries (wherever)
Capacitor, .01μF (jrhackett.net)
Header, male, 10-pin, "reverse-mountable" (jrhackett.net)
LED, resistorized (red or green) (jrhackett.net)
Mini-stereo jack, 3.5-mm, low-profile (jrhackett.net)
PICAXE-08M2 (or 08M) (jrhackett.net)
Resistor, 10k, 1/4 watt (jrhackett.net)
Resistor, 22k, 1/4 watt (jrhackett.net)
Stripboard, small (jrhackett.net)

Step 1: Construct the Stripboard Mini-Stereo Jack Adapter

We're going to begin our first project by constructing the mini-stereo jack adapter, which uses three of the listed parts: stripboard, mini-stereo jack, and 10-pin "reverse-mountable" male header, which requires a brief explanation. Most of the stripboards that we will construct (including our mini-stereo adapter) will function by being inserted into a breadboard circuit. This is usually accomplished by using a male pin-header. If we were constructing a PC board adapter, this would simply involve inserting the male header from the bottom of the PC board and soldering it on the top. Stripboards, however, only have copper traces on the bottom, so we can't do any soldering on the top. The solution to this problem (which we'll refer to as "reverse mounting") involves using a male header that is slightly longer than standard to compensate for the thickness of the stripboard, inserting the long ends of the pins through the stripboard from the top and soldering the pins on the bottom. To do this successfully, the pins should be at least 0.32 inches (8.1 mm) long, which is the length of the reverse-mountable headers that I carry. If you have male headers with pins that long, I'm sure they would work just as well.

The schematic for our stereo jack adapter is presented in Figure 1-4. The circuit is designed to make our first stripboard as simple as possible. All

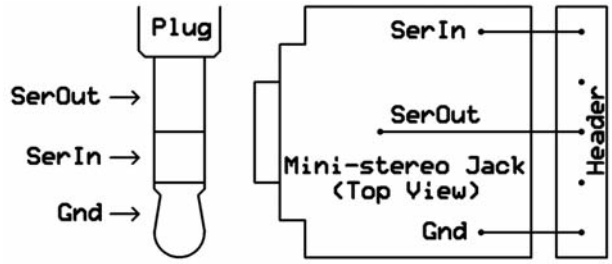


Figure 1-4 Schematic for the mini-stereo jack adapter

we are doing is bringing the three signals we need straight out from the mini-stereo jack to the pins of a five-pin male header. Figure 1-4 also includes the corresponding labeling for the functions of each of the three segments of the mating mini-stereo plug of the AXE027 cable. If you want to use your own mini-stereo jack, you will need this information to determine whether its pin-out is the same as the jack in the parts list.

To construct the stereo jack adapter, first cut a small piece of stripboard that contains five traces, with five holes in each trace. I use a band saw for this purpose, but a small coping saw also works well. The easiest approach is to "sacrifice" a series of holes for the cut and then sand the board down to its final size (see Figure 1-5).

To assemble the adapter:

1. Snap the 10-pin male header into two 5-pin pieces, and insert the longer ends of the pins of both pieces through the stripboard (from the top) so that the pins are in the holes at each end of the five traces.
2. In order to support the board while soldering, flip it upside-down (with the headers still inserted), and insert the shorter ends of the headers into a breadboard (see Figure 1-6).
3. Solder the five pins of *one* header in place. The other header is only being used to support the stripboard—we'll remove it in the next step.
4. Remove the stripboard assembly from the breadboard, save the extra five-pin header for another project, and snip off the short ends (top of board) of the soldered header. You may also want to file the cut ends of the pins smooth at this point.
5. Insert the stereo jack into the top of the stripboard so that its round opening is facing *away* from the five-pin header and its middle pin is in the end hole of the middle trace of the stripboard (see Figure 1-7).

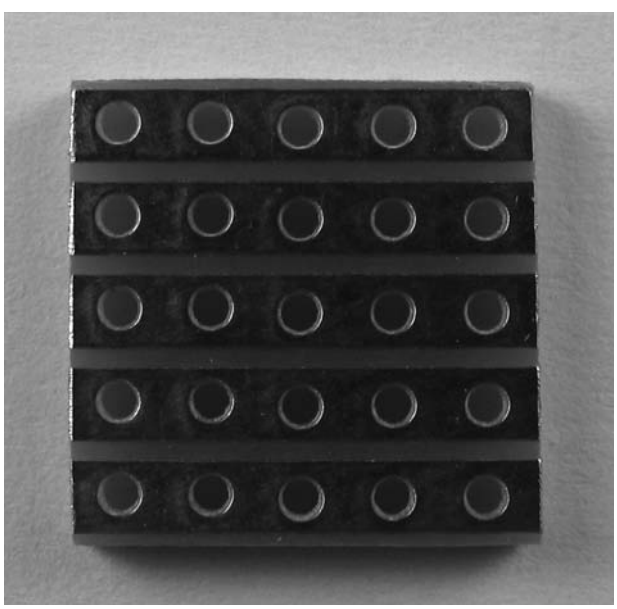


Figure 1-5 Stripboard after cutting and sanding

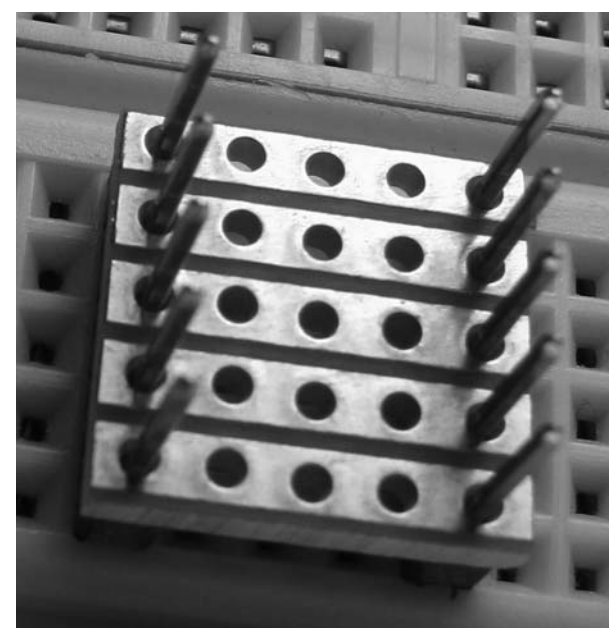


Figure 1-6 Stripboard ready for soldering

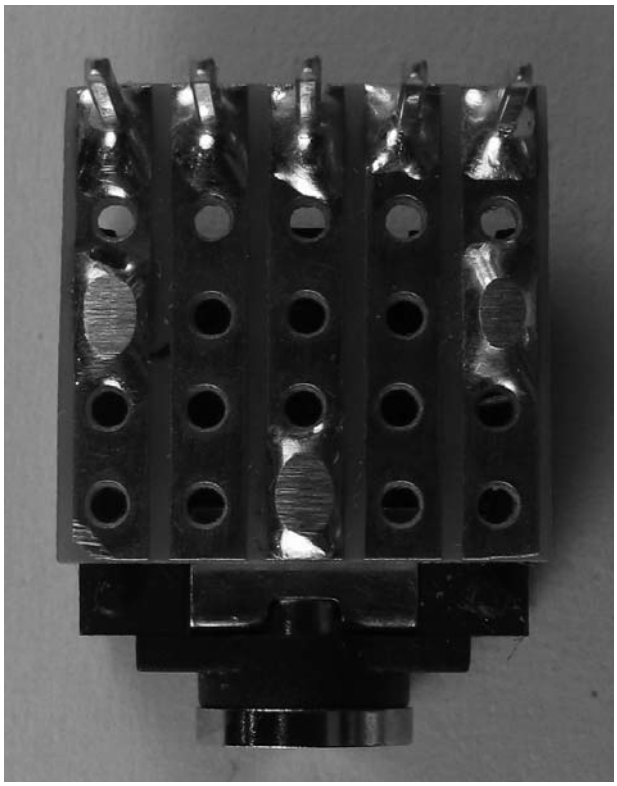


Figure 1-7 Bottom view of the completed mini-stereo jack adapter

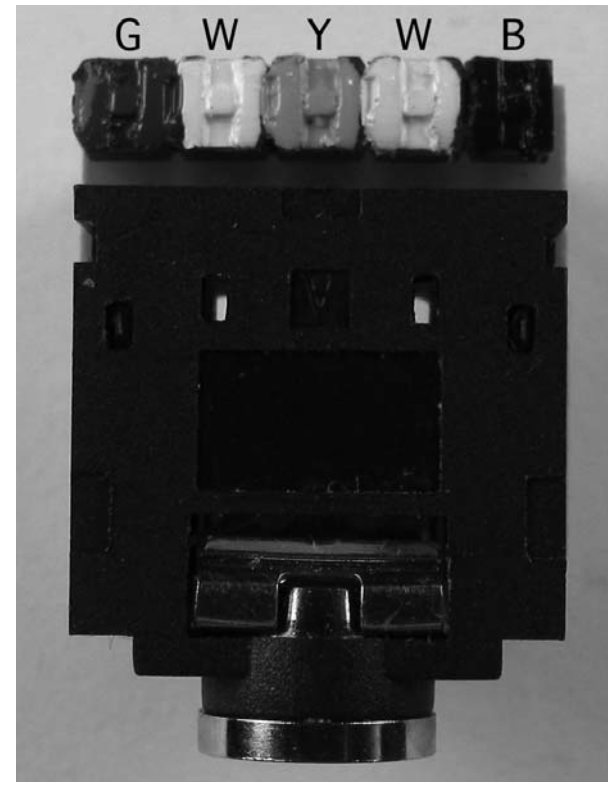


Figure 1-8 Top view of completed mini-stereo jack adapter

6. Flip the board and stereo jack upside-down again, place it on a flat surface, and solder the three pins of the stereo jack to the board. Because the adapter is going to be inserted into a breadboard, it's a good idea to snip the protruding pins of the jack as short as possible and file them smooth (again, refer to Figure 1-7).
7. Using an old toothbrush and isopropyl alcohol or paint thinner, clean the flux from the bottom of the board and allow it to dry.

Figure 1-8 is a photograph of the completed adapter. Because the photo in Figure 1-8 is not in color, you can't see that I have painted the tops of the header to remind myself of the function of each pin. (To identify the colors I used, I have added a single-letter label to each pin.) I used small jars of Testors model paints and a small detail brush from the local hobby shop for this purpose. Stripboard headers tend to be difficult to label with words or symbols, so I usually use the following mnemonic color-coding scheme to help me identify the pin functions. When you get to a certain age, you need all the memory aids you can get—trust me!

- **Black = Ground** (of course)
- **Green = SerIn** because nowadays it's "in" to be green
- **Red = +5 volts** (naturally)—not used in this project
- **White = No Connection** because I already used black
- **Yellow = Serout** because you "yell out"

Step 2: Assemble the Breadboard Circuit

The schematic for the "Hello World!" project is presented in Figure 1-9. Note the 330Ω current-

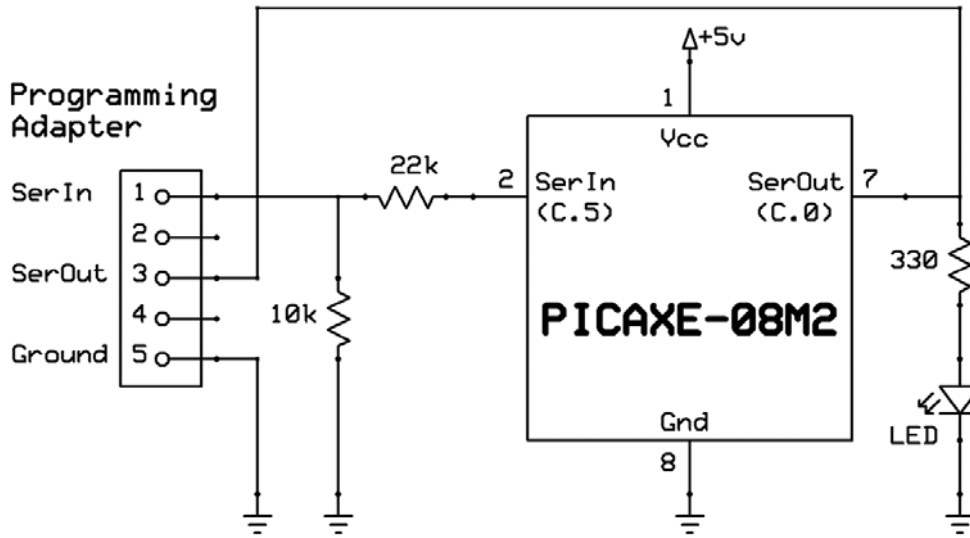


Figure 1-9 Schematic for the "Hello World!" project

limiting resistor in series with the LED. If you decide to use the "resistorized" LED from the parts list, you should omit the external current-limiting resistor. However, if you use a "regular" LED, be sure to include the external resistor.

Using the schematic shown in Figure 1-9 and/or the photo of the completed project shown in Figure 1-10, assemble the "Hello World" circuit on the breadboard. Be sure to connect the cathode (shorter lead) of the LED to Ground. If you decide to "tin" the ends of the battery holder's wires before inserting them in the breadboard, be sure to use very little solder—it's easy to make them too thick to fit in the breadboard holes. (You could also solder the battery leads to a two-pin male header if you prefer.) If you don't tin the leads, twist the ends of the wires by hand before inserting them in the breadboard.

In Figure 1-10, you can see the .01µF decoupling capacitor inserted into the power and ground rails in the upper-left corner of the photo. If you look back at the schematic in Figure 1-9, you can also see that I haven't included it there. A decoupling capacitor is a good idea in every project you build—it helps decrease unwanted "noise" in the power lines. I consider it to be part of the breadboard setup, so I don't generally include it in the schematic, and I will probably omit it from the parts list for the remainder of our projects, but don't forget to include one on every breadboard project you construct.

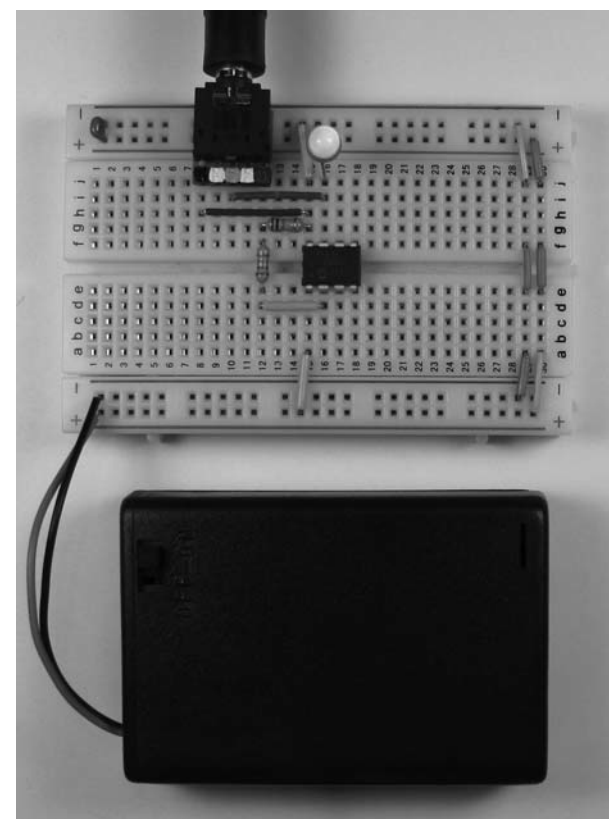


Figure 1-10 Completed "Hello World!" project

As I mentioned earlier, the serial output pin can also function as a general-purpose output. If your output is as simple as an LED, you can leave it and the programming cable connected at the same time. If you do, the LED will flicker rapidly during a program the download, which can be a reassuring indication that download is proceeding properly. We're using output C.0 for the LED in our first project so that you can see how it behaves during a program download. However, if you plan to use output C.0 for something more involved (e.g., motor control), it's best to disconnect the device during a program download. Some motors and other devices can behave erratically or even be damaged during a program download. To avoid those risks, it's easier to use output C.0 for an LED whenever possible.

Step 3: Programming the PICAXE-08M2 Processor

If you haven't already done so, install the ProgEdit software (or AXEpad, if you prefer) and the drivers for the AXE027 USB cable onto your computer. Before running ProgEdit, be sure your AXE027 cable is plugged into an available USB port. When you do run ProgEdit, the Options window will probably appear. For now, just close it—we'll discuss the various options as we need them. Our "Hello World" BASIC program listing is presented in Listing 1-1. As you can see, it's simple, so it can be quickly typed into ProgEdit. However, as I mentioned in the Prologue, all the programs in this book are also available for downloading from my website (www.JRHackett.net). Make sure you use the correct version of each program (M-class vs. M2-class) for your processor.

When you have finished typing in the program, save it with the name "HelloWorld"—the ".bas" will be automatically added. Before we actually run it, there are three aspects of the program (actually, *any* PICAXE program) that I want to emphasize.

- **Comments:** Any text that follows an apostrophe or a semicolon is treated as a comment. In other words, it's only there for

LISTING 1-1

```
' ============================== HelloWorld.bas ==============================
'  This program runs on a PICAXE-08M2.
'  It blinks an LED to say "Hello World!"

' === Constants ===
  symbol abit = 500                ' used to adjust blink rate
  symbol  LED = C.0                ' LED on output C.0 (pin 7)

' === Directives ===
  #com 3                           ' specify serial port
  #picaxe 08M2                     ' specify processor

' ============================ Begin Main Program ============================
do
   high LED                        ' LED on
   pause abit                      ' slow it down
   low LED                         ' LED off
   pause abit                      ' slow it down
loop                               ' loop forever
```

us humans. The PICAXE BASIC compiler ignores comments, and they don't add to the length of the downloaded program. They may seem tedious to type, but they are invaluable. Without them, a program that you thoroughly understood when you originally wrote it will seem incomprehensible six months later—I guarantee it! So be sure to include copious comments in all the programs you write.

- **Constants:** Constants are another convenience for us humans. In the "Hello World" program, we really don't need either of the constants that I declared. For example, in the main body of the program, I could have written "high C.0" and "pause 500" and skipped the constant declarations completely—the program would run exactly the same and be the same length. However, there are two excellent reasons for *not* taking that approach. First, constants give us the opportunity to include meaningful names in our program, which makes it much easier to read and understand. Second, in longer programs you may issue the same command in several different places. For example, suppose you have a dozen or so "pause 500" statements in your program and you decide the delay needs to be a little longer. It's much simpler to change one constant declaration than it is to change a dozen statements sprinkled throughout your program, so the bottom line is the same as it is for comments: use them!
- **Directives:** PICAXE BASIC includes more than a dozen directives (see Part II of the manual). Each one begins with the "#" symbol and can be thought of as a special instruction to the BASIC compiler. Like comments and constants, directives do not add to the length of the downloaded program, but they are a real convenience for the programmer. The two directives that I have included in the "Hello World!" program are probably the ones you will most frequently want to use. As the program comments explain, they specify which processor is being used and which communications port is used to connect our programming cable. Both of these specifications can also be set in the Options window, but it's much simpler to do it within the program itself. (In the current version of AXEpad, it is necessary to use the Options window.) For example, if you are working with two programs running on two different processors (as we will frequently do in Part Two), using the appropriate #picaxe directive in each program completely automates the process of switching back and forth between processors as you develop their respective programs. The same is true for the #com directive. I routinely use two programming connections (one on com 3 and the other on com 4). By including the appropriate #com and #picaxe directives in each program, I can easily switch between two different programs for two different processors (by clicking either one of them) and download the correct program to the correct processor.

We have one more thing to do before we can run our "Hello World!" program—we need to be sure our AXE027 USB cable is properly configured for use with the ProgEdit or AXEpad software. To do so, open the Options window, either by selecting the View | Options menu or clicking the Options button in the toolbar near the top of your editor's window. (If you're not sure which button that is, just point to each one and a tool tip will appear.) When you have opened the Options window, click the Serial Port tab to access the relevant settings. If you have more than one serial port on your computer and/or multiple USB devices connected to it, you will probably see more than one available serial port. In order to determine which serial port is connected to your AXE027 cable, simply click the button labeled "Scan for USB Cable," and a pop-up window will identify the correct serial port for your cable.

If you are using AXEpad on a Mac, the corresponding button is labeled "USB Setup" and the port identification is as follows: #com /dev/tty.usbserial-xxxx, where "xxxx" is the four-digit serial number of your AXE027 cable. In Linux, it's: #com /dev/tty/USB0.

When you have configured your AXE027 cable, you should be ready to run the "Hello World!" program. Close the Options window, turn on the power to your breadboard circuit, and select the PICAXE | Program menu item (or click the Program tool). The program should download—you will see the LED flickering as it's doing so. Once the download has completed, you should see the LED on the breadboard blink on and off about once per second. If you don't, welcome to the wonderful world of troubleshooting!

Debugging a PICAXE Project

The most important aspect of the frustrating process of debugging a project is to take things in systematic and sequentially logical steps. With that in mind, here are a couple of points to consider if you need to debug your "Hello World!" project:

- If the "Hardware not found…" dialog box appeared, make sure the power is on and properly connected to the breadboard. Recheck the AXE027 serial port configuration.
- If the "Downloading Program" dialog box completed successfully but the LED didn't flicker during the process, the LED is probably inserted backwards. Make sure its cathode is connected to Ground.
- If the LED did flicker while the program was being downloaded but it doesn't blink, recheck all the wiring connections in the circuit.

CHAPTER 2

Introduction to Stripboard Circuits

Stripboard circuit construction is becoming a popular approach to hard-wiring small circuits. The simple mini-stereo jack adapter we made in Chapter 1 was super-easy to build and reliable enough to survive being moved from one breadboard project to the next. It's the first of several general-purpose circuits that we'll construct on a stripboard so that we can use them in various projects throughout the book. Some of these circuits will be fairly complicated, so we're going to need a good grasp of some of the details of designing and constructing stripboard circuits. In this chapter, we're going to construct a small stripboard that includes the basic PICAXE programming circuit so that we won't have to duplicate that circuit on every breadboard project that we develop. We will also use the development of our small programming adapter as a vehicle for exploring some of the details of the process of designing stripboard circuits.

Designing Stripboard Circuits

Essentially, there are four approaches to designing stripboard circuits:

- Using a pencil and paper (remember them?)
- Using a computer-aided design (CAD) program specifically designed for stripboards
- Using a CAD program intended for the design of PC boards
- Using a general-purpose drawing program

Using a Pencil and Paper

When I first started working with stripboard circuits, I used a pencil (with a big eraser) and a sheet of graph paper. This approach certainly works for simple circuits, but it does have one major drawback, which we were able to avoid when we constructed our mini-stereo jack adapter. The vast majority of stripboard designs necessitate cutting the copper traces on the bottom of the board at various specific locations to correctly implement the design of the circuit. Of course, when you are looking at the bottom of a stripboard, what you see is inverted 180 degrees from the top view. For example, suppose you are looking at the top of the board and there's a trace that needs to be cut near the top-left corner of the board. When you turn the board over, that cut will be near either the bottom-left corner or the top-right corner, depending on which way you turned the board—an invitation to disaster, to say the least!

My initial way of dealing with this problem was to hang my graph-paper layout in front of a bright light with the back of the paper facing me so that I could view the inverted image and cut the traces correctly. It soon dawned on me that I could improve the situation by scanning my graph-paper layout, using a computer graphics program to flip the image, and then printing out the inverted image for use as I cut the traces. If you don't have a scanner, I'm sure you could do the same thing with a digital camera. This approach certainly works for simple circuits, but it's tedious at best.

Using a CAD Program Specifically Designed for Stripboards

It wasn't long before I started searching for software specifically designed for stripboard layouts. I found a couple of shareware programs, but they didn't seem to be actively supported and weren't much better than my paper-and-pencil approach. I found one commercial product, LochMaster from Abacom Software (www.abacom-online.de/uk/html/lochmaster.html), that's an excellent choice for stripboard layout design. LochMaster's automatic board flipping and 3-D printouts make it a pleasure to use. Figure 2-1 shows the top and bottom views of the LochMaster layout for a simple circuit. As you can see, the bottom view has been vertically flipped. For example, the short jumper that is soldered on the bottom of the board moves from below to above the middle trace when the board is vertically flipped. You can also clearly see the two places on the bottom of the board at which a trace needs to be cut. (We'll get into the details of trace cutting shortly.)

In spite of its visual sophistication, LochMaster has two major drawbacks that led me to decide not to use it in this book: It's relatively expensive and it only runs on Windows. Now that the AXEpad software is available for Macintosh and Linux users, I'm determined to make all our projects readily accessible to everyone.

Using a CAD Program Intended for the Design of PC Boards

If you are a reader of my "PICAXE Primer" column in *Nuts and Volts* magazine, you know that in the past I have also used the free expressPCB software (www.expresspcb.com) for designing stripboard circuits. While free is good, the expressPCB program also suffers from the limitation that it only runs on Windows systems, so it, too, is not suitable for use in our cross-platform approach to PICAXE projects.

The next program I considered was the EAGLE CAD program available from CadSoft (www.cadsoft.de). "EAGLE" is an acronym that stands for "Easily Applicable Graphical Layout Editor." It runs on both Macintosh and Windows computers, and a free Light Edition is available. However, in spite of its name, EAGLE is anything but easy! It's an excellent program for designing PC boards, and I'm sure it could also be used for stripboard design, but the learning curve is just too steep for our purposes. It would take an entire book to fully explain how to use EAGLE, so we're going to move on to our fourth and final approach.

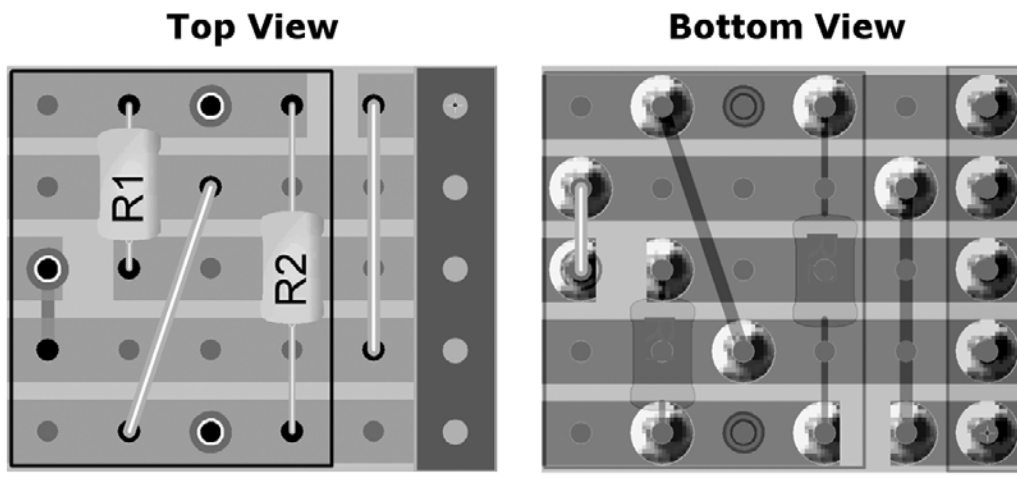


Figure 2-1 LochMaster printout of top and bottom stripboard views

Using a General-Purpose Drawing Program

I spent a fair amount of time on the Web trying to find a cross-platform drawing program for designing stripboard layouts, but nothing seemed suitable. Everything I found was either a demo version of an expensive package or a program suitable for introducing children to drawing on the computer; none had the features I wanted. In desperation, I tried a program that is far from free, but as close to ubiquitous as computer software gets: Microsoft Word. At first glance, Word might seem to be an unlikely prospect for designing stripboard layouts. But as it turns out, Word's drawing features have just the right combination of simplicity and power that we need to get the job done quickly and easily. Figure 2-2 shows the top and bottom views of the Word layout for the same stripboard circuit that we saw earlier in Figure 2-1. As you can see, the Word version isn't nearly as pretty as the LochMaster version: no 3-D effects and none of the realism of the LochMaster layout. However, all the necessary information is contained in the Word layout. Also, I think the stark simplicity of the Word layout actually makes it easier to use. In Figure 2-2, the black jumpers are installed on the top of the board, and the gray jumper is installed on the bottom—this is a standard convention that I will use throughout our projects. As an aside, both LochMaster and Word can be used to draw your stripboard layouts in color. However, because all the photos in this book are grayscale images, I have chosen to do all the stripboard layouts in grayscale.

If you have spent the last few years vacationing on Mars and don't have a copy of Microsoft Word, don't despair. Windows users can download the free expressPCB software I mentioned earlier, and Mac users have a couple of choices. Even though Apple has officially discontinued its AppleWorks suite of office applications, version 6.2 is still available on factory-sealed CDs (just search for "AppleWorks 6"), and Apple still provides a free update to the last version released (version 6.2.9). If you don't have Word, you probably need a good word processor anyway (which is also included in the AppleWorks program), and the AppleWorks drawing program is more than capable of generating respectable stripboard layouts. If you prefer software that's free, try OpenOffice. It provides all the major functionality of Microsoft Office (word processor, spreadsheet, database, presentation software, drawing, etc.) and it runs on Mac OS X, Linux, and Windows systems.

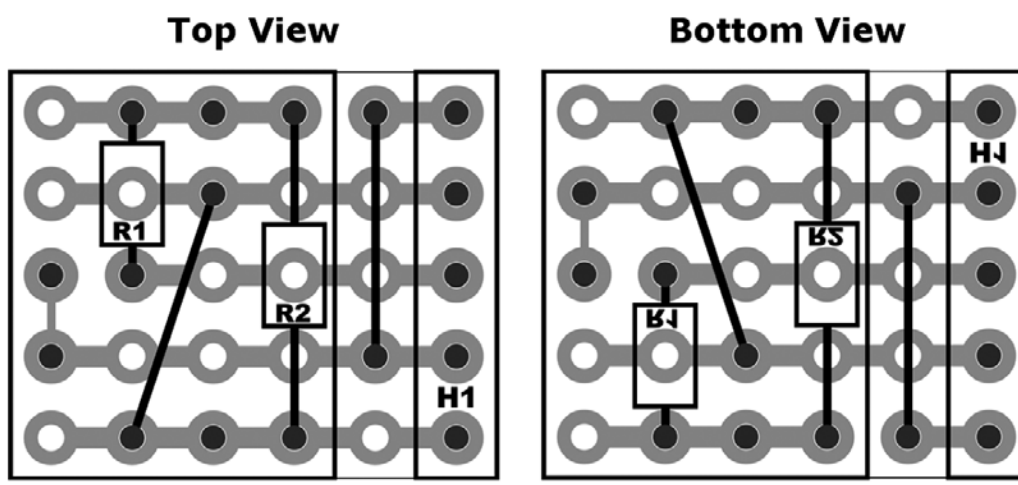


Figure 2-2 Word printout of top and bottom stripboard views

In our next stripboard project in Chapter 3, we'll get more into the details of using Word to generate stripboard layouts, but right now, let's turn our attention to the mechanics of constructing stripboard circuits.

Tools for Stripboard Circuit Construction

We can make two different types of cuts on a stripboard trace. The first type is the easier of the two and therefore the one we will use whenever we can: cutting a trace at a hole. There is a specialized tool for this purpose (appropriately called a "stripboard tool") that consists of a 3.5-mm (~ 9/64-inch) drill bit embedded in a plastic handle. (See the tool in the center of Figure 2-3.) Commercially available stripboard tools are hard to find in the United States and expensive to have shipped from Europe (where they are readily available), so Figure 2-3 also includes two other tools that can be used for the same purpose. The tool at the top of the photo is simply a 1/8-inch (3.2-mm) hex shank drill bit plugged into a matching hex socket; the one at the bottom is a General pin vise (available at Ace Hardware) that's holding a 1/8-inch drill bit. Any one of these three tools works well for cutting a stripboard trace at a hole.

Before you attack a stripboard with your new 18V lithium-ion cordless drill, let me clarify—we're not actually going to drill holes in the stripboard. We're just going to press the bit into the hole where we want to cut the copper trace and twirl it a few times by hand until the trace is completely severed around the hole. It's important to work slowly and check the result between each twirl because it's surprisingly easy to accidentally cut the adjacent traces if you don't. Good lighting and magnification also help minimize disasters.

The second type of cut involves severing a trace *between* two holes. To clarify this distinction, Figure 2-4 presents a close-up photo of both types of cuts. In it, the third trace from the left has been cut between two holes. This type of cut is more difficult than cutting at a hole because it's necessary to leave enough copper around both of the adjacent holes so that a jumper wire or

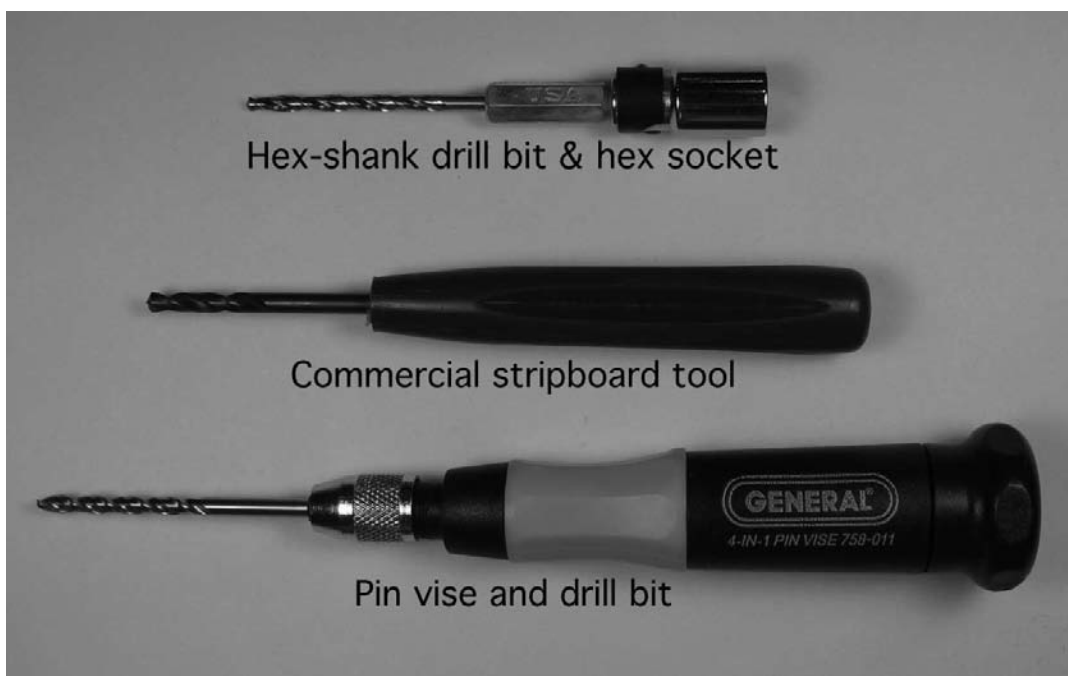


Figure 2-3 Tools for cutting a stripboard trace at a hole

component pin can be soldered into each one of them. (If both holes weren't going to be used for soldering, we could simply remove the trace at the unused hole.) If you look back at the stripboard layout in Figure 2-2, you can see the two locations where we need to make this type of cut. Looking at the top view, one cut is just to the left of the lower lead of resistor R1 and the other is just to the right of the upper lead of resistor R2. As you can see, in both cases, the two adjacent holes are going to require soldering, as indicated by the "filled-in" hole. If you are willing to make your stripboard circuits large enough, you can always avoid this type of cut. For example, if I had made each of the traces in Figure 2-2 longer by two holes, I could have spread out the components and jumpers so that there would be an available hole at which to make the easier type of cut. However, one of the most satisfying aspects of designing a stripboard layout, at least for me, is trying to make the board as small as possible.

The question remains: How do we cut a trace *between* two holes? My first approach was simple—I used a small, sharp hobby knife (again, with good lighting and magnification) to carefully remove a thin slice of copper from the trace between the two holes. This method certainly works, but it has two drawbacks. It takes a fair amount of force to pull the blade through the trace, so it can become tedious if you have many traces to cut. Also, the necessary cutting force can easily facilitate an accidental "slip" resulting in cutting the wrong trace (or worse, a finger—trust me, I know!). In an attempt to avoid these problems, I experimented for a while with using a small rotary tool and a diamond burr to cut the traces, but this approach turned out to be even more error-prone.

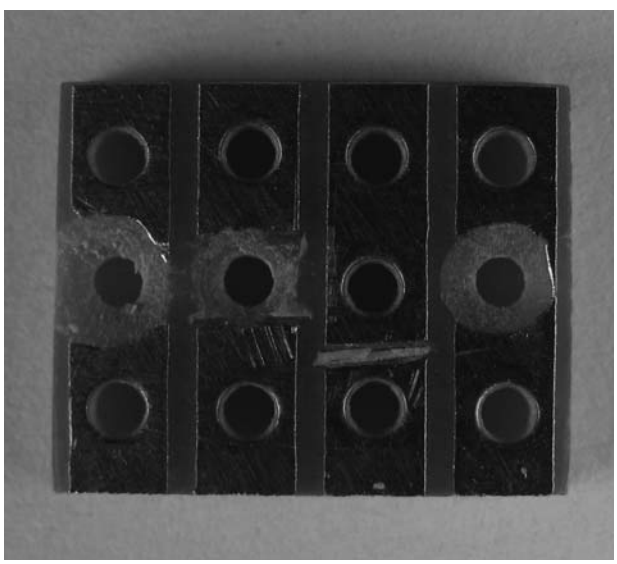


Figure 2-4 Examples of both types of cuts on stripboard traces

My third and final method of cutting a stripboard trace between two holes has completely eliminated any minor "accidents"—except, of course, for occasionally cutting a trace in the entirely wrong place. (A well-known woodworking axiom comes to mind: "Measure twice, cut once." For stripboard construction, we'll paraphrase it: "Count twice, cut once.") To implement this foolproof method, I made a simple cutting tool from a small 1/8-inch flat-bladed screwdriver. (See the top tool in Figure 2-5.) First, I used a grinding wheel to remove a little metal from each side of the blade; that is, I reduced the width of the blade from 1/8 inch (3.2 mm) to about 0.1 inch (2.5 mm) so that the blade is just slightly wider than a stripboard trace. Second, I used a sharpening stone to sharpen the flat edge of the blade. What I ended up with is a miniature chisel with a 0.1-inch blade. My homemade stripboard chisel makes a nice clean cut between two traces, but it does require frequent resharpening. The steel used in inexpensive screwdrivers just isn't hard enough to hold a sharp edge for very long.

Recently, I found a commercial chisel that's just about perfect for between-the-holes trace cutting. (See the tool in the center of Figure 2-5.) It's a 3-mm detail chisel used for small, intricate woodcarvings. Three millimeters is approximately 0.12 inches, which is also slightly too wide for our purpose, so the chisel does require minor grinding or filing of both sides to reduce the blade width to 0.1 inch. However, once its width is reduced, this

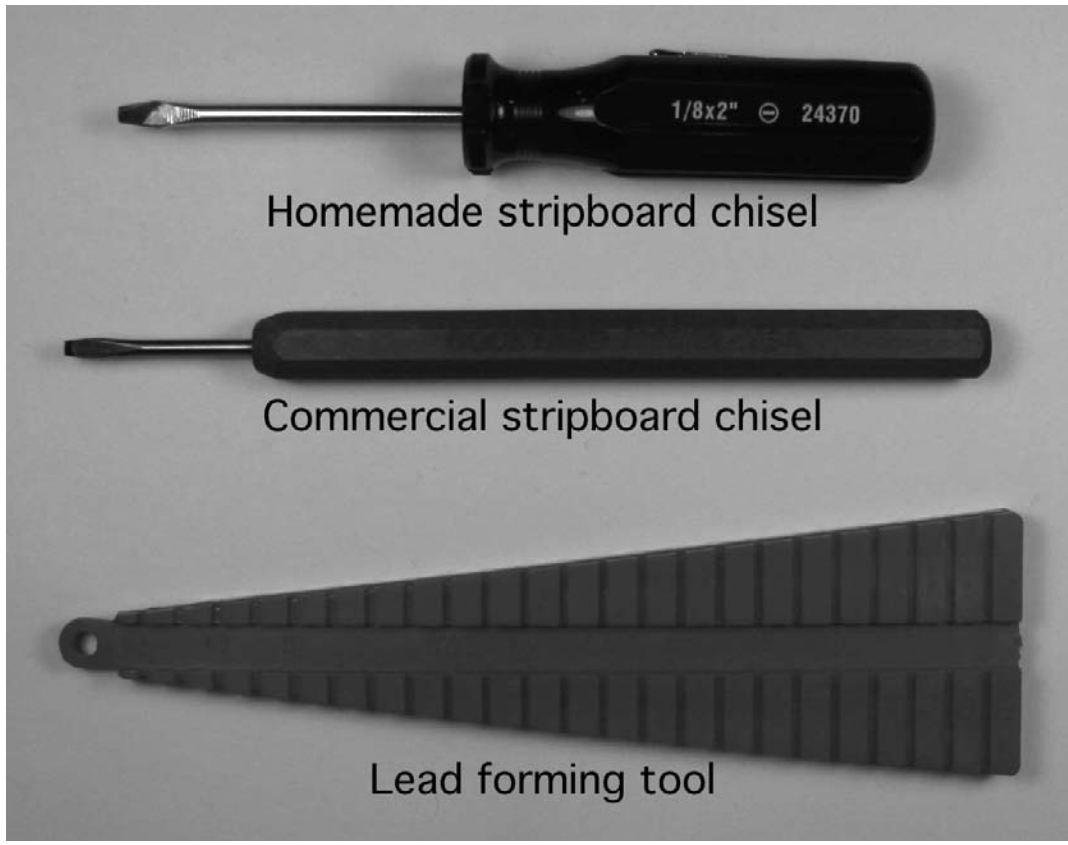


Figure 2-5 More stripboard tools

little chisel makes precise cuts between two adjacent holes. (See the cut between the two holes in the third trace from the left in Figure 2-4). Also, the commercial chisel is made from hardened steel, so it remains sharp much longer than the homemade version. For readers who are interested, this 3-mm chisel is available on my website.

Whether you use a homemade or commercial chisel, cutting a trace between two holes is a simple and safe procedure—just place the chisel midway between the two holes (holding it at a slightly acute angle) and press down into the trace until it is severed. Next, rotate the board 180 degrees and make a second cut very close to the first one. This results in "popping out" a small sliver of copper, which severs the trace at that point. In a way, it's similar to using a hobby knife, but I have found that "pushing down" with a chisel blade is much less error-prone than "pulling across" with a knife blade. Whichever method you decide to use for making cuts, it would probably be a good idea to sacrifice a small piece of stripboard to practice both types of cuts before you actually begin to construct our next project.

At this point, I need to back up briefly and clarify something about Figure 2-4. In it, you can see that three traces have been cut at a hole. The hole that's cut in the far left trace is a good example of a typical cut made with a 1/8-inch drill bit. The cut in the second trace from the left was also made with a drill bit, but then "squared up" with a chisel. I will sometimes do this if I am concerned that a trace may not be completely severed (or if my compulsivity gets the best of me). For example, if you look closely at the cut on the far right, you can see that the trace is not completely severed (a small sliver remains on its right edge). To correct this, it's usually better to use the chisel than it is to go deeper with the drill bit, because that runs the risk of accidentally

cutting into the adjacent trace on the opposite side. In any case, before beginning to solder the board, it's a good idea to use a continuity checker to be sure that each cut completely severs its trace. It's much easier to correct problems before rather than after soldering.

There is one more tool shown in Figure 2-5 that we haven't discussed yet. It's the long triangular piece of plastic at the bottom of the photo that's used to bend component leads in exact multiples of 0.1 inch (2.54 mm) so they fit perfectly into a stripboard. Of course, this tool is not at all necessary; you can certainly do the same thing by hand, but a lead-forming tool is a real convenience and time-saver when working with stripboard circuits. If you are interested in getting one, they're available at Jameco Electronics (www.jameco.com)—just search for "lead forming tool."

Project 2
The USBS-PA3 PICAXE Programming Adapter

The simple mini-stereo jack adapter that we constructed in Chapter 1 is all we really need to connect a breadboard project to the AXE027 programming cable. However, this time around we're going to construct another adapter that will further simplify the interface between your computer and any PICAXE breadboard project. This adapter (which we'll call the USBS-PA3) includes the 10k and 22k resistors required in the standard programming circuit, so we'll be able to use it in all the projects we're going to implement in Part One of the book.

All the parts required for this project are available on my website and are listed in the Parts Bin. Two of the parts require a brief explanation. First, the high-profile mini-stereo jack has the same pin-out as the low-profile version that we used in our first project. The difference is that the high-profile jack sits high enough on the stripboard to allow us to include parts underneath it before soldering the jack in place. This feature will enable us to significantly reduce the size of the stripboard. Also, both resistors need to be the smaller 1/6-watt size to save a little more space.

PARTS BIN
Stripboard, small
Mini-stereo jack, 3.5-mm, high-profile
Resistor, 10k, 1/6 watt
Resistor, 22k, 1/6 watt
Resistor, 100k, 1/4 or 1/6 watt (see text)
Header, male, 10-pin, "reverse-mountable"

Figure 2-6 presents the schematic for the USBS-PA3 adapter, and Figure 2-7 is the stripboard layout, which includes column and row labels analogous to those of a standard spreadsheet. I'll use this labeling arrangement in all our stripboard projects to make it easy to refer to specific locations when necessary; for example, "Next install the 22k resistor between holes D1 and D5." Note that the row labels in Figure 2-7 are reversed for the top and bottom views—this reflects the physical reversal that happens when you flip the board to view the bottom. Of course, there are at least two ways to flip a board (horizontally or vertically). When working on the stripboard, it's important to make sure you always flip the board in the same manner that the layout has been flipped to avoid the possibility of cutting a trace at the wrong spot.

One final point before we actually begin construction: The leads on most of the components that we will be using in our projects (resistors, capacitors, diodes, etc.) are fairly long; when they are snipped off after soldering the component in place, most of them are still about an inch long. These "off-cuts" are worth saving because they can

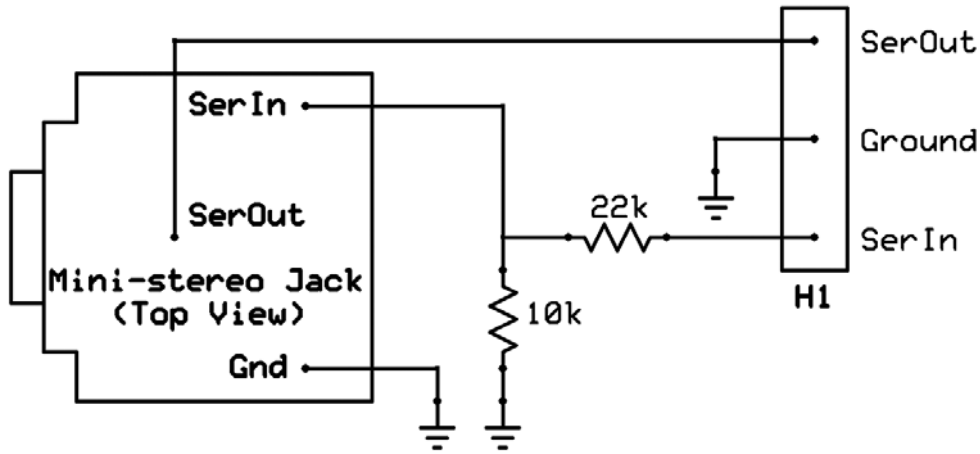


Figure 2-6 USBS-PA3 schematic

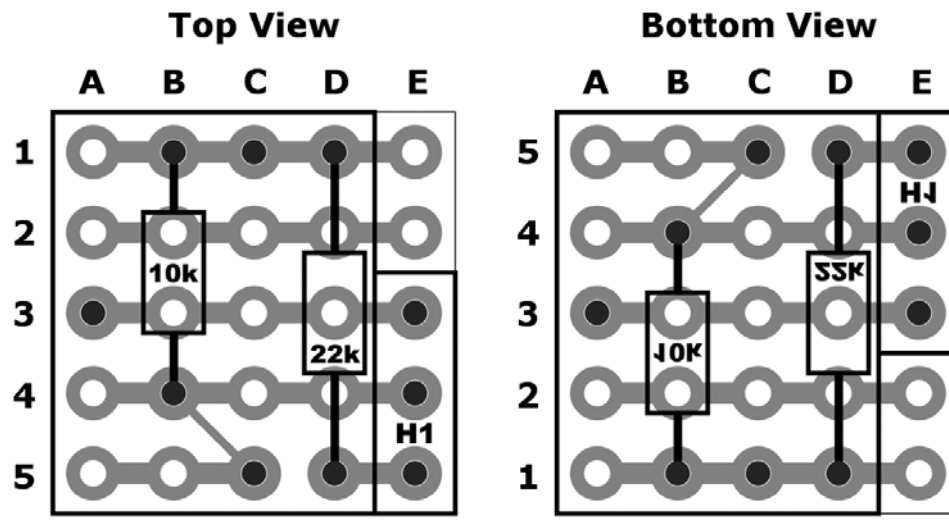


Figure 2-7 Top and bottom views of USBS-PA3 stripboard layout

serve three helpful functions. First, you can use them as short jumper wires. Second, because they are significantly thinner than standard jumper wire, the ends of two of them can fit in the same hole, which is sometimes helpful in a stripboard layout. Third, there are times when it's necessary to solder a jumper between two traces on the bottom of a stripboard, and the thinness of off-cuts makes them ideal for this purpose.

Okay, we're finally ready to construct the USBS-PA3 board. The following list of directions may seem a little long for such a simple project. In subsequent projects, I'll assume you are familiar with some of the details involved and make the instructions mercifully shorter. But for this project, I think the extra detail is helpful. Finally, it's always a good idea to read through the entire project to be sure you fully understand the sequence before actually beginning the assembly procedure.

1. Cut a piece of stripboard to the required size (five tracks of five holes each) and smooth the edges.
2. Use a small pair of diagonal pliers to snip off the pin on the jack that would have been inserted into hole C4 (the one in the middle of the row of three pins).

3. Use a 3/64-inch drill bit (1.5 mm should also work, but I haven't tested that) to widen the holes at A1, A3, A5, C1, C3, and C5 to accommodate the mini-stereo jack pins and plastic supports. If you can't find a suitable drill bit, file the jack's pins slightly until they fit in the holes at A3, C1, and C5, and slice off the small round plastic nipples that would have sat in the holes at A1, A5, and C3.
4. Use a pair of needle-nose pliers to straighten the pins of the jack and test-fit it in the stripboard, but do *not* solder it at this point.
5. Cut the trace between holes C5 and D5.
6. Clean the traces with a Scotch-Brite or similar plastic abrasive pad.
7. Insert the two resistors as indicated. Solder and snip the leads at B1, D1, and D5, but do *not* solder or snip the lead at B4 yet.
8. Snap off a three-pin section of the reverse-mountable male header and insert it from the top of the board in the position indicated in Figure 2-7. Using the same procedure we employed in Project 1, support the stripboard and header on a breadboard while you solder the header in place.
9. Remove the stripboard assembly from the breadboard and snip off the short ends (on the top of the board) of the soldered header. You may also want to file the cut ends of the header pins smooth at this point.
10. Insert the mini-stereo jack into the board so that its three remaining pins are inserted into holes A3, C1, and C5. Make sure that the jack is fully inserted into the board.
11. Flip the board and stereo jack upside-down again and place it on a flat surface. Bend the unsoldered lead from B4 to the stereo jack pin at C5. Snip the lead so that it's just long enough to bend around the pin at C5.
12. Use needle-nose pliers to pinch the lead around the pin at C5, and use a small, flat screwdriver blade to press the lead flat against the stripboard.
13. Solder the pins/leads at holes A3, B4, C1, and C5. Because the adapter is going to be inserted into a breadboard, it's a good idea to snip the protruding pins of the jack as short as possible and file them smooth.
14. Use an old toothbrush and isopropyl alcohol to clean the flux from the bottom of the board, and allow it to dry.
15. Inspect the stripboard carefully for accidental solder connections and other problems.

Hello Again

Figure 2-8 is a photo of the completed USBS-PA3 adapter. In case you decide to paint the tops of the three header-pins, I have again indicated the colors that I used.

Figure 2-9 shows the USBS-PA3 installed on the "Hello World" breadboard circuit that we developed in Chapter 1. If you look closely at the photo, you will see a resistor tying the 08M2's

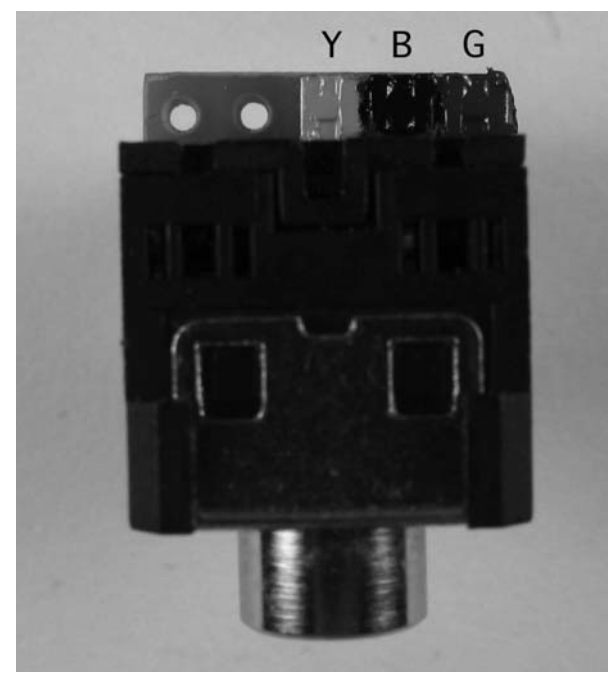


Figure 2-8 Completed USBS-PA3 adapter

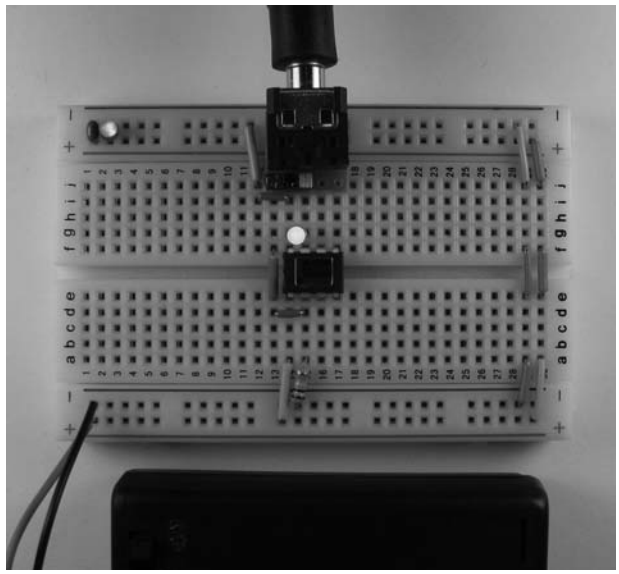


Figure 2-9 USBS-PA3 adapter installed on "Hello World" breadboard

serin pin to ground. What you can't see in the grayscale photo is that its value is 100k. This small addition to the circuit is necessary because of how the Programming Editor initiates a program download to a PICAXE processor, but we need to back up a bit before explaining the details of the download process.

When an input pin is not connected to any part of a circuit, it's said to be "floating," which means that it isn't tied either high or low and the voltage level on the pin can randomly fluctuate, or "float," between high and low values. Theoretically, all unused input pins should always be tied either high or low, but in practice, this rule is often ignored. If you look at our "Hello World" circuit, you can see that input 3 (external pin 4) has been left floating. In spite of that omission, the circuit works fine. However, all inputs that *are* used in a circuit *must* be tied high or low at all times, and the serin input line is always in use in every PICAXE circuit. This is because the processor is continually checking the state of its serin pin (in the background of the running program) to determine whether ProgEdit (or AXEpad) wants to initiate a new program download.

In other words, whenever a program is actively running on a PICAXE chip and you decide to download an edited (or entirely different) program, ProgEdit "interrupts" your running program to initiate the new download. It does this by pulling the serin pin high. Whenever the serin pin goes high (even for 1 microsecond), your program will stop running and the processor will start looking for the new download that it expects to receive. Since a floating pin can easily go high or low randomly, the serin pin must be held low at all times in order for your program to run reliably. That way, the only time the serin pin is high is when ProgEdit has deliberately raised it to initiate a new program download.

If you look back at the USBS-PA3 schematic presented in Figure 2-6, you can see that the 10k and 22k resistors in the standard programming circuit tie the serin pin to Ground. However, the USBS-PA3 is specifically designed to be easily movable from one project to another, and there definitely will be times when we will want to be able to run the program in a project that does not have the USBS-PA3 attached. The 100k resistor that ties the serin line to Ground is what enables us to do this. If we didn't include it and then removed the USBS-PA3, the circuit would function erratically or not at all.

To test this assertion, try the following experiment. Set up the "Hello World" circuit with the USBS-PA3 installed but without the 100k resistor installed. Next, power the circuit; the LED should start blinking. Now, remove the USBS-PA3 from the breadboard two or three times; the LED may or may not stop blinking. If it stops blinking, reinsert the USBS-PA3 and it will start again. If it continues to blink without the USBS-PA3 in the circuit, touch the serin pin with your finger and the blinking will probably stop because your finger has changed the voltage level on the floating serin pin. Next, try the same procedure but this time with the 100k resistor tying the serin pin to

Ground. In this case, the LED should continue blinking when the USBS-PA3 is removed and when you touch the serin pin with a finger. In other words, the serin line must always be tied to Ground in order for a program to function reliably.

At this point, I can almost hear you murmuring, "What's the big deal—just leave the 10k and 22k resistors on the breadboard!" That may be the simpler approach, but it does have two disadvantages. First, it requires more board space—for a breadboard circuit, this doesn't matter very much, but for some very small PC boards, even an extra resistor can use up too much board real estate. More importantly, putting both resistors in every circuit provides just one more opportunity to make a mistake. In my experience, it's better to tie the serin pin to Ground with a 100k resistor and keep the PICAXE programming circuit separate, as we have done with the USBS-PA3. That's the approach I will use throughout this book, but you can certainly modify any or all of the projects to include the programming circuit if you prefer. In Part Two of the book, we'll construct additional programming adapters for specific purposes as the need arises, but the USBS-PA3 adapter is all we need to complete the projects in Part One.

This page intentionally left blank

CHAPTER 3

Designing and Building a +5V Regulated Power Supply

Our simple 4.5V battery-powered supply was a quick and easy way to get started with our "Hello World!" project. In fact, 4.5V is enough to power any current PICAXE processor, as you can see from the PICAXE supply ranges presented in Table 3-1. In addition, the M2-class chips, as well as the 20X2 and the low-voltage (3V) versions of the 28X2 and 40X2, can actually run on a 2-AA supply. However, with a supply voltage that low, the downloading process is not always reliable, so if you design a low-voltage project, you will still need at least a 4.5V supply for downloading your program to the processor.

When designing a project, it's important to check the voltage requirements of each of the components that you may be using. For example, many LCD displays require a +5V supply to function correctly, so a 3-AA supply isn't sufficient. There's another problem with battery-powered projects that also must be taken into consideration: Over time, the output voltage will gradually decrease and then drop rapidly as the battery pack becomes depleted. In projects as simple as "Hello World!" all that will happen is the LED will gradually grow dim and eventually not light at all. However, if a project involves any sort of critical timing functions and/or analog-to-digital conversion (ADC) voltage measurements, the accuracy of the program's computations will suffer as the voltage decreases over time. Because of the problems associated with low-voltage battery supplies, it's a good idea to also have a regulated +5V power supply available for projects that require it. In fact, it's simpler to power all your PICAXE projects with a regulated +5V supply, at least during the development stage, and only switch to battery power in the final version of a project that requires it.

Of course, you could always purchase a commercial +5V power supply for this purpose, but the fact that you're reading this book suggests that you prefer building your own equipment, so that's exactly what we're going to do in this chapter. Also, we're going to use our +5V regulated supply project as a "design study" for the process of using Microsoft Word to develop stripboard layouts.

TABLE 3-1 Supply Range for Selected PICAXE Processors

PICAXE Processors	Supply Range
08M2, 14M2, 18M2, 20M2	1.8V to 5.5V
20X2	1.8V to 5.5V
28X2, 40X2 (3V)	1.8V to 3.6V
28X2, 40X2 (5V)	4.2V to 5.5V

Designing a +5V Regulated Power Supply for Breadboard Circuits

Figure 3-1 presents the schematic for our +5V regulated power supply. The circuit is a typical design that you have probably seen many times before. All of the parts are available on my website. Two of them require a brief explanation: the power connector and the switch.

PARTS BIN
Stripboard, large
Two bypass capacitors, 0.01µF each
Capacitor, electrolytic, 47µF
Capacitor, electrolytic, 100µF
Diode, 1N4001
LED, 3-mm, resistorized
7805 voltage regulator
Power connector
Switch, DPDT
Two headers, male, 10-pin, "reverse-mountable"

There are several different sizes of power connectors on the market. The one I chose mates with a power plug that has a 2.1-mm ID (inner diameter) and a 5.5-mm OD (outer diameter). That size plug is commonly available and easily obtained from surplus vendors or obsolete answering machines, modems, etc. If you decide to use a different size power connector, you may need to modify the layout slightly to accommodate it.

The pins of the switch I chose are spaced on 0.1-inch (2.54-mm) centers, which makes it easy to install in a stripboard circuit, but it's only capable of switching currents up to a maximum of 300mA. The switch's current limitation may seem a little too low, since the 7805 regulator is capable of sourcing a maximum of 1A. However, the 7805 requires a good heat sink to handle currents that large. Without one, it's only capable of managing something in the vicinity of 300mA. Also, more robust switches tend to have pins that are too large and inconveniently spaced to be easily accommodated in a stripboard circuit. Those considerations, as well as the fact that 300mA is more than ample for all the projects we will be constructing, led me to choose the specific switch in the parts list. This switch has two rows of pins that attach it more solidly to the stripboard than the more usual single-row arrangement, but it's still a single-pole, double-throw (SPDT) switch. If you would prefer maximum output from the power supply, I'm sure the layout could be modified to accommodate a heftier switch.

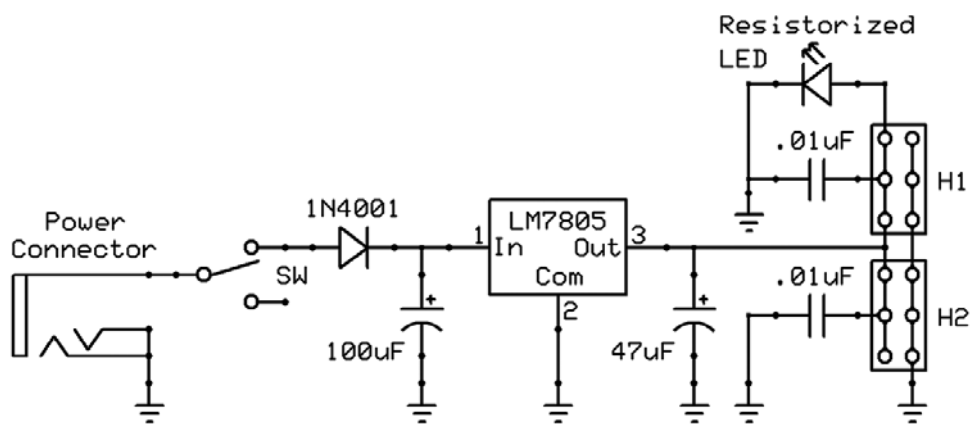


Figure 3-1 Schematic for +5V regulated power supply

Using Templates for Stripboard Layouts

Before we actually construct our circuit, let's focus on the process of using the drawing features of Microsoft Word to develop the stripboard layout. Since we want our power supply to be convenient to insert into a breadboard, it needs to be as wide as a standard breadboard (i.e., 2.0 in., or 5.1 cm). In other words, our circuit will require 20 traces to span the two sets of power rails on a standard breadboard. Also, as we'll soon see, the circuit is going to require nine holes in each trace to accommodate the necessary parts; that's a total of 180 holes. Furthermore, a trace can't be a single object because we need to be able to delete a section between any two holes to indicate that the trace is to be cut between those two holes. (We'll delete a hole to signify a cut *at* the hole.) Consequently, each trace in our layout must be composed of nine holes and eight short segments, for a grand total of 340 objects (holes and trace segments) in our layout—and that's before we even begin to draw the components and connections for our layout!

Needless to say, that's something you don't want to do more than once, so it's a good idea to make a template for a stripboard layout and save it as a separate file that you can open each time you start a new project. (It's also a good idea to make the template a read-only file so you don't accidentally save it as your new file and therefore erase the template.) Actually, you will need to go through this process twice because some projects require that the traces run in the "long" direction on the board. Since a computer monitor is much wider than it is high, it makes sense to draw the traces vertically in a "short-trace" layout and horizontally in a "long-trace" layout. Figure 3-2 shows the Word template that I made for a horizontal layout, and Figure 3-3 is my template for a vertical layout.

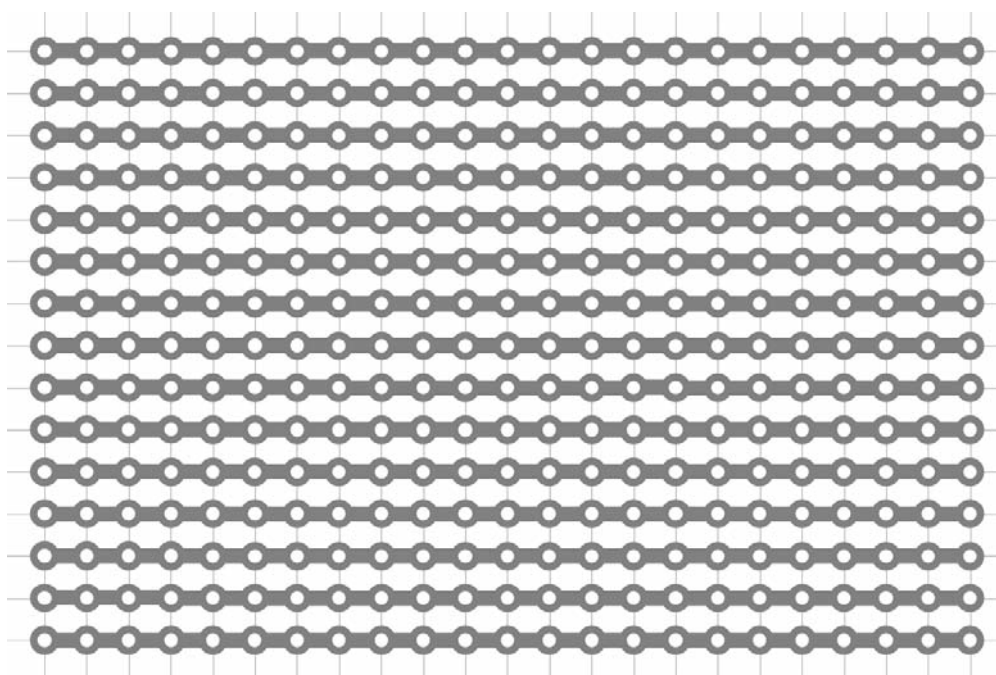


Figure 3-2 Microsoft Word template for a horizontal stripboard layout

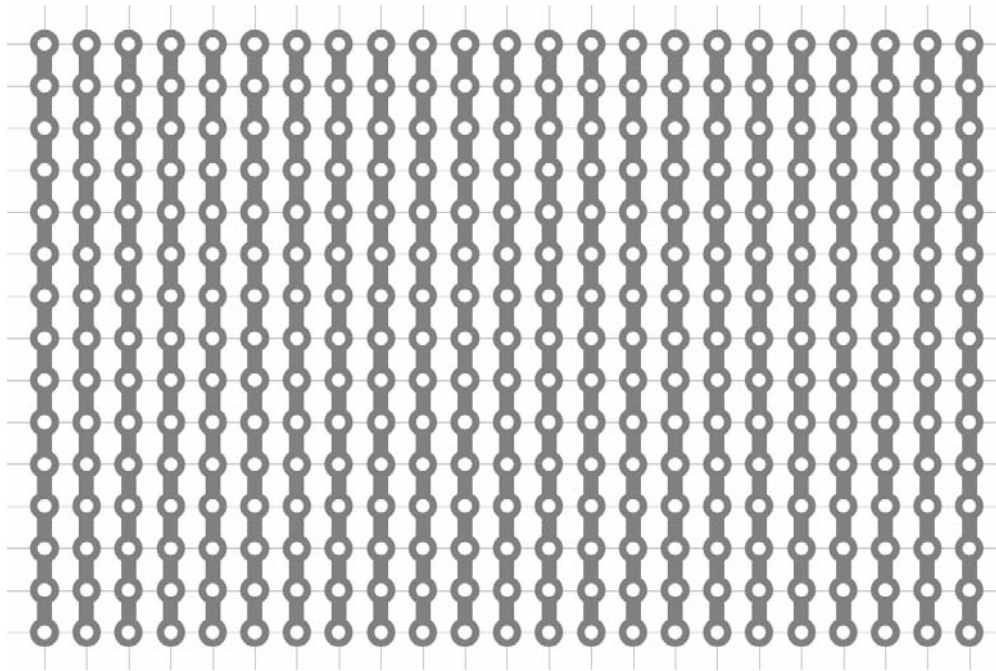


Figure 3-3 Microsoft Word template for a vertical stripboard layout

Let's take a closer look at the vertical template in Figure 3-3, since that's the one we need for our power supply project. To make it, I used the following settings:

- **Page:** landscape orientation
- **Margins:** top and bottom = 1.0"; left = 0.6"; right = 0.8"
- **Grid:** snap = 0.05"; display = 0.2" (multiples of 4)
- **Holes:** circles with 0.2" diameter; 5-points wide; gray color
- **Trace segments:** 0.2" lines; 10-points wide; gray color

To convert the template for our power supply project, I simply opened my vertical template file, deleted all the holes and trace segments that we won't need, and saved the file with a new name. The resulting blank layout is shown in Figure 3-4.

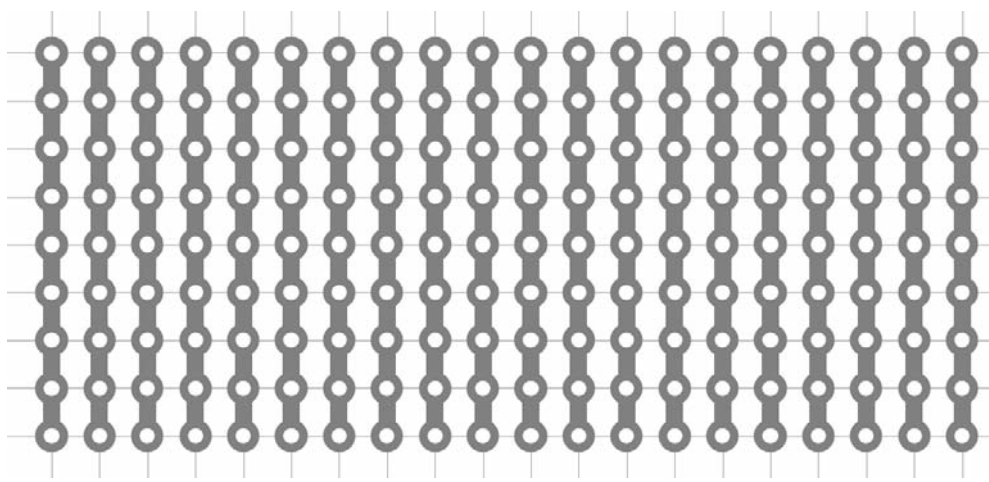


Figure 3-4 Blank Microsoft Word layout for +5V regulated power supply

Designing Components

Figure 3-5 presents the top and bottom views of my completed layout for the project. The first aspect of the layout that we need to discuss is the process of designing a component. As a simple example, let's consider headers H1 and H2. They are going to be reverse-mounted so that they can be inserted into the two sets of power rails on a standard breadboard. Using six pins for each header may seem excessive, but it serves two functions. First, the extra pins will hold the power supply in place more firmly when it's inserted into the breadboard. Second, they make it impossible for the power supply to be incorrectly inserted along one side of the breadboard. If that were possible, a direct short would occur, so this is just a simple safety precaution.

Each header is composed of three parts: the outline, the pins, and the label. The outline can be drawn with either the line tool or the rectangle tool. If you use the rectangle tool, be sure to set its fill color to "no fill" or you won't be able to see what's behind it. To do so, right-click the rectangle and choose Format AutoShape from the context menu. The pins are drawn with the oval tool (holding down SHIFT to make a circle). They are 0.1 inch in diameter, with a dark gray line and fill color. (Use Format AutoShape to set their

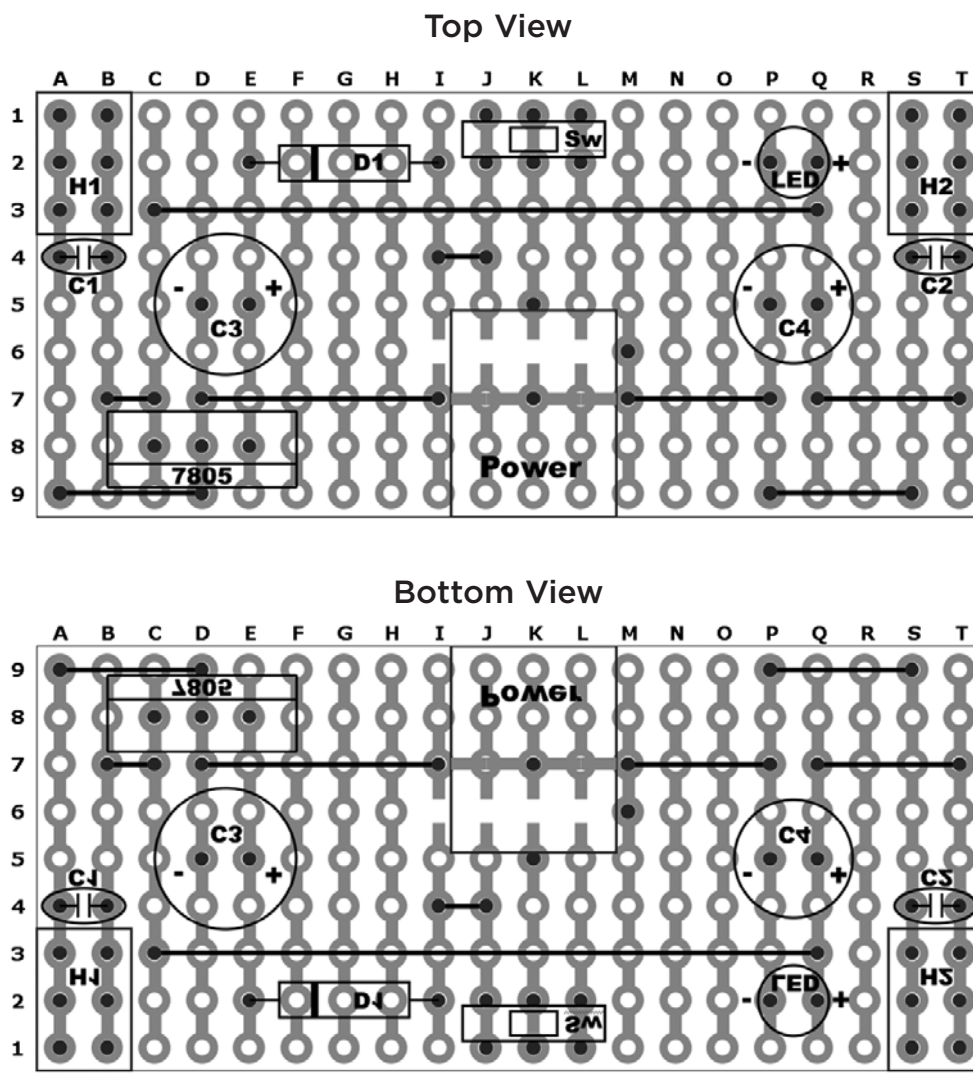


Figure 3-5 Completed stripboard layout for power supply project

properties.) Of course, you only need to make one, then copy and paste as many as you want. The text (which can be any color and size that you want) is drawn with the Text Box tool. In order to place the text precisely where you want it, temporarily turn off the Snap To Grid option. To do so, choose Draw and then Grid in the drawing toolbar, drag the text box to the desired position, and reactivate Snap To Grid. Each of the other components in the layout in Figure 3-5 was created in a similar manner. For example, the two bypass capacitors are each comprised of an oval (with no fill), two circles, four short line segments, and the text label. Finally, I usually draw a rectangle (fine line, no fill) to outline the entire board.

When you have completed designing a component, you can SHIFT-click to select every object in its definition (except the Text label—I'll explain shortly) and then choose the Group option under the Draw icon in the drawing toolbar. That way, the component is a single object that can easily be moved, copied, etc. If you need to modify a component's design, you can always "ungroup" it, make the necessary changes, and then "regroup" it again. The reason for omitting the label from the component's definition is that it makes it easier to copy and paste when multiple instances of the same component are required in a design. If the label were included, you would have to ungroup the component, edit the label, and then regroup it each time—it's much easier to simply add the label separately to each component.

Finally, whenever you are designing a new component, make sure you know its physical measurements so that the layout accurately reflects the amount of space required by the component. When you are actually soldering the parts in place, there's nothing more frustrating than discovering that there isn't enough room to insert a component. Also, you may want to make a separate Component file into which you can paste a copy of each component that you design. That way, when you begin a new stripboard layout, you can open your Component file and simply copy and paste each component that you need in the new layout.

Creating the Bottom View of a Stripboard Layout

When I first experimented with using Word for designing stripboard layouts, one of the features that I really appreciated was that once a layout was finished, all you had to do was select all of the objects in the layout (using the Select Objects pointer in the drawing toolbar and dragging across all the objects, not Select All in the Edit menu—that doesn't work), group them, and then simply flip the layout to get a bottom view. Unfortunately, my early experiments didn't include text labels for component identification. Once I started including them, the Flip option was grayed out and no longer available. (I'm using Microsoft Word X for Macintosh; if you have a different version, you may find that you can flip the layout using this approach.)

However, there are at least two other reasonably easy ways to flip a layout to obtain the bottom view. In Word, you can use the following sequence of steps:

1. Group all the objects in your layout (as described previously).
2. Make a mental note of the position of the top and bottom edges of the layout on the grid.
3. Grab the middle handle at the top of the layout.
4. Drag it below the bottom of the layout and release the mouse button.
5. Grab the middle handle at the bottom of the layout (which is now at the top!).
6. Drag it up to where the top was at the beginning of this procedure.
7. Adjust the (new) bottom to its original position.

Interestingly, in the bottom view that results from this procedure, the text labels move appropriately, but they are not inverted (i.e., they

are still readable). You may want to save the file with a new name so that you have two printable files: top view and bottom view.

The second method of generating a bottom view of the layout is to capture the window (using Windows Print Screen or Macintosh Grab command) that contains the top view, and then open the resulting image in a graphics program (such as Windows Paint or Macintosh Preview) and flip it. (You may also want to crop the image to remove any extraneous portions of the window.) Again, you may want to save two different files, one for each view.

Labeling the Rows and Columns of the Layout

Whichever method you use to generate the bottom view of the layout, you will need to label the rows and columns. The simplest approach is to handwrite the row and column labels on each view. (Of course, it's important to remember to reverse the order of the row headings!) If you would prefer to have computer-generated labels similar to the figures in this book, you can use Microsoft Excel to accomplish the task. If you used Word to flip the layout, you can copy the resulting object from the Word document and then paste it into a new Excel spreadsheet. If you used the graphics approach, you can insert the cropped graphic into the Excel spreadsheet. Either way, you will need to adjust the column width and row height until it matches the layout. Once you have done that, you can simply type in the appropriate row and column headings in the cells that are adjacent to the layout. (Again, don't forget to reverse the row headings for the bottom view!) That's the approach I used to generate all the layouts in the book.

I can imagine that the entire process of generating a stripboard layout must sound complicated when you first read through the details, but after you have done it a few times, it really is a reasonably quick and easy process. If you build up your own library of components as I described earlier, it primarily becomes a simple matter of copying and pasting, and then drawing in the appropriate jumper connections. The best part is the ease with which you can modify the layout as you work on the design—no more holes in the graph paper from excessive erasures and redraws! When the next "great idea" for a stripboard project occurs to you, give Microsoft Word a try. In the meantime, we're finally ready to construct our +5V regulated power supply.

Project 3
More Power, Scotty!

Before you actually fire up your soldering iron, I want to mention a little detail about jumpers. The insulation on jumper wires (especially the short ones) tends to melt a bit when the jumper is soldered in place. Also, accurately measuring and stripping the ends of every jumper wire tends to be the most tedious aspect of stripboard circuit construction. To avoid both of these problems, I have developed the habit of using bare jumper wire for all my stripboard projects. (Fortunately, I happen to have a large roll of bare wire that I found on a surplus site several years ago, which makes things even easier.) Even if you don't have access to bare wire, stripping the insulation from a four- or five-inch piece of wire is relatively easy, and then you can cut a few jumpers from one piece of stripped wire. The only time you may want to include an insulated jumper wire in a stripboard circuit is when you think there's a risk that a bare wire might accidentally contact another connection on the board.

The following list of directions may seem a little daunting, but I think more detail is better than less, at least at first. Read through the entire list before you begin construction, and have fun!

1. Cut a stripboard to the required size (20 traces of 9 holes each), and smooth the edges.

2. Use a stripboard tool to cut the traces at the following holes: I6, J6, K6, and L6.
3. Make sure that the pins of the power connector will fit easily into the stripboard, either by using a 3/64-inch drill bit to slightly enlarge the holes at K5, K7, and M6 or by filing the connector's pins slightly until they fit.
4. Set the power connector aside and use an abrasive plastic pad to clean all traces on the bottom of the board.
5. Insert the following bare jumper wires on top of the board: C3-Q3, I4-J4, B7-C7, Q7-T7, A9-D9, and P9-S9. Solder and snip the leads.
6. Insert jumpers D7-I7 and M7-P7. For the M7-P7 jumper, make sure there's sufficient lead length at M7 to extend on the bottom of the board to I7 as shown on the layout. Solder and snip leads at D7 and P7, but not yet at I7 and M7.
7. On the bottom of the board, bend the jumper from M7 to I7 and snip it so that it's just long enough to touch the lead at I7. Press it flat against the bottom of the board and hold it there with a small spring clamp as shown in Figure 3-6.
8. Solder the jumper leads at I7 and M7, and snip the excess lead at I7.
9. Insert bypass capacitors C1 and C2; solder and snip the leads.
10. Insert diode D1. (Observe correct polarity.) Solder and snip the leads.
11. Insert the LED. (Observe correct polarity.) Solder and snip the leads.
12. Insert and solder the switch in place (in either orientation).
13. Insert the power connector on the top of the board, making sure it's fully seated. Invert the board and solder the pins at K5, K7, and M6. At K7, make sure that the jumper wire is also soldered to the connector pin.

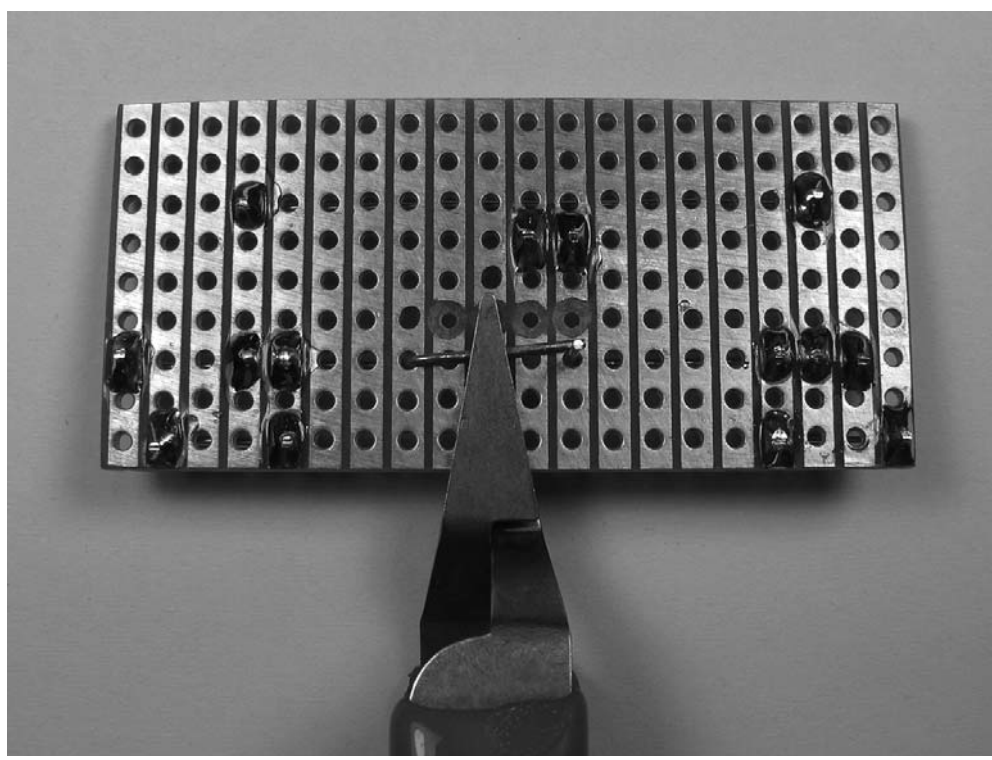


Figure 3-6 A powered-up Arduino board with LED lit.

14. Insert capacitors C3 and C4 as indicated on the layout. (Observe correct polarity.) Solder and snip the leads at D5, E5, P5, and Q5.
15. Insert the 7805 regulator. Make sure its orientation matches the shape on the stripboard layout. (The thinner portion containing the "7805" label is the metal tab.) Solder and snip the leads.
16. Sand or file the bottom of the board to remove all sharp protrusions.
17. Reverse-mount headers H1 and H2; invert the board and support it with female headers appropriately spaced on two or three breadboards as shown in Figure 3-7.
18. Solder the header pins on the bottom of the board.
19. Use isopropyl alcohol or paint thinner and an old toothbrush to clean the excess flux from the bottom of the board.
20. Inspect the board carefully for accidental solder connections and other potential problems.
21. Allow the board to dry thoroughly before testing.

Figure 3-8 is a close-up of the bottom of the completed power supply circuit, and Figure 3-9 shows it installed on a breadboard, ready for testing. The three header pins on each side of the board that appear gray in the photo have been painted red to make sure that I insert the supply into the breadboard power rails in the correct orientation. The two LEDs that are inserted into the two power rails are, of course, resistorized. Since they are the smaller 3-mm variety, there's no flat section in the plastic to identify the cathode, so if you cut the leads, be sure to make the cathode lead slightly shorter for easy identification. When you test your power supply, also measure the voltage on both positive rails to make sure that they're close to +5.0V (±0.2V should be fine).

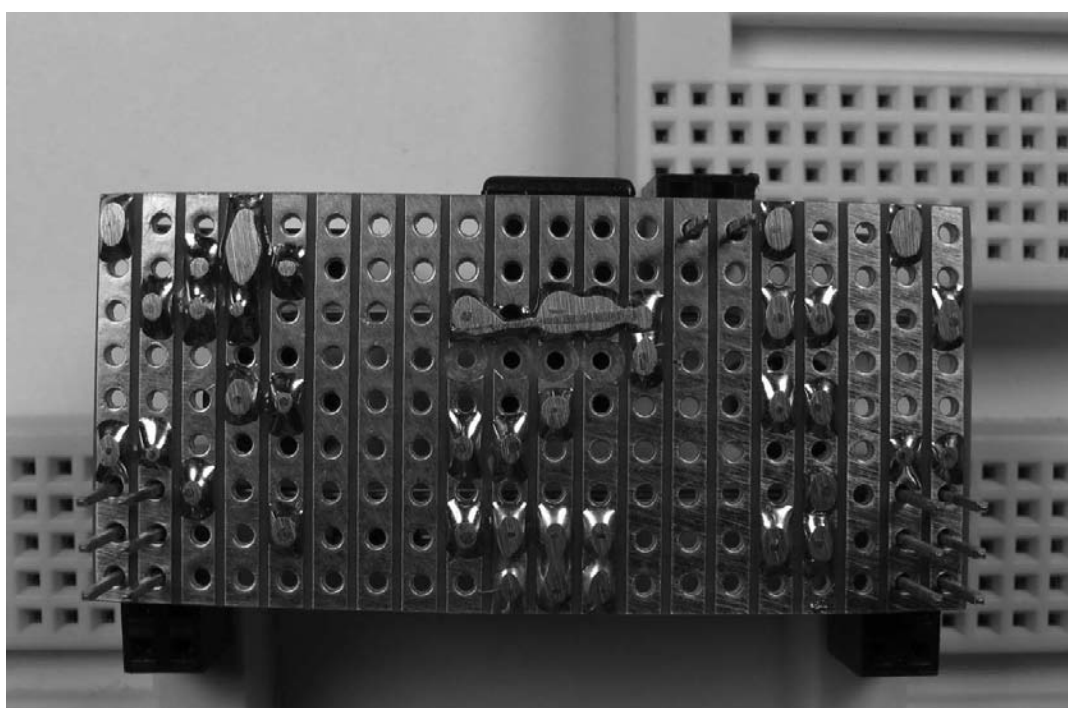


Figure 3-7 Reverse-mounted headers prepared for soldering

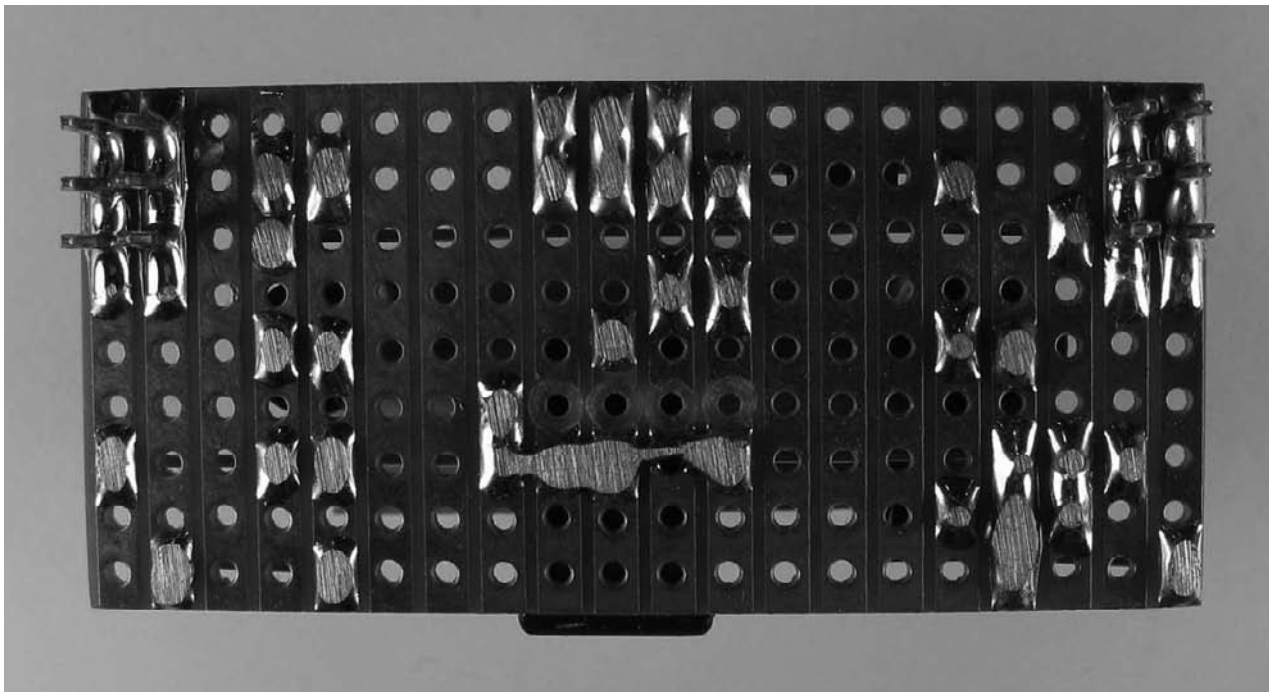


Figure 3-8 Close-up of bottom of completed power supply circuit

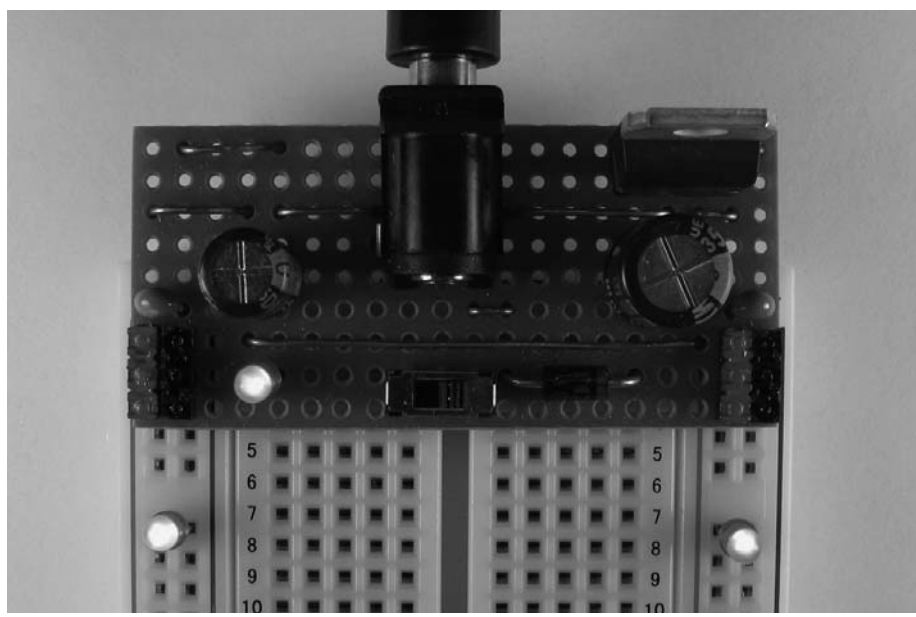


Figure 3-9 Completed power supply being tested on breadboard

Hello and Good-bye!

Figure 3-10 again presents our "Hello World!" project—I promise this is the last time you will see it! In the photo, you can see the arrangement we will be using for the next few projects, including our regulated supply and the USBS-PA3 programming adapter.

CAUTION **If you plan to move the adapter from project to project, don't forget to include the 100k resistor that ties the serout pin to Ground.**

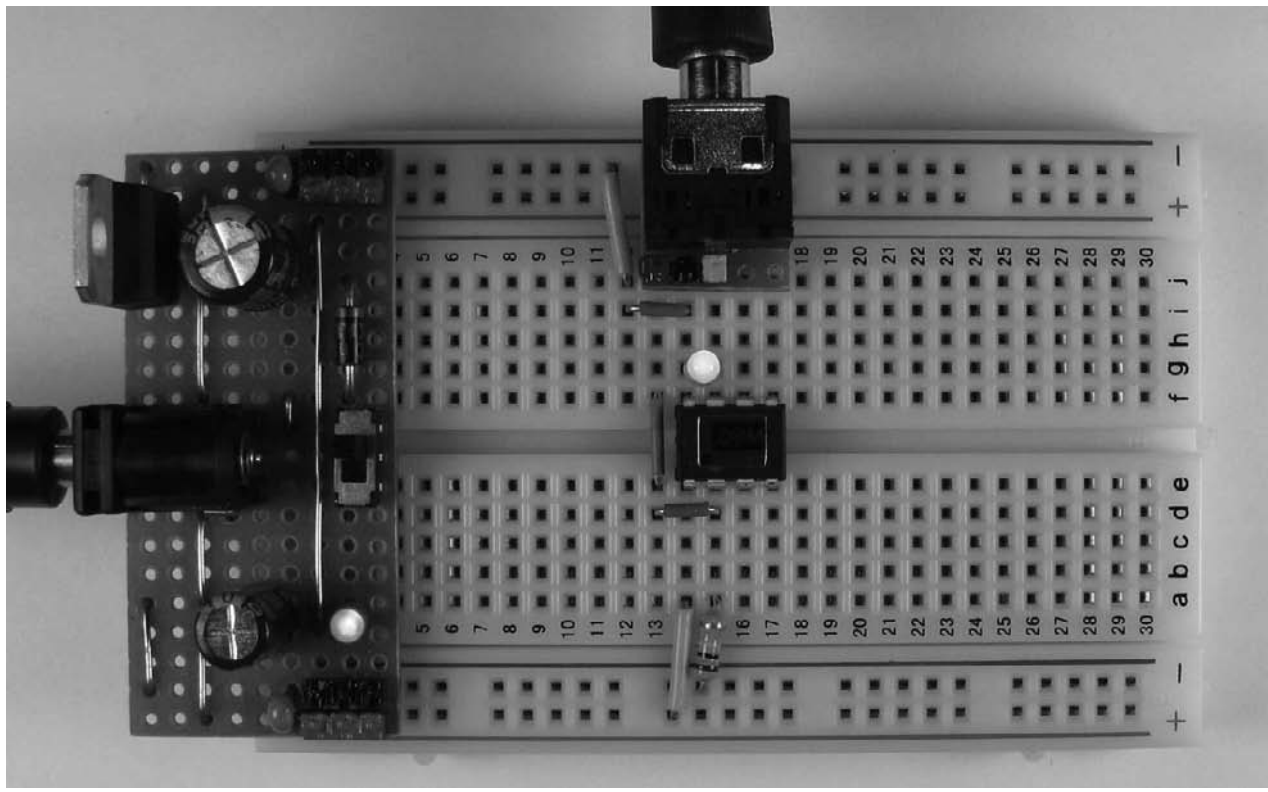


Figure 3-10 Power supply and USBS-PA3 installed on “Hello World!” breadboard

This page intentionally left blank

CHAPTER 4

Hardware Overview of the PICAXE M2-Class Processors

Revolution Education's recently announced "M2" processors (the 08M2, 14M2, 18M2, and 20M2) include an impressive array of enhancements and innovations to their Educational line of microcontroller products. The pin-out for each of the new processors is shown in Figures 4-1 through 4-4, and the following list presents a brief summary of the most important new or improved features that are available on each of the PICAXE M2 processors:

- **Program memory and variable storage:** Much larger than older "M" processors.
- **Operating speed:** Maximum resonator speed increased to 32MHz.
- **Supply voltage:** All M2 processors can operate on voltages as low as 1.8V; a 2-AA or 2-AAA battery pack is all that's required.
- **Flexible I/O pins:** Almost all I/O pins can now be individually configured as inputs or outputs.
- **Parallel processing:** All M2 processors are capable of executing four separate programming tasks in parallel.
- **ADC inputs:** As many as 11 on a processor.
- **Touch sensors:** New capacitive touch sensor inputs provide a one-wire interface to a simple copper sensor.
- **DAC output:** New digital-to-analog-converter output pin on each processor.

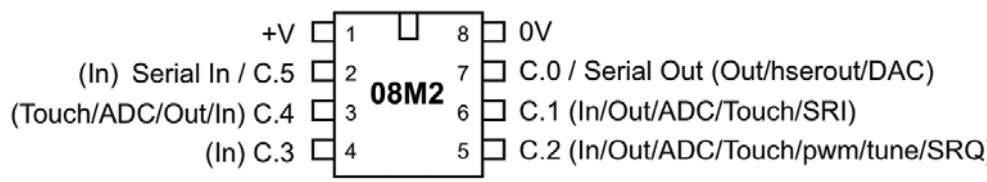


Figure 4-1 PICAXE-08M2 pin-out

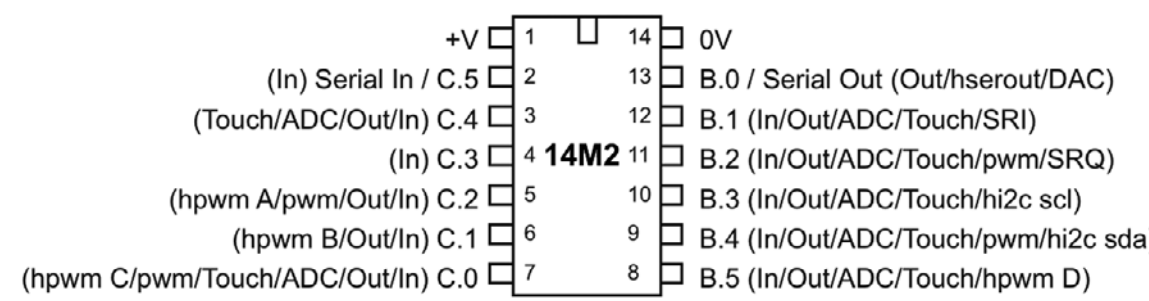


Figure 4-2 PICAXE-14M2 pin-out

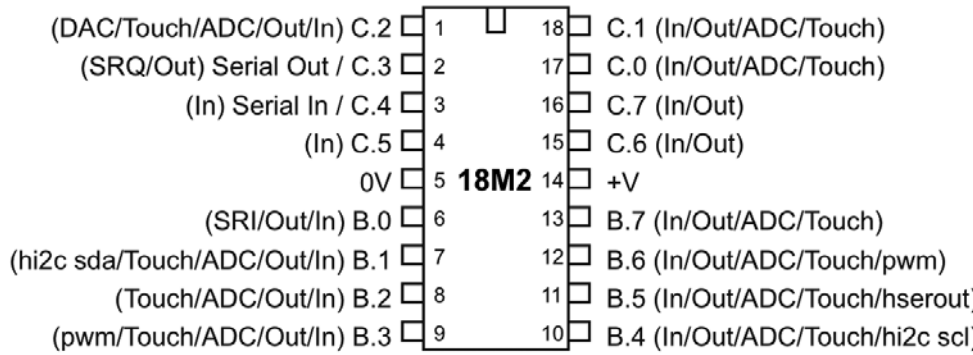


Figure 4-3 PICAXE-18M2 pin-out

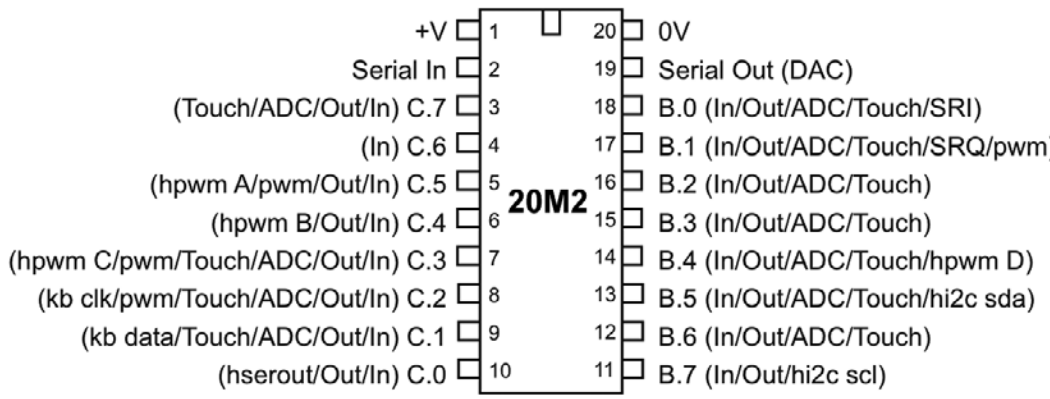


Figure 4-4 PICAXE-20M2 pin-out

- **Internal hardware set-reset (SR) latch:** Able to detect rapidly changing inputs.
- **Built-in "time" variable:** Provides access to an elapsed seconds counter.

From a hardware point of view, the four processors that comprise RevEd's M2 class are similar. All the M2 processors have 28 general-purpose variables, and they are also all capable of running at 4, 8, 16, or 32MHz. The big hardware difference, of course, is in the number of I/O pins that each processor provides: 08M2 = 6, 14M2 = 12, 18M2 = 16, and 20M2 = 18. Another distinction is that the 08M2 can store approximately 200 lines of BASIC code and 48 storage variables, while the three larger processors are each capable of storing approximately 600 lines of BASIC code and 256-storage variables.

In this chapter, we're going to take a detailed look at some of the M2-class capabilities. By the time we're finished, we will have covered the following topics in detail:

- General-purpose variables
- Storage variables
- Special-function variables

General-Purpose Variables

All M2-class processors have 28 bytes of RAM dedicated to the storage of general-purpose variables that are labeled from b0 to b27, organized as shown in Table 4-1.

TABLE 4-1 General-Purpose Variables of the M2-Class Processors

Word Variables	Byte Variables
w0	b1: b0
w1	b3: b2
w2	b5: b4
w3	b7: b6
. . .	. . .
w13	b27:b26

NOTE **As you may know, a *byte* consists of 8 bits, while a *word* consists of 16 bits; therefore, a *word* is composed of 2 bytes (an upper byte and the lower byte).**

As you can see, each pair of byte variables can also be used as one word variable. If you declare a word variable (e.g., *symbol myWord = w0*), make sure you don't also define a byte variable using either one of the two associated bytes (e.g., *myGoof = b1*). The compiler won't flag this as an error, but your program is bound to behave unpredictably, because every time you change the value of w0 you are also accidentally changing the value of b1, and vice versa.

In addition to the word/byte arrangement for the 28 general-purpose variables, there are 32 individually accessible bit variables labeled bit0, bit1, bit2, etc. These variables are actually subdivisions of b0 through b3, as follows:

- b0 = bit7:bit6:bit5:bit4:bit3:bit2:bit1:bit0
- b1 = bit15:bit14:bit13:bit12:bit11:bit10:bit9:bit8
- b2 = bit23:bit22:bit21:bit20:bit19:bit18:bit17:bit16
- b3 = bit31:bit30:bit29:bit28:bit27:bit26:bit25:bit24

Using bit variables in your program can be a powerful technique, but, as mentioned, if you do this, it's important to remember that the associated byte and word variables should not also be used. For example, if you declare "*symbol myBit = bit10*" then don't use either w0 or b1 in another variable declaration; if you do, your program may behave erratically.

Storage Variables

All M2-class processors contain an area of memory that can be used for the temporary storage of data, including the value of a variable. On the 08M2, this area is composed of 48 bytes; on the three larger M2 processors, it contains 256 bytes. On all four M2 processors, the first 28 bytes (0–27) are used to store the 28 general-purpose variables. Therefore, it's important to not use a location that has already been assigned to a general-purpose variable. Except for the general-purpose variables, these locations cannot be declared with the "symbol" instruction and the data they store cannot be used in mathematical calculations; however, the storage variables are still a useful feature. For example, suppose you are developing a complex program in which you have already used all 28 of the general-purpose variables and now find that you need just one more variable to make it all work. You could temporarily put the contents of variable b0, for example, into storage, use the b0 variable to accomplish the task at hand, and then retrieve b0's original contents and move on to the next task. Another important use for this area of memory is to store simple strings or data arrays that need to be easily accessed. We'll discuss this when we look at the M2 special-function variables.

There are two different methods of storing and accessing data in the memory storage area. The simpler method is to use the *poke* and *peek* commands. The syntax for the basic version of the *poke* command is *poke location, data* (see the manual for the more advanced versions), where *location* is the address of the storage area location

into which you want to place the data byte referenced by *data*. Assuming you want to avoid the locations occupied by the general-purpose variables, the available locations would be 28 to 47, for the 08M2, or 28 to 255 for the other M2 processors. For example, if you wanted to temporarily store the current value of b0 in storage location 28 of any M2 chip so that you could use that variable for another purpose, you could write *poke 28, b0*.

The basic syntax for the *peek* command is similar: *peek location, variable*. *Location* has the same meaning as it does in the *poke* command, and *variable* is the variable into which you want to place the data byte that's stored at *location*. (Again, there are more advanced forms of the *peek* command; see the manual for details.) For example, if you now want to retrieve the value of b0 that you had earlier placed in storage location 28, you could write *peek 28, b0*.

The second way of accessing data in the storage locations is more complicated. In order to understand it, we need to first discuss the special-function variables available on the M2 processors.

Special-Function Variables

In addition to the general-purpose and storage variables, there are 11 special-function variables available for use with M2-class chips, as shown in Table 4-2.

TABLE 4-2 Special-Function Variables of the M2-Class Processors

Variable	Function
time	real-time elapsed seconds
***dirsB**	portB data direction register
dirsC	portC data direction register
***pinsB**	portB input pins
pinsC	portC input pins
***outpinsB**	portB output pins
outpinsC	portC output pins
bptr	byte scratchpad pointer
@bptr	byte scratchpad value pointed to by bptr
@bptrinc	byte scratchpad value pointed to by bptr (post increment)
@bptrdec	byte scratchpad value pointed to by bptr (post decrement)

* Not available on 08M2

time

Let's begin with the newest special-function variable, which was recently introduced on all the M2-class processors. *Time* is simply an elapsed seconds counter that is continually updated in the background while your program is executing. It's a word variable, so it can count from 0 to 65535, at which point it rolls over to 0 again. If your program is running continuously for much more than 18 hours (64,800 seconds), you'll need to take that into account. If you want to implement a real-time clock, all you need to do is define variables for *minutes, seconds, hours*, etc., check the value of time periodically, and update your variables whenever necessary. I thought about including a real-time clock project in the book, but it's really so easy on an M2 processor that I decided not to. Instead, in Chapter 13 we'll develop a countdown timer on the 20X2 processor, which doesn't include the built-in *time* variable so it's a little more of a challenging project.

dirsB and dirsC

Before we get into the details of the two *dirs* variables, I want mention a powerful aspect of all PICAXE processors that can sometimes be a little confusing. The directionality of some I/O pins is fixed (i.e., some pins are either inputs or outputs and that can't be changed); other pins are bidirectional (but they do have a "default"

direction that they assume whenever the processor is powered on). In addition, many pins can implement more than one function (but not at the same time). With this in mind, consider the PICAXE-08M2 pin-out presented earlier in Figure 4-1. The C.1, C.2, and C.4 pins are bidirectional, as indicated by the "In/Out" in their descriptions. The C.0 pin is fixed as an output, and the C.3 and C.5 pins are fixed as inputs. In addition, most of the I/O pins are multifunctional, as shown in their descriptions. The function that's listed closest to its corresponding pin indicates both the default direction and the default function for that pin. For example, on power-up, C.0 is configured as the Serial Out pin, C.5 is configured as the Serial In pin, and the remaining four I/O pins are configured as general-purpose inputs.

There are two ways your program can change the default direction of a bidirectional pin. The simpler approach is to just use a command that indicates the direction you want to use. For example, if you connect an LED and current-limiting resistor from the 08M2's C.1 pin to Ground and execute a *high* C.1 command in your program, that pin immediately becomes an output and the LED is lit. The second method of changing the default direction of a bidirectional pin is by using the appropriate special-function variable: *dirsB* or *dirsC*. This approach is more powerful than the first one because it enables us to change the directionality of multiple pins with just one command. Since all the 08M2's I/0 pins are portC pins, only *dirsC, pinsC*, and *outpinsC* are implemented on the 08M2, but the following discussion also applies to *dirsB, pinsB*, and *outpinsB* on the larger M2 processors.

DirsC is an eight-bit variable, but the only bits of *dirsC* that are enabled are the ones that correspond to the processor's I/O pins that are bidirectional. Therefore, for the 08M2, *dirsC* consists of the following eight bits: x : x : x : dir4 : x : dir2 : dir1 : x. Placing a 1 in the appropriate bit position configures the corresponding I/O pin as an output; a 0 makes it an input. If you place either a 1 or a 0 in any of the bits designated by x, nothing happens because that bit is not implemented on the 08M2. In PICAXE BASIC, the % symbol is used to indicate a binary number. Therefore, on the 08M2, the statement *dirsC = %00001111* would configure I/O pin 4 as an input and I/O pins 2 and 1 as outputs (C.0 is fixed as an output anyway). Since only the bidirectional pins are affected, the statement *dirsC = %11100110* would have the same effect.

pinsB and pinsC

The *pinsB* and *pinsC* special-function variables provide access to the real-time values of each of the processor's input pins. Similarly to *dirsB* and *dirsC*, the *pinsB* and *pinsC* variables are composed of eight individual bit variables, which correspond to the eight (possible) pins on an I/O port. However, the only bits of *pinsB* and *pinsC* that are actually implemented are the ones that correspond to a valid input pin on the processor you are using. For example, if you look at the pin-out of the PICAXE-14M2 presented earlier in Figure 4-2, you can see that there are only six pins in each of its two I/O ports.

Let's assume that your 14M2 program has just executed a *dirsB = %00110011* statement. At this point, B.3 and B.2 are configured as input pins, and B.5, B.4, B.1, and B.0 are output pins. (B.0 is fixed as an output anyway, so it really doesn't matter whether you put a 0 or a 1 in the *bit0* position of *dirsB*, but I think it's a good idea to always match the bit and the direction of every pin.) With this configuration of the portB pins, *pinsB* now consists of the following eight bits: x : x : x : x : pinB.3 : pinB.*2* : x : x, so *pinB.3* and *pinB*.2 are the only valid input pin variables. You can use these variables as you would any variable (e.g., if *pinB.3 = 1 then*….), and you can even include a *symbol* statement in your variables declarations to make it easier to remember what's

connected to each input pin (e.g., *symbol pushBtn = pinB.3*); in that case, you can also write *if pushBtn = 1 then…*, etc. There are also other approaches to accessing the valid input data; refer to the "Variables – Special Purpose" section in the manual.

Finally, it's worth pointing out that it wouldn't make any sense to refer to an input pin on the left side of an assignment statement. The value of an input pin is determined by the circuitry to which it is connected (i.e., an input pin is pulled either high or low by some external wiring) so we can't assign a value to an input pin; we can only read the value that it already has.

outpinsB and outpinsC

OutpinsB and *outpinsC* have a similar structure to that of *pinsB* and *pinsC* in that they are each composed of eight separate pin variables—for example, *outpinsB = outpinB.7 : outpinB.6 : outpinB.5…*, etc.; also, only valid output pin variables are actually implemented. Both the *outpins* variables can be used in two different ways. To write to the outputs, simply place the *outpins* variable on the left side of an assignment statement. For example, *outpinsB = b0* will transfer the bit values of variable b0 to the output pins. It's important to remember that only valid output pins will be affected.

For example, consider the pin-out of the PICAXE-20M2 processor shown earlier in Figure 4-4. Suppose there are eight LEDs with current-limiting resistors connected to portB and the following code snippet has just been executed:

```
dirsB = %11110000
outpinsB = %11111111
```

The result is that only the LEDs connected to outputs B.7 through B.4 will be lit. The other four pins have been defined as inputs, so they will be unaffected by the *outpinsB* statement. Of course, you can also write to individual outputs (e.g., high B.7), which is often easier than using this function of an *outpins* variable.

The second use of the two *outpins* variables is to read the current value of the valid output pins. In this case, the *outpins* variable is placed on the right side of the assignment statement. Therefore, executing *b0 = outpinsB* will transfer the current pin values of the portB outputs to the variable *b0*. If you use this approach, it's again important to extract the valid pin data from the extraneous data—see the manual for details. However, you can also access the level of individual output pins by simply using the valid bit variables of the *outpins* variable (e.g., *if outpinB.6 = 1 then…*), which is frequently all that's needed. In the project for this chapter, we'll actually connect eight LEDs to portB of the 20M2, so at that point you may want to experiment with *outpinsB* and observe the results.

bptr, @bptr, @bptrinc, and @bptrdec

I have saved the most powerful (and most complicated) special-function variables until last. Earlier in this chapter (in the "Storage Variables" section) we discussed the use of the *peek* and *poke* commands to access the data in the memory storage area. That approach is known as *direct addressing* because we are directly accessing the data that is stored at a specific location. There is a second approach to data access, known as *indirect addressing*, which uses a data structure known as a *pointer* because it "points to" the data in which we are actually interested. At first glance, this may seem unnecessarily complicated, but it provides a much more flexible and powerful method of data storage and retrieval.

In Chapter 10 we're going to use indirect addressing to implement a serial LCD display; for

now, however, a simple example should help to clarify the concept of indirect addressing. Suppose your project needs to receive an eight-byte string of serial data from a remote processor and then send it on to the terminal window for display. One approach (which would involve direct addressing) would be to use a statement such as "*serrxd b0, b1, b2, b3, b4, b5, b6, b7*" to store the incoming data in eight variables, followed by a similar *sertxd* statement to send the data on to the terminal window. However, that's a fairly wasteful use of general-purpose variables; in addition, if you had to deal with even longer strings of data, you could easily run out of general-purpose variables for use in your program.

To accomplish the same task via indirect addressing, you could use the pointer variable *bptr* and the virtual variable @*bptr*. The word *virtual* is used to indicate that @*bptr* does not refer to a fixed location in memory. Instead, @*bptr* contains the value that is stored in the location "pointed to" by *bptr*. In other words, whenever the value of *bptr* changes, the value of @*bptr* also changes because it now refers to a different location in memory.

To further increase the power (and complication!) of indirect addressing, the remaining two special-function variables can also be used. Whenever @*bptrinc* is used in a command, the value of the *bptr* pointer is automatically post-incremented (increased by 1 after the data has been accessed); whenever @*bptrdec* is used, the value of the *bptr* pointer is automatically post-decremented by 1.

I realize that this is fairly complicated, so let's return to our example to clarify how it all works. Suppose you decide to store the eight incoming bytes in locations 31 through 38 in the memory storage area. In that case, the following code snippet accomplishes the entire task:

```
bptr = 31
serrxd @bptrinc,@bptrinc,@bptrinc,
    @bptrinc,@bptrinc,@bptrinc,@bptrinc,
    @bptrinc
bptr = 31
sertxd (@bptrinc,@bptrinc,@bptrinc,
    @bptrinc,@bptrinc,@bptrinc,@bptrinc,
    @bptrinc)
```

In the *serrxd* statement, the first time @*bptrinc* is accessed, it stores the incoming byte in memory location 31. At that point, *bptr* is automatically incremented, so the next time @*bptrinc* is accessed it stores the incoming byte in memory location 32, and so on. The *sertxd* statement works in exactly the same way because *bptr* has been reinitialized to 31. The eight bytes have been sequentially stored and then retransmitted in the same order.

By this time, I would imagine that you have had more than enough theory for a while, so let's actually build something!

Project 4
Cylon Eye

If you're a science fiction fan (and old enough!), you probably remember the 1978 television series *Battlestar Galactica*. Although the show only lasted one season, it did introduce a fairly impressive race of robots called Cylons. Their most memorable feature was a single "eye" that consisted of a sinister red light that oscillated back and forth as a Cylon scanned its environment. Ever since the original appearance of the Cylons, no self-respecting microcontroller book would fail to include a "Cylon Eye" project, usually as a fancy version of "Hello World." In Project 4, we're going to continue the tradition and develop our own Cylon Eye. Naturally, we'll also use our regulated power supply and our USBS-PA3 programming adapter, as well as a small stripboard circuit that will further simplify the breadboard setup.

Cylon Eye Hardware

The schematic for our Cylon Eye project is presented in Figure 4-5, and the necessary parts are listed. The project requires eight LEDs but to simplify our breadboard setup, we're going to use a ten-LED bar display and a 470Ω nine-resistor single in-line package (SIP) network (both available on my website). Before we actually set up our breadboard circuit, we're going to construct a small stripboard circuit for the display portion of the project. This board can also serve as a convenient troubleshooting device to easily check the output port on any PICAXE processor if you suspect that you may have a bad output pin.

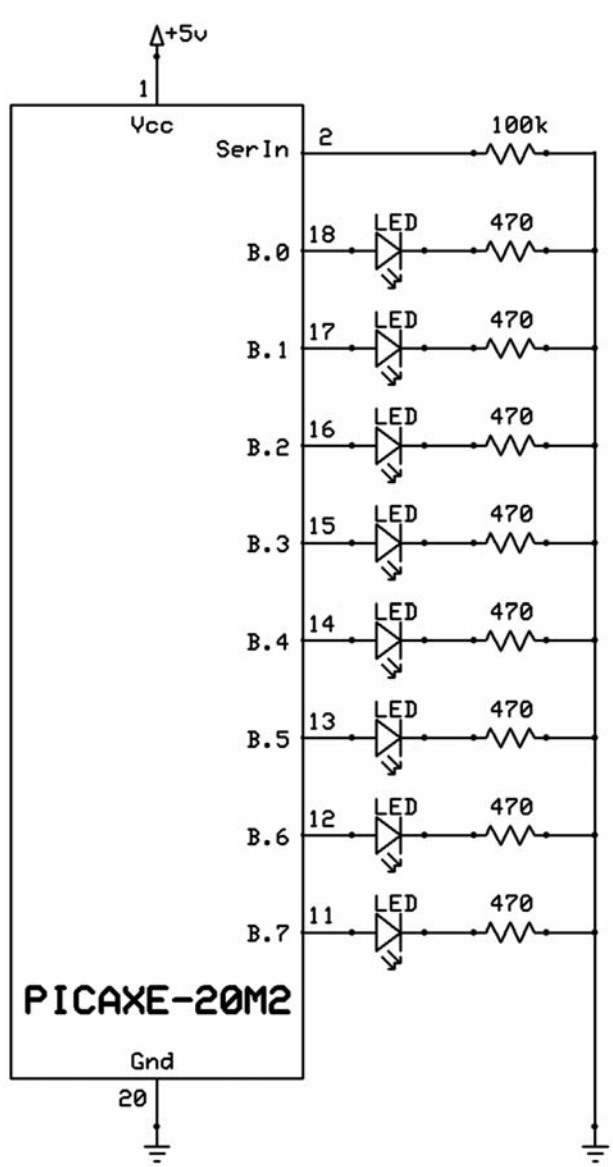


Figure 4-5 Schematic for Cylon Eye project

PARTS BIN
Stripboard, small
PICAXE-20M2 (or 20M) processor
Ten-LED bar display
470Ω nine-resistor SIP network
Resistor, 100k, 1/4 watt
Header, male, 10-pin, "reverse-mountable"

Constructing the Stripboard

Our stripboard circuit is quick and easy to construct, as long as you make sure that you have the LED bar display and the SIP resistor network oriented correctly before soldering them in place. The LED display is a 20-pin dual in-line package (DIP) unit, with its pins numbered in the same fashion as a typical integrated circuit (numbers running down one side and up the other). If you look at the LED from the bottom, pin 1 is identified, and the ten pins on that side are the anodes of each individual LED. The SIP resistor network contains nine resistors and ten pins. Pin 1, which is identified with a dot, is the common Ground connection for the nine resistors, and the nine remaining pins each connect to the other end of one of the resistors. If you're not positive about which end of the SIP is Ground, you could use a multimeter to determine which end measures 470 to all the other pins.

The Microsoft Word stripboard layout for the LED bar display is presented in Figure 4-6. As you can see, the stripboard is simple. The only aspect that requires care is that trace A is *not* severed at hole 3, but the other traces are severed at all the other holes in row 3. Trace A is the Ground trace; if it's severed, none of the LEDs will light. (We're

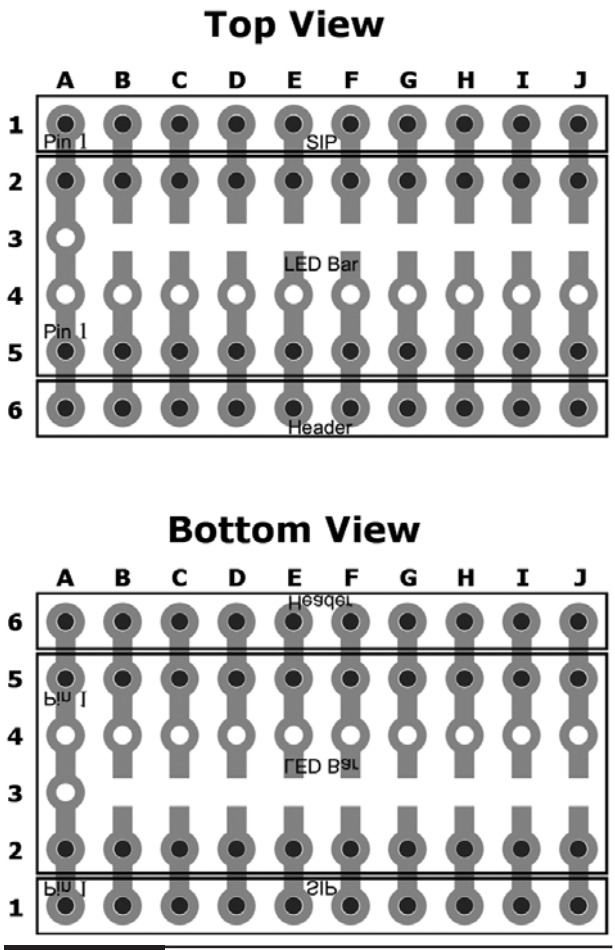


Figure 4-6 Stripboard layout for LED bar display

"sacrificing" the first LED for our Ground connection; only the remaining nine will be functional.) Be sure to orient the LED display and the SIP resistor network so that they both have pin 1 in the proper position as indicated in Figure 4-6.

The list of assembly instructions for our LED bar display board is mercifully short:

1. Prepare a piece of stripboard of the required size (10 tracks with 6 holes each).
2. Sever the traces at holes B3 through J3. (Do *not* sever the trace at A3.)
3. Insert the SIP resistor network as indicated. Make sure pin 1 is in hole A1, and solder the SIP in place.
4. Insert the LED bar display as indicated. Make sure pin 1 is in hole A5, and solder the LED in place.
5. Snip all the leads on the bottom of the board as short as possible, and sand or file the cut ends to remove any sharp edges.
6. Snip off the short ends of every pin in the ten-pin male header. (You may want to file the cut ends to remove any sharp edges.)
7. Insert the header from the top of the board as indicated. Invert the board and support the header with a small block of wood so that it remains fully seated in the board, and solder it in place.
8. Clean the flux from the bottom of the board and allow it to dry.
9. Inspect the board carefully for accidental solder connections and other problems.

To test the completed LED bar display board, insert it into a powered breadboard as shown in Figure 4-7. Ground the header pin that is in line with pin 1 of the LED display and pin 1 of the SIP resistor network. (You may want to mark this pin with a fine-point Magic Marker to indicate the Ground pin.) Power up the breadboard, and connect one end of a jumper wire to the +5V rail. Sequentially insert the other end into the breadboard at each of the remaining nine positions to make sure all the LEDs (except for the one we have sacrificed for our Ground connection) are functioning properly. Once you're sure everything is okay, we're ready to set up our Cylon Eye project.

Setting Up the Breadboard Circuit

To set up your breadboard circuit for the project, refer to the schematic presented earlier in Figure 4-5 and the photograph of the completed breadboard setup, shown in Figure 4-8. As you can see in the photo, the LED display prevents us from inserting the USBS-PA3 programming adapter in

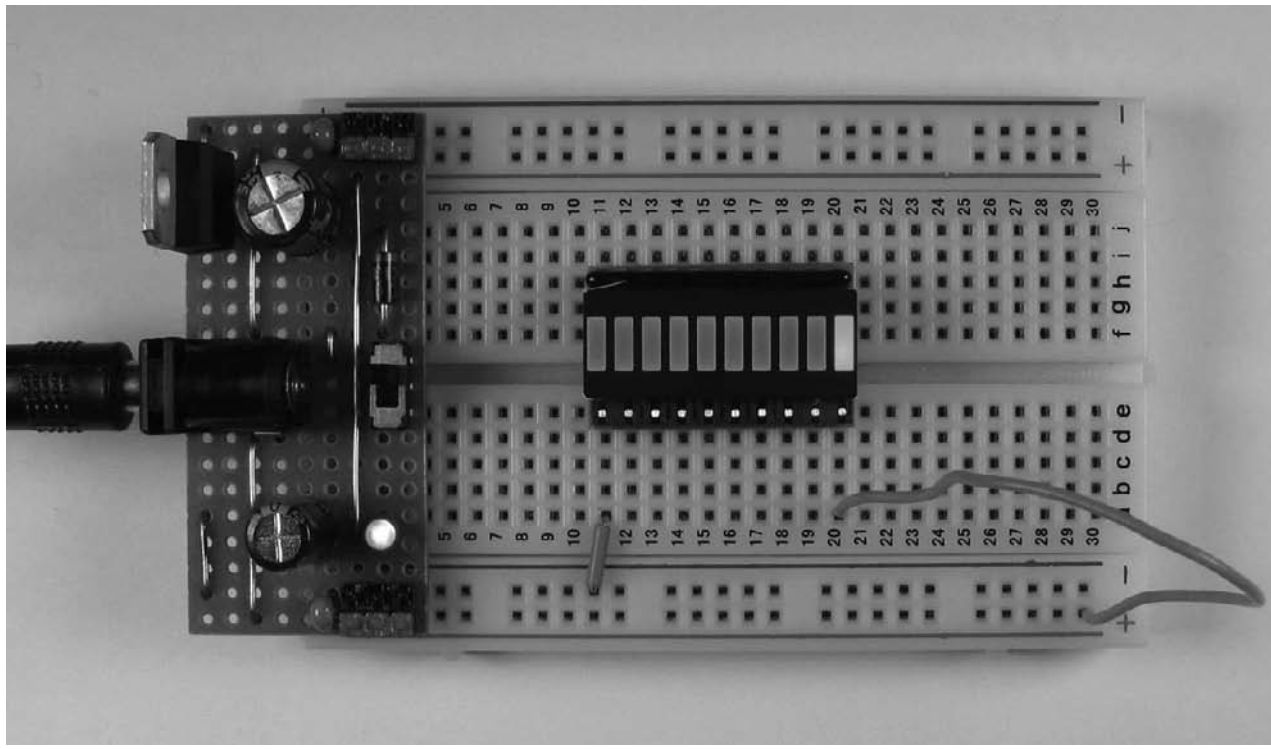


Figure 4-7 Testing the completed LED bar display board

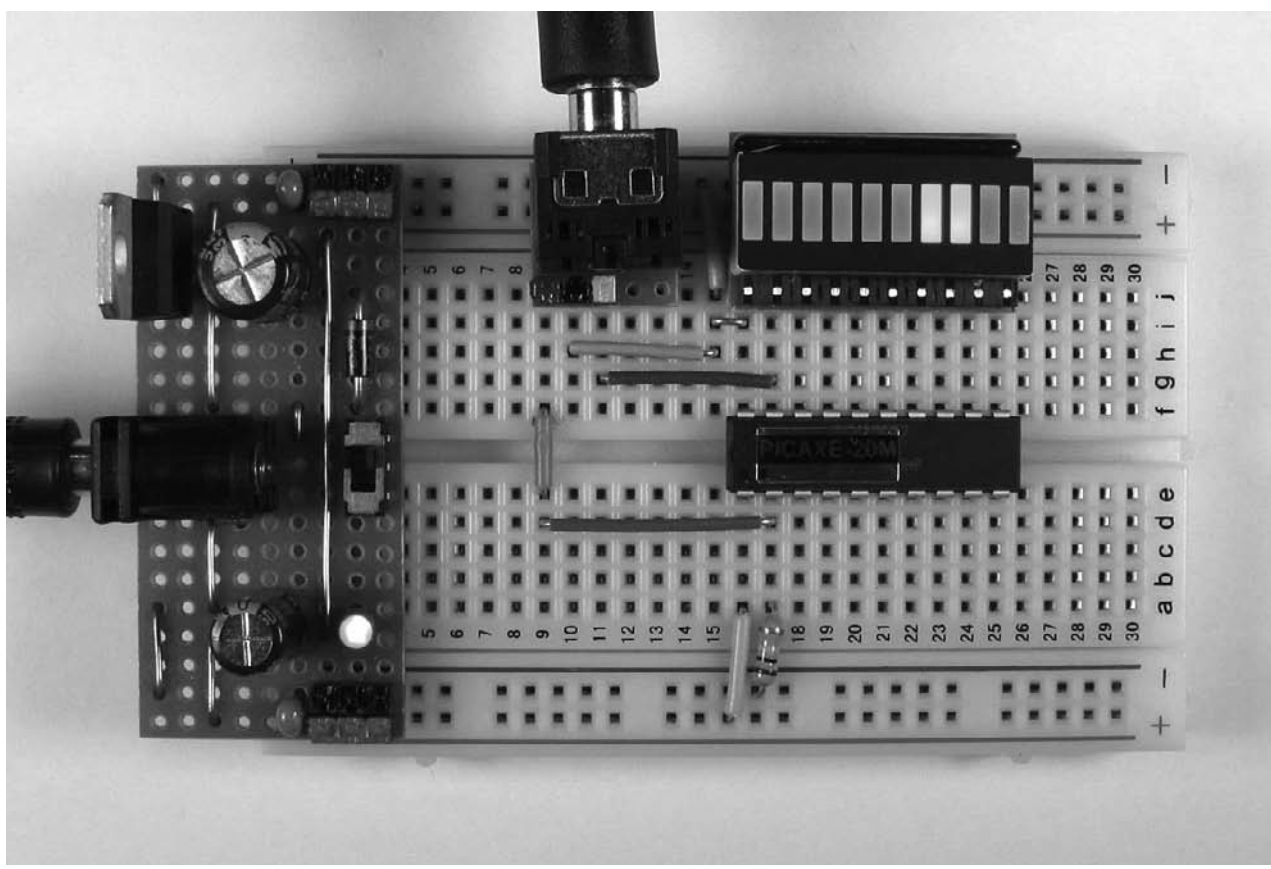


Figure 4-8 Breadboard setup for Cylon Eye project

line with the 20M2's Serial Out and Ground pins, but the required connections are still fairly simple.

Cylon Eye Software

We're going to experiment with three different programs for our Cylon Eye project. In each program, we'll be working with eight LEDs. We sacrificed the first LED for our common Ground connection and the second LED is attached to the 20M2's Serial Out line, so we'll just watch that one flicker whenever we download a new program to the 20M2. We'll be using the remaining eight LEDs, which are directly connected to the portB pins on the 20M2. For our first program, I'll provide the complete program listing; for Program 2, I'll give you the program specifications and a hint about how to proceed; you'll have an opportunity to come up with your own program before looking at my solution; Program 3 is a "programming challenge"—I'll again give you the program specifications, but leave the solution entirely up to you.

Program 1: "SimpleCylon"

For our first program (Listing 4-1), we're simply going to light the eight LEDs sequentially from output 0 to output 7 and back again in an infinite do…loop. As the name implies, it's a simple program, but the use of the *LEDs* variable may require a brief clarification. First, note that *LEDs* has been declared as another name for *outpinsB*. We can certainly use *outpinsB* directly, so this isn't

LISTING 4-1

```
' ================================ SimpleCylon.bas ================================
' Eight segments of a Red LED bar display are connected to the
' 20M2's output portB. Each bar has a 470 ohm resistor to ground.
' Current draw is > 7.25mA per LED. When all nine are lit, current = 65mA.

' === Constants ===
  symbol abit   = 100                          ' used in pauses

' === Variables ===
  symbol index = b0                            ' used in for/next loops
  symbol LED = outpinsB                        ' used to vary lit LED

' === Directives ===
  #com 3                                       ' specify serial port
  #picaxe 20M2                                 ' specify processor
  #no_data                                     ' save download time
  #terminal off                                ' disable terminal window

' ============================== Begin Main Program ==============================
  dirsB = %11111111                            ' all outputs
  LED   = %00000001                            ' light first LED

do
   for index =  2 to 8                         ' for 7 LEDs
      LED = LED * 2                            ' shift right
      pause abit                               ' slow down a bit
   next index                                  ' loop
```

(continued)

LISTING 4-1 (continued)

```
    for index = 7 to 1 step -1              ' for 7 LEDs
       LED = LED / 2                        ' shift left
          pause abit                        ' slow down a bit
    next index                              ' loop
 loop
```

necessary, but meaningful variable names tend to make a program more readable. Also, if you're wondering why *LEDs* is multiplied and divided by 2 in the for/next loops, it's because of the way binary arithmetic operates. For example, consider the binary number %00000101 (decimal 5). If you multiply it by 2, you get decimal 10, which is %00001010. In other words, multiplying a binary number by 2 shifts every digit one position to the left. Similarly, dividing a binary number by 2 shifts every digit one position to the right. So, each time through the first for/next loop, the next LED (down) gets lit and each time through the second for/next loop, the next LED (up) gets lit.

Assuming you have downloaded all the programs from my website, open SimpleCylon.bas in ProgEdit (or AXEpad) and download it to your breadboard circuit; the Cylon Eye should begin its scan. You may want to experiment with adjusting the abit constant to see how fast it can go. If you are up for a challenge, see if you can produce the same scanning by using the *high* and *low* commands instead of *outpinsB*.

Program 2: SimpleCylon2

This time, you're going to produce the same scanning effect, but with two adjacent LEDs lit at the same time. The necessary changes are minor: *outpinsB* (aka *LEDs*) needs to be initialized differently, and the limits of both for/next loops need to be adjusted. Give it a try before you look at your downloaded program.

Program 3: Cylon3

As its name implies, this one involves three LEDs being lit at the same time, but the sequence is more of a challenge. Here's how it should proceed:

1. All LEDs off
2. Output 0 LED on
3. Outputs 0 and 1 LEDs on
4. Outputs 0, 1, and 2 LEDs on
5. Three LEDs scan across to the other edge of the LED bar display
6. Outputs 6 and 7 LEDs on
7. Output 7 LED on
8. All LEDs off

The overall effect is that the three-LED group enters (one at a time) from the left, scans across the display, exits (one at a time) from the right, and then does the same thing back in the opposite direction.

Good luck, and have fun!

CHAPTER 5

The Ins and Outs of PICAXE Interfacing

PICAXE PROCESSORS SUPPORT a powerful and sometimes bewildering array of I/O functions. Even the entry-level M2-class processors have a surprisingly wide range of advanced I/O capabilities (which were listed back in Chapter 1). We will be covering many of these I/O functions throughout the subsequent chapters, but in this chapter, we're going to focus on the more basic aspects of I/O interfacing that apply to all PICAXE processors.

PICAXE I/O Interfacing

In this section, we're going to examine the basics of digital outputs and inputs. We'll also conduct two simple programming experiments to illustrate various points in the discussion. In the next chapter, we'll take a similar approach to understanding the basics of implementing analog inputs with PICAXE processors.

The special-function variables that we have already discussed are a powerful means of manipulating multiple I/O pins with one statement. PICAXE BASIC includes several additional I/O commands that are usually used to manipulate a single I/O pin, but can also be applied to multiple pins in the same statement. Here's the complete list:

- **high:** Switches an output pin to a "high" level. If the specified pin is bidirectional and happens to be set as an input when the command is issued, the pin is first converted to an output and then set to a high level.
  - **Examples:** high C.1 or high B.2, B.4, C.1,…
- **low:** Switches an output to a low level. Again, if the specified pin is bidirectional and happens to be set as an input when the command is issued, the pin is first converted to an output and then set to a low level.
  - **Examples:** low C.1 or low B.2, B.4, C.1,…
- **toggle:** Changes the level of an output pin. If it had been "high," it becomes low, and vice versa. If the specified pin is bidirectional and happens to be set as an input when the toggle command is issued, the pin is converted to an output and then set to a "low" level. Subsequent toggle commands referencing the same pin toggle its level.
  - **Examples:** toggle C.1 or toggle B.2, B.4, C.1,…
- **output:** Makes the specified pin an output and sets it to a low level.
  - **Examples:** output C.1 or output B.2, B.4, C.1,…
- **input:** Makes the specified pin an input.
  - **Examples:** input C.1 or input B.2, B.4, C.1,…

- **reverse:** Reverses the direction of the specified pin; an input becomes an output and vice versa.
    - **Examples:** reverse C.1 or reverse B.2, B.4, C.1,…

All these commands are straightforward to implement; we'll see many examples of their usage throughout the programs in this book. If you need clarification on any of them, refer to the documentation in Part II of the manual.

Implementing Digital Outputs

There's an important consideration to keep in mind whenever you're working with an output function on a PICAXE processor. In Part Two of this book, we'll be implementing several projects that involve more than one PICAXE processor. In projects like that, it's common for an output of one processor to be connected to an input of a second processor. If that input happens to be a bidirectional pin, there's always the chance that it could accidentally be reconfigured as an output. If that were to happen and the two pins were directly connected, you could easily have a situation where the output of one processor is high and the connected output of the other processor is low. Of course, that would result in a direct short and possibly damage one or both processors. To avoid this potentially disastrous situation, it's always a good idea to protect the processors by including a current-limiting resistor in any I/O connection between them. A typical value is 1k because it limits the current to 5mA.

Implementing Digital Inputs

There are two broad categories of digital inputs. First, an input pin on a processor can be connected to an output pin on a second processor (or other electronic device) to implement a connection between the two devices. In this case, there are two important considerations: protecting the devices from excessive current (as we just discussed) and protecting them from excessive voltage (which we need to discuss). Subjecting an input pin to a voltage greater than that of the processor's supply voltage is likely to damage the input pin and possibly the processor as well. Of course, if we're talking about two PICAXE processors that are powered by the same supply, this is a non-issue. The projects in this book are all in this category, so we won't have to worry about it. However, if you decide to connect a PICAXE input to voltages higher than +5V (or any negative voltage whatsoever), you will definitely need some form of level-shifting circuit to protect the PICAXE input pins from possible damage.

The second category of input device is a simple input switch that's included in a circuit to enable the user to interact with the project. From an electronics point of view, this type of switch is easy to interface. However, from a mechanical point of view, input switches present two types of problems for our breadboard-based projects. The pins on many input switches are not spaced in even multiples of 0.1 inch (2.54 mm); even if they are, the pins are frequently not long enough (or thin enough) to be easily inserted into a breadboard. Fortunately, both of these problems can be readily overcome through the use of a small stripboard circuit. Before we get into the details of interfacing input switches, we're going to build a switch adapter for use in our I/O experiments.

Building a Super-Simple Switch Stripboard

This little mini-project is harder to say than it is to build! We're going to construct a tiny breadboard adapter for a momentary push-button switch. The layout presented in Figure 5-1 is sized for a switch that I had in my miscellaneous parts collection, but it should be a simple matter to modify the layout for any small switch you happen to have on hand.

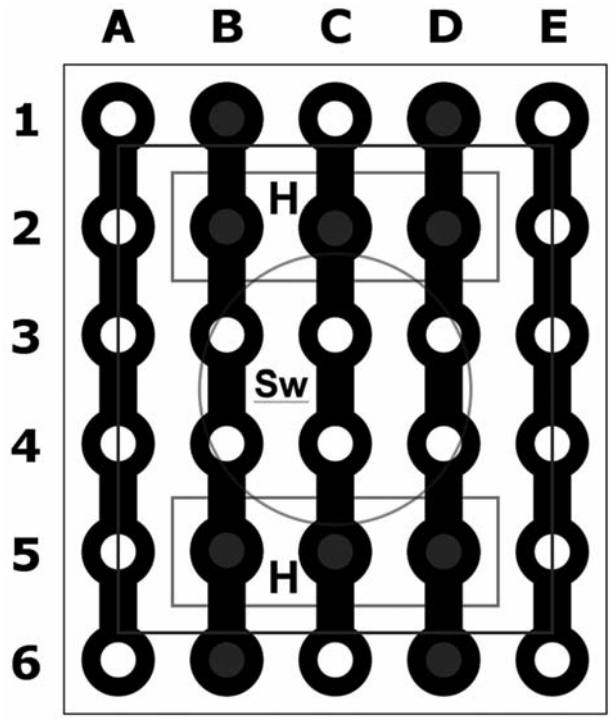


Figure 5-1 Top view of stripboard layout for momentary push-button switch

The layout in Figure 5-1 only shows the top view. We don't need both views because no traces need to be severed on the board. Note that all the traces in the layout are black, not the usual gray. This is because we're actually going to build this one upside-down—the stripboard traces will be on the top of the board when we build and use the adapter.

The parts list for the push-button switch adapter is short: a small piece of stripboard, two 3-pin sections of male headers (reverse-mountable or regular length), and a momentary push-button switch. The assembly procedure is slightly different from all our previous stripboard circuits. As I already mentioned, we're going to assemble and use this board "upside-down"—its traces will be on top of the finished board. The following list of assembly instructions is for the switch that I used; you may need to modify it to accommodate your switch:

1. Prepare a piece of stripboard of the required size (five traces with six holes each).
2. Insert the *long* ends of the two 3-pin headers (appropriately spaced) into a breadboard. Invert the stripboard and place it on the short ends of the header pins as indicated in the layout. Solder the header pins in place.
3. Remove the stripboard from the breadboard and snip off the short ends of the header pins as close as possible to the board.
4. File the cut ends of the header pins so that the push-button switch can sit close to the board on top of the soldering on the trace side.
5. Reinsert the header pins into the breadboard, and insert the push-button switch from the top (trace side) of the stripboard as indicated in the layout. (Depending on the switch that you use, you may need to bend the pins slightly to get them to fit into the holes.) Solder the pins to the traces. (Be careful not to melt the plastic of the switch.)
6. Clean the flux from the board and allow it to dry.
7. Inspect the board carefully for accidental solder connections and other problems.

Figure 5-2 is a close-up of the completed switch adapter installed on a breadboard for testing. The second switch on the left in the photo is the slide switch from our power supply project. Just for fun, I made an adapter for that one as well. (It's even simpler than the push-button adapter; you may want to give it a try.) The push-button switch (which I am pressing in the photo) is installed so that the two 3-pin headers straddle the midline of the breadboard. When pressed, the switch operates as follows: The signal that is applied to a pin on one end of either three-pin header is connected to the pin at the other end of both headers. In the photo, +5V is applied to the pin on the right end of the top header, so whenever the push button is pressed, both LEDs are powered.

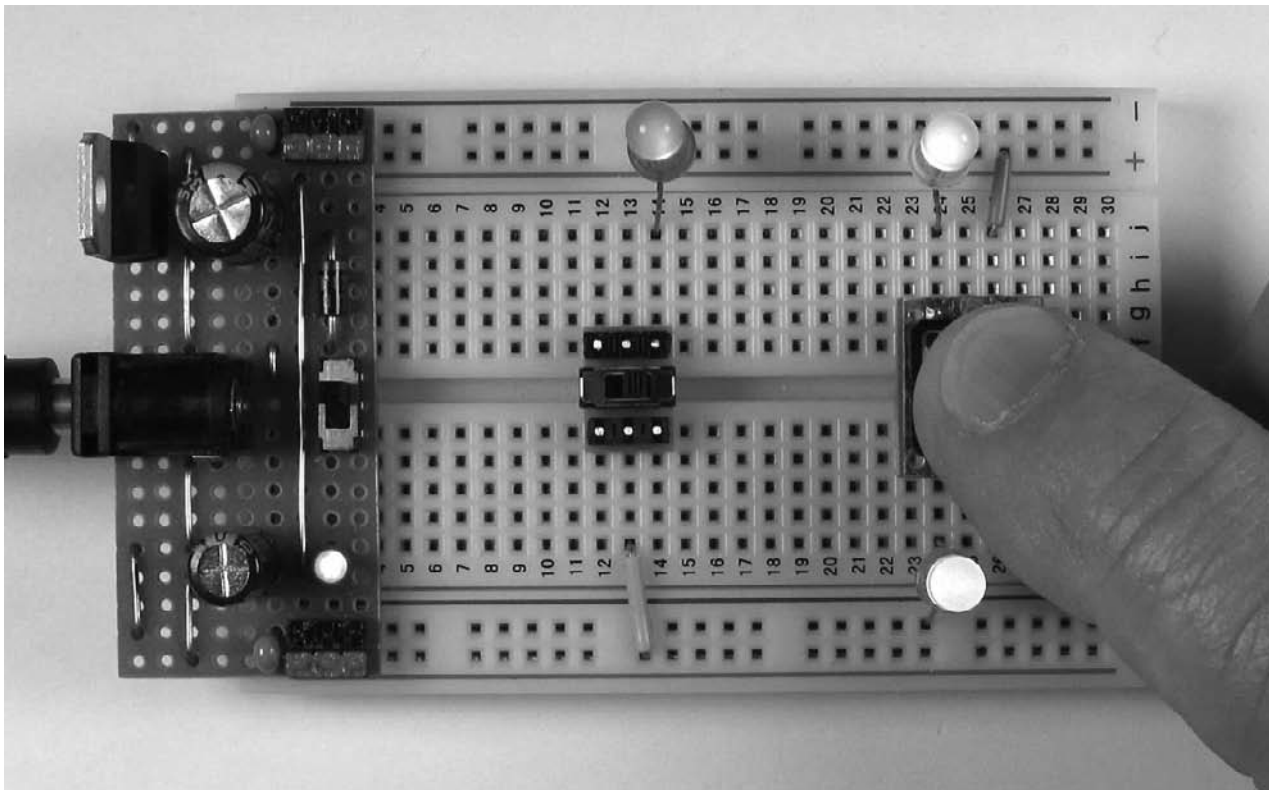


Figure 5-2 Testing the completed switch adapter

Experiment 1: ButtonCount.bas

Before we implement our first experiment, we need to discuss a problem associated with all mechanical switches. Whenever a mechanical switch is pressed or flipped from one state to another, the contacts do not make an immediately solid connection; they actually "bounce" several times before settling into the new position. Appropriately enough, this phenomenon is known as *contact bounce*, and it presents a problem that needs to be addressed in any program that includes an input switch. For example, in our first experiment we're going to program a PICAXE-20M2 to count from 1 to 127 and display each digital count pattern on the LEDs of the bar display. The count will start at %00000001 and should increment by one each time you press the push button.

If we were to ignore the contact-bounce problem, a single button-press would probably increment the count by several steps because the program is running fast enough to misinterpret one (bouncing) button-press as several presses. To make matters worse, the contacts will also bounce when the button is released. PICAXE BASIC includes a *button* command that, among other things, "de-bounces" a button-press (as usual, refer to the manual for details). However, because *button* is a powerful and flexible command, it can also be somewhat complicated to implement. A much simpler approach is to just include a short *pause* command whenever the button is pressed or released. That way, the bouncing will stop before the next command is executed. Most buttons settle in 25mS or less, so a *delay 50* statement should be more than enough, unless you are using a cheap or corroded switch.

Figure 5-3 presents the schematic that we will use for both our experiments, and Figure 5-4 is a photo of the completed breadboard circuit. In the photo, the breadboard is rotated 180 degrees from

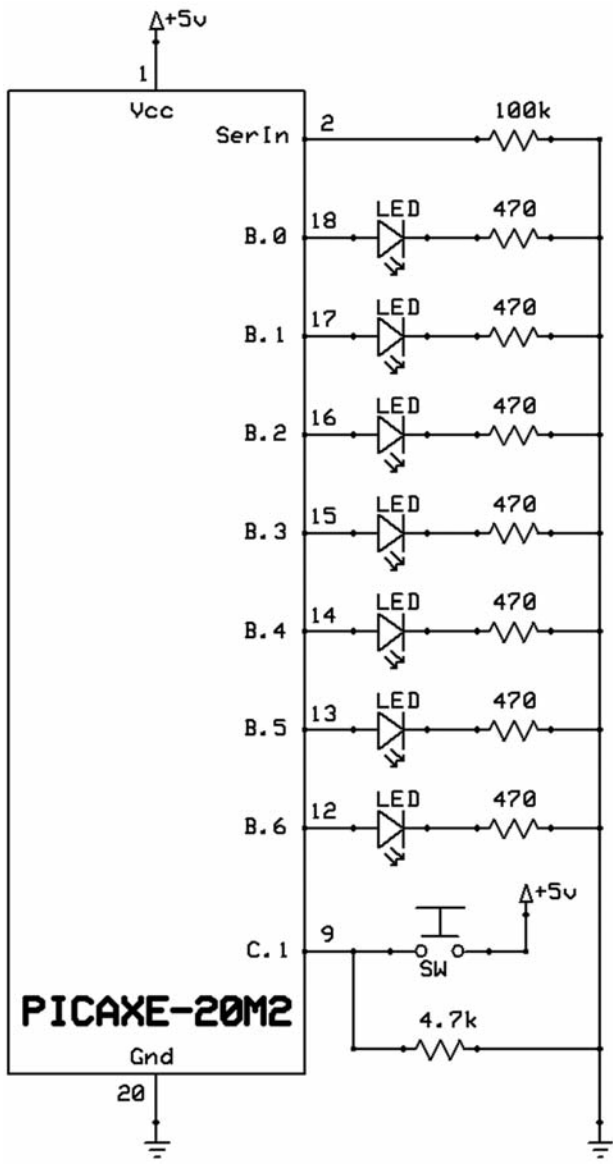


Figure 5-3 Schematic for Experiments 1 and 2

its usual orientation so that the LED lighting pattern matches that of a standard binary number—what that means will become clear when we run the program. As you see, the momentary push-button switch is connected to C.1 as specified in the schematic, and the LED bar display circuit that we developed in Chapter 4 is *not* connected to B.7 of the 20M2. It certainly could be, but this arrangement was necessary in order to maintain compatibility with the older 20M processor in case you prefer that option. (If you are using a 20M2, when you have completed both experiments, you may want to rewire the circuit to see if you can get it to count from 0 to 255.)

As long as the button is not pressed, the 4.7k resistor holds input C.1 low; when the button is pressed, C.1 is pulled high. You could reverse this arrangement (not pressed = high and pressed = low), but there are two advantages to doing it as specified in the schematic. First, it's more intuitive (high = on and low = off); second, it enables us to use two of the compiler's built-in constants (*on* = 1 and *off* = 0). Being able to write *if btn is off then…* makes a program highly readable.

In the following program listing (Listing 5-1), you can see how the previously mentioned considerations have been implemented. As long as the button is not pressed, the first *if…then* statement in the loop is skipped. Whenever a button-press occurs, that *if…then* statement is executed so *value* is incremented. Also, the two *pause 50* statements eliminate any contact bounce problems on both the "make" and "break" ends of the button-press.

The *pause abit* statement near the beginning of the program may seem unnecessary, but I included it to simulate the effect of a longer, more complex program and to demonstrate a problem with this approach. Set up the circuit, download the program to your 20M2, and see if you can diagnose the problem!

If you experiment with the circuit for a while, you will probably discover that the program occasionally fails to respond to a button-press. This happens whenever the button is pressed rapidly enough so that the "press" and "release" both occur during the *pause abit* statement. As a result, the program never notices that the button was just pressed and released. In longer programs, this same problem can occur even without the

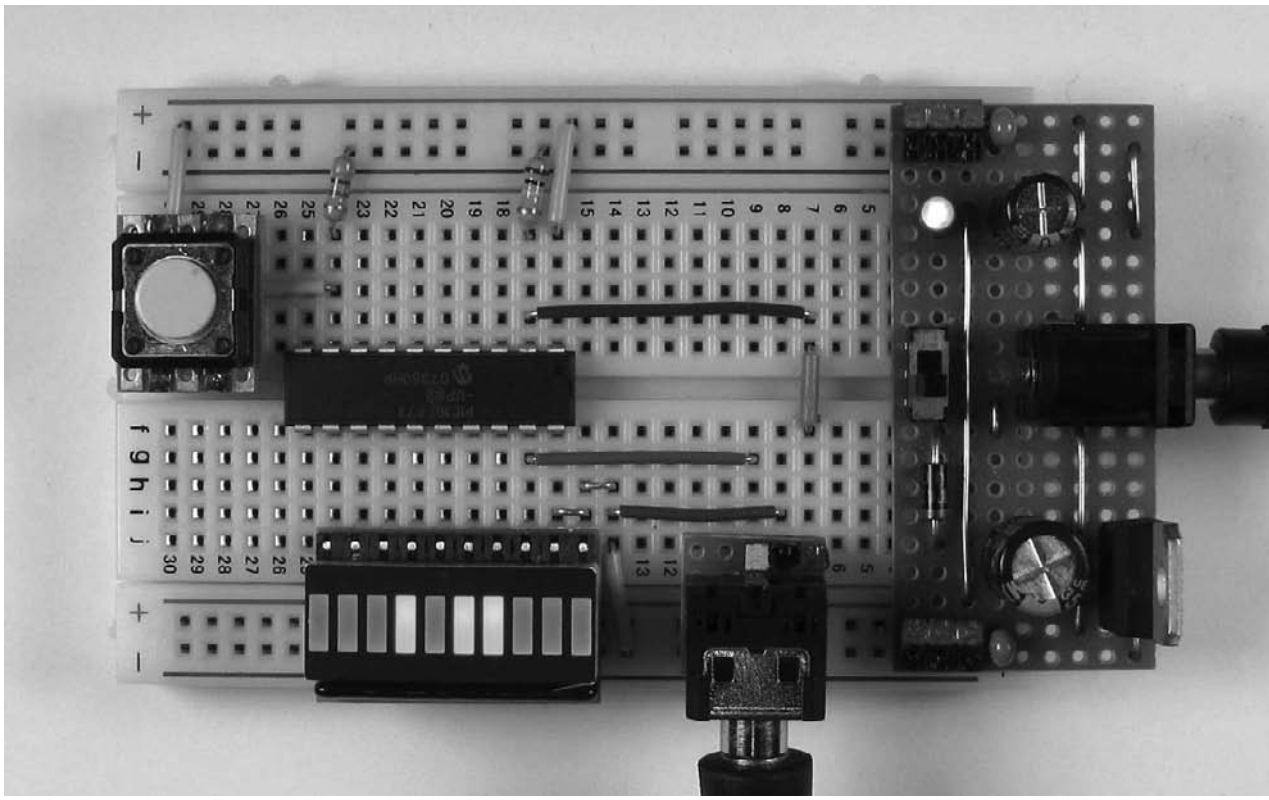


Figure 5-4 Breadboard setup for Experiments 1 and 2

presence of a *pause* statement because the processor is busy doing something else when the button is pressed and released. Fortunately, there's a powerful and efficient solution to the problem, which will be the focus of our second experiment. But before we get to that, we need to understand how to implement an *interrupt* routine in our programs.

Setting Up an Interrupt Routine

If you are familiar with the concept of an *interrupt* in a program, you will know where this is going. If not, let me briefly explain. As you know, all programs tend to proceed in an orderly fashion, either sequentially executing the program statements in the order they are written, or by jumping or branching as specified in the program. If we want a program to be able to respond to a button-press, the main program loop has to execute fast enough so that it doesn't miss the button-press while it's busy doing something else in the loop. For simple programs, this isn't usually a problem, but as the length and complexity of a program increases, it quickly becomes an impossible task.

The solution is to implement an *interrupt* routine in the program. An *interrupt* is a high-priority routine that takes precedence over anything else the program may be doing at the time. In effect, an *interrupt* tells the processor "stop what you're doing, take care of this, and then go back and pick up where you left off."

In order to implement an *interrupt*, we need to make two additions to our program. First, at the beginning of the program we need to issue a *setint* command that specifies the input conditions that we want to trigger the *interrupt*. Second, we need to include a subroutine (which must be named *interrupt*) that contains the code to be executed.

LISTING 5-1

```
' ================================ ButtonCount.bas ================================
'  Program runs on a PICAXE-20M2. Seven segments of an LED
'  bar display are connected to outputs B.0-B.6 (pins 18-12).
'  An input switch is connected to input C.1 (pin 9)
'  and held low with a 4.7k resistor.
'  However, there's a problem. Can you diagnose it?

' === Constants ===
  symbol abit = 200                          ' used for pauses

' === Variables ===
  symbol  btn = pinC.1                       ' switch on input C.1
  symbol LEDs = outpinsB

' === Directives ===
  #com 3                                     ' specify com port
  #picaxe 20M2                               ' specify processor
  #no_data                                   ' save download time
  #terminal off                              ' disable terminal window

' ============================== Begin Main Program ==============================
  dirsB = %11111111                          ' all outputs
  dirsC = %10111101                          ' C.1 & C.6 are inputs
  LEDs  = %00000001                          ' start count at 1
do
  pause abit                                 ' simulate a longer program
  if btn is on then
    inc LEDs
    pause 50                                 ' debounce time for press
tarry:
    if btn is on then tarry                  ' wait for release
    pause 50                                 ' debounce time for release
  endif
  if LEDs > 127 then
    LEDs = 1
  endif
loop
```

The syntax for the *setint* command is *setint input, mask, port*, where *input* is an eight-bit constant or variable that specifies the required input condition, *mask* is another eight-bit constant or variable that specifies which inputs are to be ignored (a 0 in any bit position of the *mask* tells the compiler to ignore that input), and *port* is B or C, depending on which one you want to use. To clarify the syntax, let's use the circuit from Experiment 1. We want the *interrupt* to occur whenever input C.1 goes high (i.e., the button is pressed) and we want to ignore the values of all the other inputs. Therefore, the required syntax is *setint %00000010, %00000010, C*—the *mask* parameter (the second one) instructs the compiler to ignore all the inputs except for input 1, and the

input parameter (the first one) indicates that we want a "high" level on input 1 to trigger the *interrupt*. The syntax may seem a little more complicated than necessary, but it enables us to specify various combinations of inputs to trigger an *interrupt*, so *setint* is a powerful command.

PICAXE interrupts are called *polled* interrupts because the input values are checked (or "polled") immediately after each line of the program is executed in order to determine whether the specified "trigger" pattern has occurred (in this case, a "high" on input 1). In addition, the inputs are also polled between each note of a *play* or *tune* command and (most importantly for our button-press example) continuously during *pause* and *wait* commands. That makes it impossible for a button-press to occur without being responded to appropriately by the program.

Experiment 2: ButtonIntrpt.bas

This experiment uses the same breadboard setup as the previous one; the only difference is in the software (Listing 5-2). We have already discussed the *setint* command, so let's take a look at the main program. All it does is configure the *interrupt*, initialize *value*, and display the count on the LEDs. After that, the main *do…loop* does nothing—it's just pretending to be a busy program! All the processing related to a button-press occurs in the *interrupt* subroutine. Whenever the button is pressed (even in the middle of the *wait* command), the program jumps to the *interrupt* subroutine, which updates the variables, handles the "make" and "break" contact-bounce, and resets *value* to 1 whenever it reaches 127.

The only potentially confusing aspect of the *interrupt* subroutine is the final *setint* command that's executed just before returning to the main program. As you can see, it's identical to the *setint* command at the beginning of the program. This command is necessary for the following reason: As soon as the *interrupt* routine is called, the compiler immediately disables the original *setint* command. If it didn't, during the entire button-press (which could last a few hundred mS), the *interrupt* routine would be repetitively interrupting itself, resulting in a deepening recursion that would certainly result in the program behaving erratically or locking up altogether. Immediately disabling the *interrupt* avoids this situation, but then it's necessary to re-establish it just before returning from the *interrupt* subroutine. Download *ButtonIntrpt.bas* to your 20M2 and see how it functions as compared to our earlier experiment. It should give you some idea of the advantages of using an *interrupt* routine to handle user input.

Interrupts are a powerful feature of PICAXE programming. If you decide to use one in your own program, it would be a good idea to read the relevant documentation in the manual because there are a couple of details we didn't cover here. However, I do want to mention two specific aspects of using interrupts before we move on to our project for this chapter, First, there are times when you might want to issue a *setint off* command. The background polling of the input pins changes some of the timing on the PICAXE chips, so you might want to disable interrupts during any portion of a program that involves exact timing and then issue another *setint* command when the critical routine has finished. Also, when the *interrupt* subroutine has finished, program execution resumes at the *line after* the one that was interrupted. As a result, the four commands that can be interrupted in the middle of their execution (*pause, play, tune, and wait*) will not fully complete their execution. For example, if a *wait 1* command is interrupted near the beginning of its execution, the total delay may only be a small fraction of a second.

LISTING 5-2

```
' ============================ ButtonIntrpt.bas ============================
'  Program runs on a PICAXE-20M2. Seven segments of an
'  an LED bar display are connected to outputs B.0-B.6
'  An input switch is connected to input C.1 (pin 9)
'  and held low with a 4.7k resistor.
' ========================================================================

' === Constants ====
  symbol abit = 200                          ' used for pauses

' === Variables ====
  symbol  btn = pinC.1                       ' switch on input C.1
  symbol LEDs = outpinsB

' === Directives ===
  #com 3                                     ' specify com port
  #picaxe 20M2                               ' specify processor
  #no_data                                   ' save download time

' ============================ Begin Main Program ==========================
  dirsB = %11111111                          ' all outputs
  dirsC = %10111101                          ' C.1 & C.6 are inputs
  LEDs  = %00000001                          ' start count at 1
  setint %00000010, %00000010, C

do
  wait 1                                     ' simulate a longer program
loop

' ======= End Main Program - Subroutines follow ======

interrupt:
  inc LEDs
  pause 50                                   ' debounce time for press

tarry:
  if btn is on then tarry                    ' wait for release
  pause 50                                   ' debounce time for release

  if LEDs = 127 then
    LEDs = 1
  endif

  setint %00000010, %00000010, C
  return
```

Project 5
Mary

"Mary" is an unusual project; on the surface, it appears to be about making music with a PICAXE processor, but it really isn't! Mary's true purpose in life is to give us some experience coordinating different types of outputs on PICAXE processors. We aren't going to get into the details of the PICAXE music-making capabilities, but if you would like to know more about the subject, you may want to read the first two installments of the "PICAXE Primer" column in *Nuts and Volts* magazine (December 2007 and February 2008). If you actually *enjoy* reading manuals, the details are also covered in the PICAXE documentation for the *play, sound,* and *tune* commands.

In this project, we're going to take a familiar tune (bet you can't guess which one) and intersperse the individual notes with a coordinated pattern on the LED display. We can use exactly the same hardware setup as our two experiments, with one small addition—a piezo "beeper." Piezos are commonly available on the various surplus sites and elsewhere, but you can also use a small speaker if you prefer. If you do, you will need a couple of additional components (see the PICAXE documentation for the *tune* command). The advantage of a piezo is that it requires no additional parts whatsoever—its positive terminal can be directly connected to a PICAXE output, with its negative terminal connected to Ground. Whichever device you decide to use, just connect it to pin C.7 on the 20M2, as shown in the photo of the breadboard setup presented in Figure 5-5.

Once you're sure that your circuit is set up correctly, we're ready to take a look at Mary's program (Listing 5-3). Before we get into the details, download it to your 20M2 and run it. As each note is played, you will see a corresponding

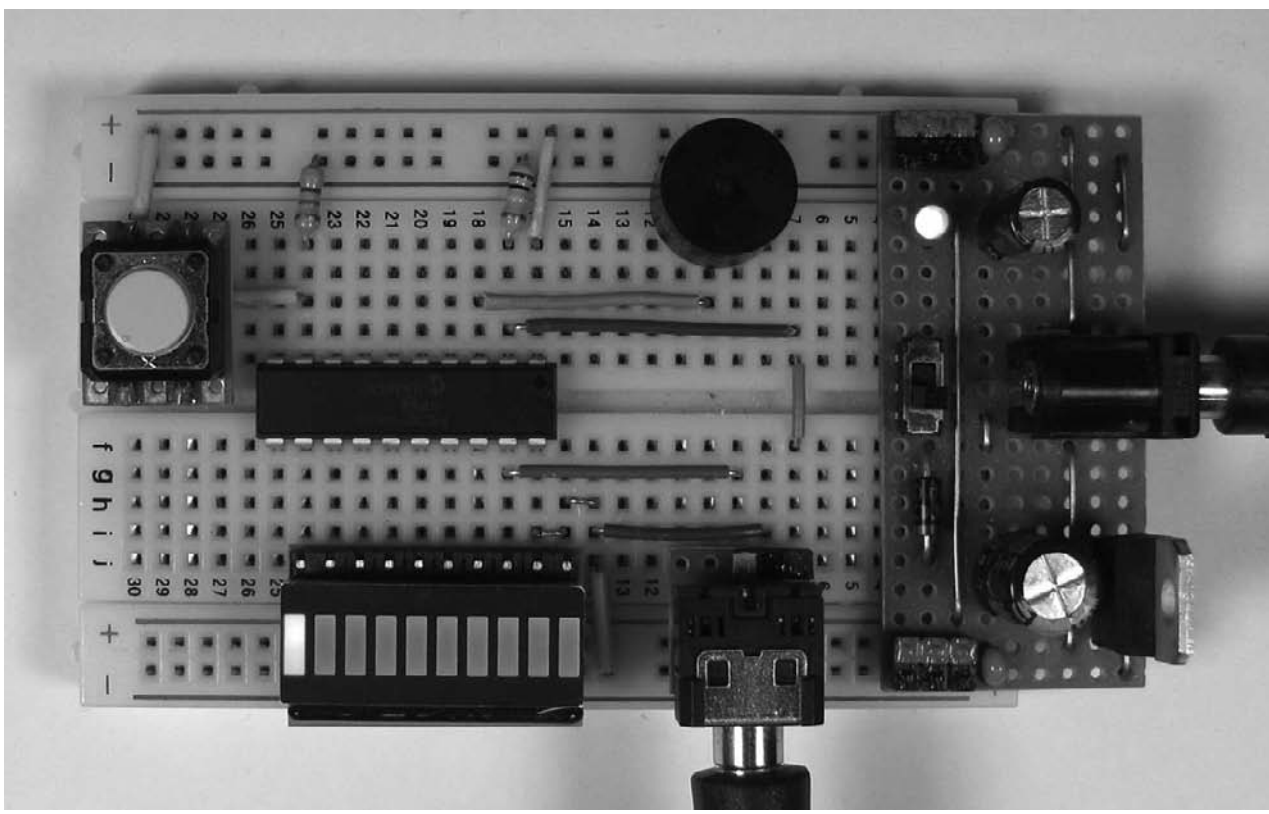


Figure 5-5 Breadboard setup for "Mary"

LED light on the display. By "corresponding," I mean that higher notes will light LEDs further up the display ("up" being towards the programming connector). In effect, the display is mimicking a simple keyboard instrument.

You don't need to know how to read music to experiment with Mary, but in case you do, Figure 5-6 presents the tune that I used to come up with the eight program constants that correspond to the eight different notes in the song. (I used the PICAXE documentation for the *tune* command to arrive at those values.) If you feel comfortable with musical notation, you may want to experiment with an entirely different tune for a project of your own design.

LISTING 5-3

```
' ================================ Mary.bas ==================================
'  Program runs on a PICAXE-20M2. Seven segments of an LED
'  bar display are connected to outputs B.0 through B.6
'  A piezo beeper is connected to output C.7.
' ============================================================================

' === Constants ====
  symbol piezo = C.7                        ' piezo on output C.7

  symbol C1 = $00                           ' C quarter-note
  symbol C4 = $80                           ' C whole-note
  symbol D1 = $02                           ' D quarter-note
  symbol D2 = $C2                           ' D half-note
  symbol E1 = $04                           ' E quarter-note
  symbol E2 = $C4                           ' E half-note
  symbol G1 = $07                           ' G quarter-note
  symbol G2 = $C7                           ' G half-note

' === Variables ====
  symbol LEDs = outpinsB
  symbol note = b0
  symbol tone = b1

' === Directives ===
  #com 3                                    ' specify com port
  #picaxe 20M2                              ' specify processor
  #no_data                                  ' save download time

' ============================ Begin Main Program ============================
do
  for note = 0 to 12
    lookup note,(E1,D1,C1,D1,E1,E1,E2,D1,D1,D2,E1,G1,G2),tone
    gosub PlayNote
  next note
  LEDs = %00000000
```

(continued)

LISTING 5-3 (continued)

```
  pause 40

  for note = 0 to 12
    lookup note,(E1,D1,C1,D1,E1,E1,E1,E1,D1,D1,E1,D1,C4),tone
    gosub PlayNote
  next note
  LEDs = %00000000
  wait 1
loop

' ==================== End Main Program - Subroutines Follow ====================

' === PlayNote Subroutine ===
PlayNote:
  select case tone
     case C1,C4
    LEDs = %01000000
     case D1,D2
    LEDs = %00100000
     case E1,E2
    LEDs = %00010000
     case G1,G2
    LEDs = %00000100
  end select

  tune piezo,4,(tone)
  return
```

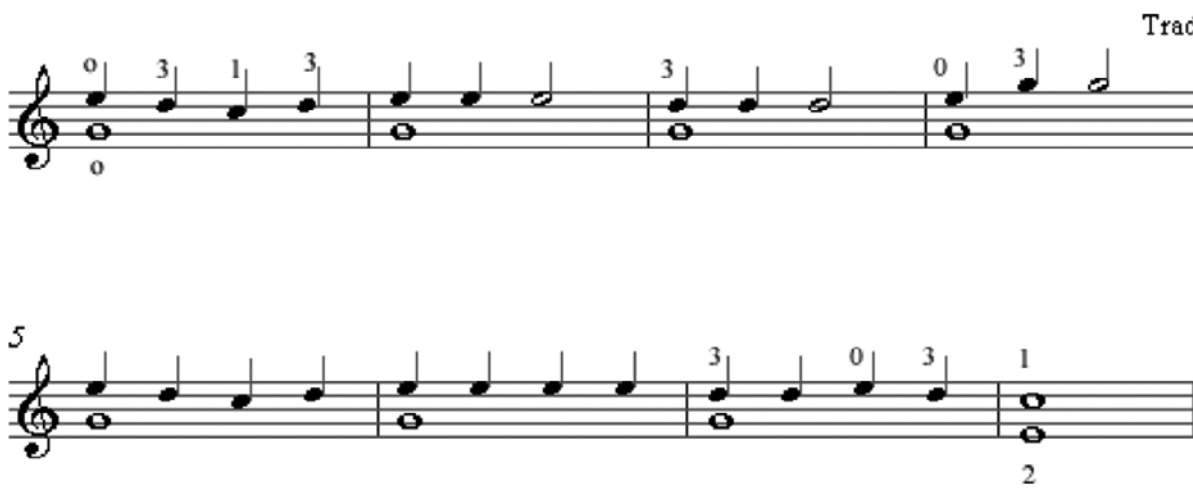


Figure 5-6 Music for "Mary Had a Little Lamb"

The song consists of 26 notes, which I split in half and entered in two *lookup* statements in the main program. (Actually, the same thing could be done with one very long *lookup* statement, but the printout would be awkward because you can't include a *line-return* character in the middle of a program statement.) Each time through a *for…next* loop, one note is fetched from the lookup table and stored in the *tone* variable. Then the program jumps to the *PlayNote* subroutine, where a *select case* statement coordinates the playing of each individual note with the lighting of the corresponding LED. You could also use four separate *if…then* statements to accomplish the same goal, but I think that the *select case* command is easier to understand—we'll find many uses for it throughout our projects.

If you don't know how to read music, I hope you didn't find Mary to be too frustrating or confusing. The most important aspect of the project is the coordination of two different types of outputs within a program. Also, even if you're not interested in making music on PICAXE processors, you may want to keep the *sound* command in mind. It's simple to use, and audio feedback can be useful in many projects.

This page intentionally left blank

CHAPTER 6

Introduction to ADC Inputs on M2-Class Processors

Now that we have covered the basics of digital I/O, we're ready to investigate how the M2-class processors handle analog-to-digital conversions (ADC). In one sense, this is a much simpler topic because there are essentially only two different commands to learn:

- calibadc (or calibadc10)
- readadc (or readadc10)

The basic concept underlying all analog-to-digital conversion on PICAXE processors is that the value of an analog voltage is approximated by converting it to a digital number that is proportional to the original value. The *readadc* command, for example, converts an input voltage level to an eight-bit binary number. If you're familiar with binary numbering, you know that an eight-bit binary number can represent 256 (i.e., 2^8) different values ranging from decimal 0 to decimal 255. In comparison, the *readadc10* command, which performs a ten-bit ADC, produces a binary value that ranges from 0 to 1023. Naturally, the result of a *readadc10* command is much more accurate than that of the *readadc* command, but *readacd10* takes longer to execute and it requires a *word* variable because a *byte* variable isn't large enough to hold the ten-bit result. Frequently, the increased accuracy isn't really necessary and the *readadc* command is more than adequate for the task at hand. The necessary computations are also easier, so that's what we'll use in our first example.

PICAXE processors base all analog-to-digital conversions on the processor's supply voltage. That means an input voltage that's exactly equal to the supply voltage would be assigned a value of 255 (the largest eight-bit binary number) by the *readadc* command, or a value of 1023 (the largest ten-bit binary number) by the *readadc10* command. Lower voltage levels would be assigned proportionately lower values, and if the input voltage were at Ground level, both commands would assign it a value of 0. As I mentioned in the previous chapter, applying a voltage to a PICAXE pin that is greater than the processor's supply voltage or lower than Ground (i.e., negative) is likely to damage or destroy the pin and/or the processor. That's an important point to keep in mind as we work through our experiments.

The M2-class processors also include two ADC calibration commands (*calibadc* and *calibadc10*) that can be especially helpful in battery-powered applications. In these situations, the supply voltage gradually decreases over time as the battery weakens. Therefore, the ADC reading of the same voltage level will also change over time. This problem can be overcome by using one of the calibration commands, because they provide access to an internal fixed-voltage reference of +1.024V that remains constant even as the supply voltage decreases. In effect, the *calibadc* (or *calibadc10*) command executes a *readadc* (or *readadc10*)

command on the internal fixed-voltage reference. Once your program has determined the ADC reading associated with the fixed-voltage reference, you can use that value to compute a more accurate value for an external ADC measurement (e.g., the real-time battery-supply voltage). We will experiment with the calibration command in the next chapter, but for the experiments and project in this chapter we'll assume our supply voltage remains constant at approximately 5.0V.

Voltage Dividers

An underlying concept in many analog-to-digital conversions is that of the "voltage divider." In its simplest form, a voltage divider is a circuit consisting of two resistors in series, as shown in Figure 6-1. The math is included in the figure, but let's state the voltage divider rule in English: The voltage drop across a resistor is proportional to the magnitude of the resistor. In other words, in Figure 6-1, the output voltage (which is the voltage across Rb) is the same proportion of the input voltage as Rb is of the total resistance. Using the values from the figure: Vout/5 = 4700/5700, so Vout is approximately 4.12V.

Experiment 1: A Simple Voltage Divider

For our first experiment, we're going to implement the voltage divider circuit from Figure 6-1 and see how close we can come to the predicted voltage level of 4.12V. To keep things simple, we'll use an 08M2 processor. If you check the pin-out of the 08M2, you'll see that it has three ADC inputs on pins C.1, C.2, and C.4. We're going to use the ADC input on C.1 (external pin 6), but they all function identically, so it really doesn't matter. The schematic is presented in Figure 6-2, and the breadboard layout is shown in Figure 6-3.

The software for our first experiment is presented in Listing 6-1. As usual, it's fairly simple. The only two aspects that require brief explanations are the *readadc* and *sertxd* statements. Since we are using *readadc* (*not readadc10*), we only need an eight-bit byte variable to store the result, so *Vin* has been declared accordingly. In the *sertxd* statement, the "#" symbol is crucial. Without it, the statement would send the value of Vin; while that may sound reasonable, it would actually produce "garbage" in the terminal window. The reason is that all communication with any serial terminal is based on the ASCII code, which assigns a value to all printable and nonprintable characters.

$$Vout : Rb = Vin : (Ra + Rb)$$

$$Vout : Rb = Vin : Rtotal$$

$$\frac{Vout}{Vin} = \frac{Rb}{Rtotal}$$

$$\text{or, } Vout = \frac{Rb}{Rtotal} * Vin$$

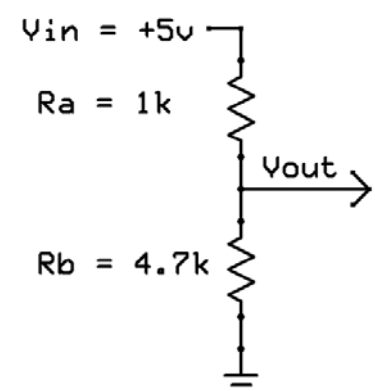


Figure 6-1 A basic voltage divider circuit

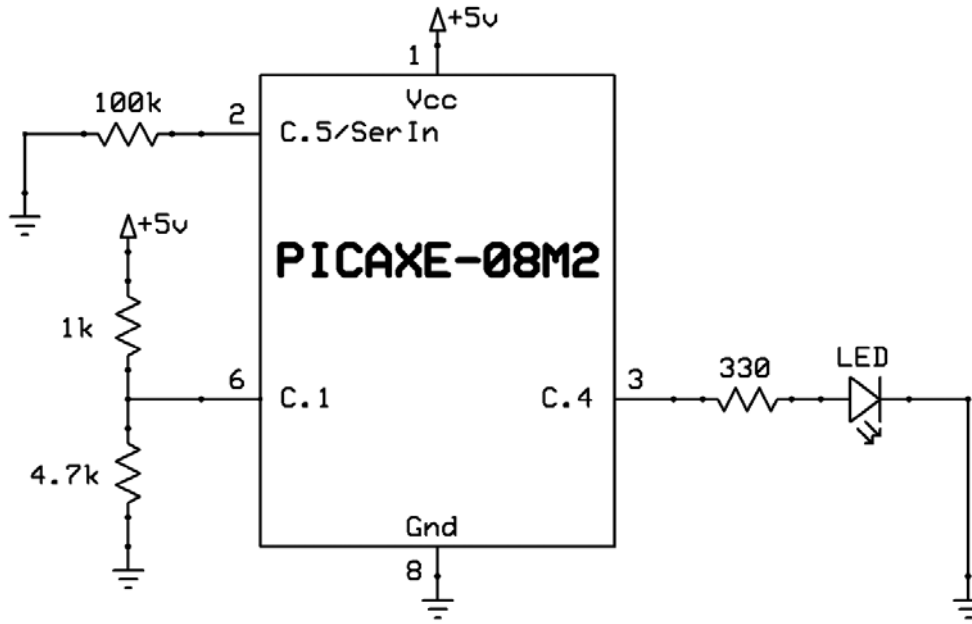


Figure 6-2 Schematic for voltage divider experiment

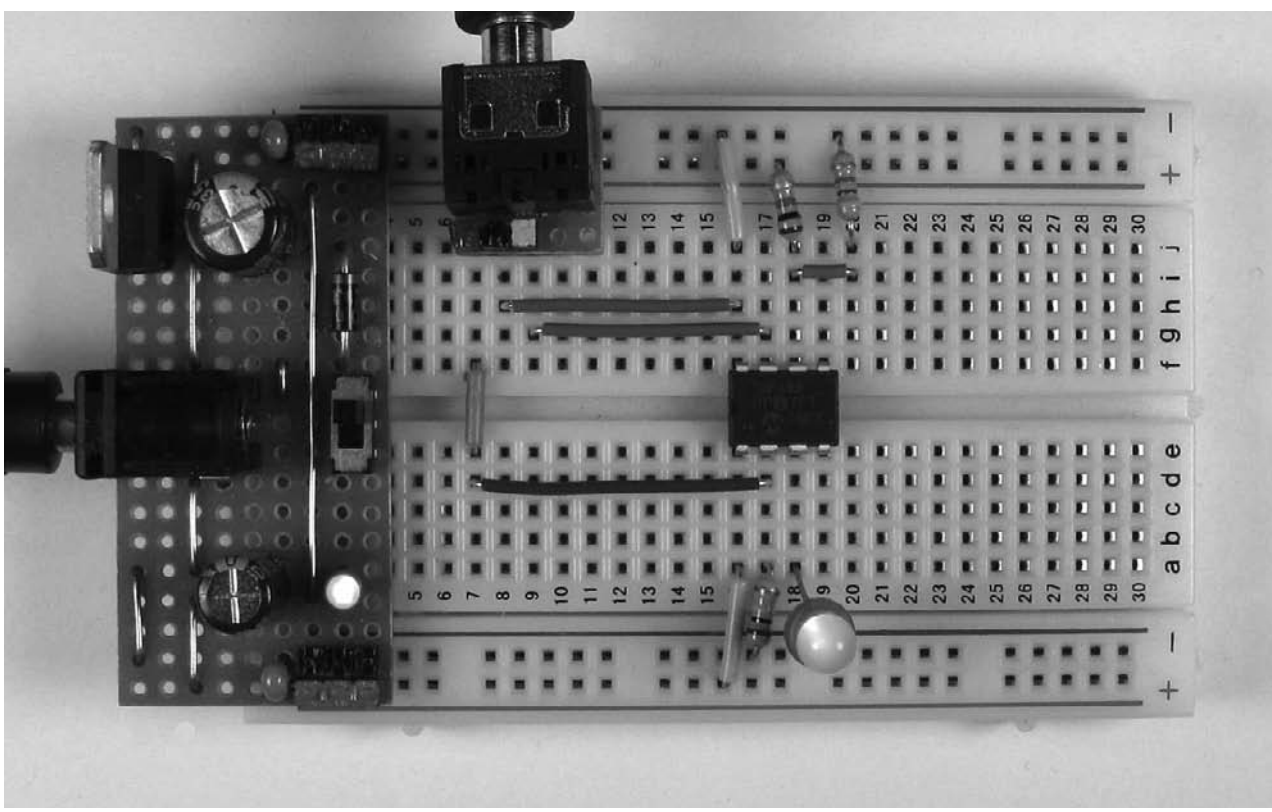


Figure 6-3 Breadboard layout for voltage divider experiment

Consequently, transmitting one value to the terminal window will result in one character being displayed (if it's a printable character). For example, if our ADC result happened to have a value of 122 and we transmitted it without the "#" symbol, we would see a "z" in the terminal window because that's the ASCII character associated with the value of 122. Obviously, we want to see "122," not "z," and the "#" does exactly that; it breaks the value down into individual digits and transmits the appropriate ASCII codes to display them. So, "122" becomes "1," "2," and "2," and that's what will be displayed in the terminal window.

The *cr* and *lf* arguments in the *sertxd* statement also require an explanation. They are both predefined constants in the PICAXE compiler. *Cr* equals 13 (the ASCII code for "carriage return"), and *lf* equals 10 (the ASCII code for "line feed"). They are two nonprintable characters that, in effect, move the cursor in the terminal window to the next line.

Download the program to your breadboard setup and run it (Listing 6-1). The terminal window should open automatically, because a *#terminal 4800* directive is included in the program. If not, just choose PICAXE | Terminal in the menu or

LISTING 6-1

```
' ============================== VoltageDiv1.bas ==============================
' Program runs on a PICAXE-08M2 processor.
' It repetitively updates an ADC voltage
' measurement on ADC1 and displays the results
' in the terminal window. The program also
' blinks an LED to show it's working.
' =============================================================================

' === Constants ====
  symbol  LED = C.4                          ' LED on C.4 (pin 3)
  symbol Vdiv = C.1                          ' voltage divider on C.1

' === Variables ====
  symbol ADCval= b0

' === Directives ===
  #com 3                                     ' specify com port
  #picaxe 08M2                               ' specify processor
  #terminal 4800

' ========================== Begin Main Program ============================
  dirsC = %00010101                          ' set up I/O directions
do
  high LED                                   ' for debugging
  readadc Vdiv, ADCval                       ' get ADC value
  pause 500                                  ' slow down
  low LED
  sertxd (#ADCval,cr,lf)                     ' send ADCval to TW
  pause 500
loop
```

press the F8 key. Also, make a note of the value(s) that are displayed in the terminal window.

When I ran the program, I got an ADC value of 210. Your results may not be identical, but they should be fairly close. Standard carbon resistors are usually rated at 5 percent tolerance, which means that a 1k resistor can measure anywhere between 950 and 1050 ohms. If an application requires more accuracy than that, you can use the measured values of the resistors (rather than their nominal values) in your computations. A second alternative would be to purchase 1 percent precision resistors for use in your ADC projects. However, standard 5 percent resistors are good enough for Evil Genius work.

Let's assume you obtained the same reading that I did—the question remains, what does 210 mean in terms of the value of the input voltage? All it takes to figure out the answer is another proportion: Vin/5 = 210/255. (In English, the unknown voltage is to +5V as the ADC reading is to the number of ADC steps.) Solving that one gives us 4.12V, which happens to be the exact result we obtained earlier in Figure 6-1.

Experiment 2: Let Your PICAXE Do the Math!

In this experiment we're going to use the same breadboard setup, but modify our program so that the 08M2 takes over the chore of computing the value of the input voltage (see Listing 6-2). This would be a simple matter, except for the fact that PICAXE processors are only capable of doing integer math—no fractions or decimals allowed! The following program listing only includes the necessary changes to our first program. You can either edit the first program to include the changes or download the complete program (VoltageDiv2.bas) from my website.

LISTING 6-2

```
' ================================ VoltageDiv2.bas ================================
' This time, the 08M2 does the input voltage computation.

' === Variables ===
' Variables b0 through b4 are directly used in
' the bintoascii command, so don't use them here.

  symbol ADCval = b5
  symbol Vin = w3                          ' we need a word variable (see text)

' ============================== Begin Main Program ==============================
  dirsC = %00010101

do
  high LED
  readadc Vdiv, ADCval
  pause 500
  low LED
  Vin = ADCval * 100 / 51
  bintoascii Vin, b4, b3, b2, b1, b0
  sertxd (b2,".", b1, b0," volts",cr,lf)
  pause 500
loop
```

Let's take a look at the changes we need to make in the program. First of all, *Vin* needs to be declared as a word variable, not a byte variable—we'll soon see why that's necessary. The rest of the changes are contained in the three statements immediately following the *low* LED statement:

```
1.) Vin = ADCval * 100 / 51
```

In order to understand this statement, we need to go back to the proportion we just solved and do a little algebra.

$$\frac{Vin}{5} = \frac{210}{255}$$

$$\frac{Vin}{5} = \frac{ADC}{255}$$

$$Vin = 5 * \frac{ADC}{255}$$

$$Vin = \frac{ADC}{51}$$

If the PICAXE compiler could do decimal calculations, we would be finished at this point because this equation yields the same result as before (4.12V). However, because the compiler can't handle decimals, we need to include one more step. We're going to multiply the right side of the equation by 100 to trick the compiler into thinking that the final answer is 412V rather than 4.12V. This little trick is also why we needed to declare *Vin* as a word variable; *ADCval* could possibly be as large as 255 and multiplying that by 100 yields 25,500, which is much too large to be held in a byte.

```
2.) bintoascii ADC, b4, b3, b2, b1, b0
```

This statement may seem a little daunting at first, but *bintoascii* is actually a powerful and simple-to-use command. Essentially, it separates out each of the digits in its argument (in our case, *ADCval*). Since *ADCval* is a word variable, its largest possible value is 65,535 so it can have as many as five digits. Therefore, we need to include five variables to hold each possible digit of *ADCval.* I could have declared the five variables as *tenthousands, thousands, hundreds, tens,* and *units*, but this is one of the few times that I think it's better to directly use the default variable names instead. By using the variable names *b4* through *b0*, each name tells us which digit it is (just think in terms of powers of 10); 10^4 = the ten-thousands digit, 10^3 = the thousands digit, etc. Of course, if you use this approach, it's important to not declare variables b4 through b0 for another use in the same program.

```
3.) sertxd (b2,".", b1, b0," volts",cr,lf)
```

At this point, we have separated out the digits of *Vin*, but its value is 100 times too large. Because the maximum input voltage is +5V, our *Vin* variable could now be as large as 500. Therefore, we only need to deal with b2, b1, and b0 (the hundreds, tens, and units digits). The *sertxd* statement simply transmits the three required digits (hundreds first) and inserts a decimal point in the appropriate place to return the number to its original value (1/100th of *Vin*). Finally, we'll also transmit " *volts*" (note the leading space) to make the output more meaningful.

Project 6
A Three-State Digital Logic Probe

A logic probe circuit can be as quick and easy as an LED and a current-limiting resistor with a wire soldered to each end of the circuit. Simply connect the ground lead to Ground and touch the other lead to the point in a circuit that you want to test. If the LED lights, that point is at a high level; if it doesn't, the point is not at a high level. However, this approach has two drawbacks—it can't

distinguish between a point that is low and a point that is entirely disconnected from the circuit, and it draws a fair amount of current, which could affect the circuit's operation.

Our digital logic probe project employs a simple ADC circuit that solves both these problems. We're actually going to construct two versions. The first is a breadboard version that we'll use to test the circuit. Once we're sure the project is functioning correctly, we'll build a stripboard version of the circuit and install it in a project case (actually, a plastic test tube). Before we get started, it's important to note that this probe is only designed for working with digital logic circuits powered by +5V. *Connecting the logic probe to voltages greater than +5V will most likely damage or destroy the 08M2 processor.*

Breadboard Version of Digital Logic Probe

The necessary parts for the breadboard version are listed in the Parts Bin; as usual, all the parts are available on my website. The schematic for the project is presented in Figure 6-4, and the breadboard layout is shown in Figure 6-5. The three diodes in the circuit require a brief explanation. They aren't really necessary in the breadboard version, but they will be in the final stripboard project; I decided to include them here rather than present two different schematics for the same project. As we'll soon see, the final project will connect to the power rails of a breadboard, and it would be relatively easy to insert the connector into the rails incorrectly (reversing the +5V and Ground), which would probably damage the 08M2. Diode D1 blocks any current flow in the wrong direction, protecting the 08M2 from possible damage. Even with both LEDs lit, the project doesn't draw much more than 30mA. At that level, the BAT85 has a forward voltage drop of approximately 0.5V. Therefore, the probe is actually powered by +4.5V, which is more than adequate for the 08M2. Diodes D2 and D3 simply ensure that any voltages present at the probe tip also will not exceed 4.5V. As I already mentioned, you don't really need to include the diodes in the

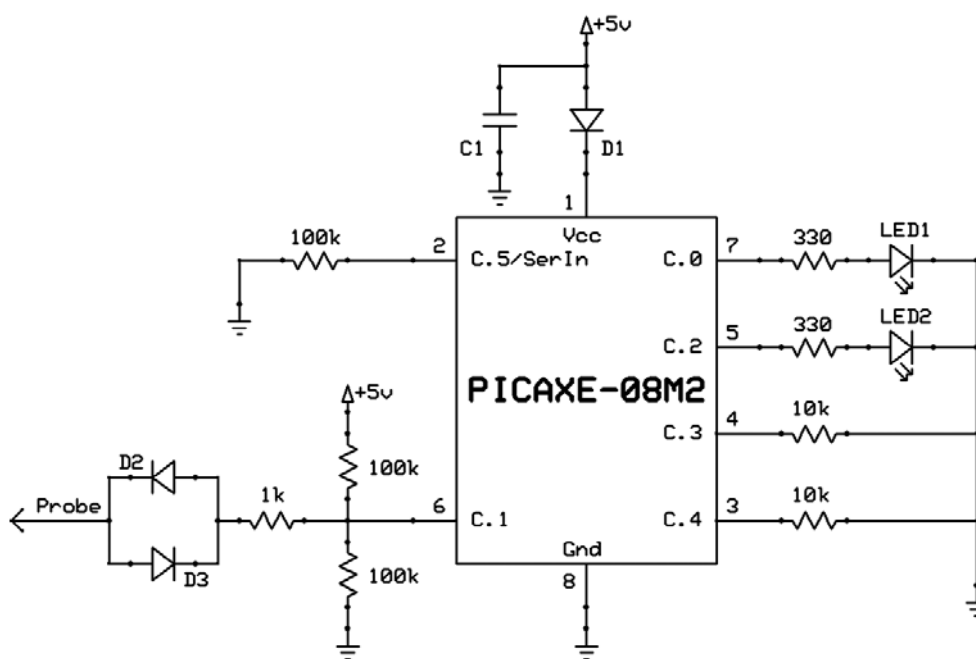


Figure 6-4 Schematic for digital logic probe circuit

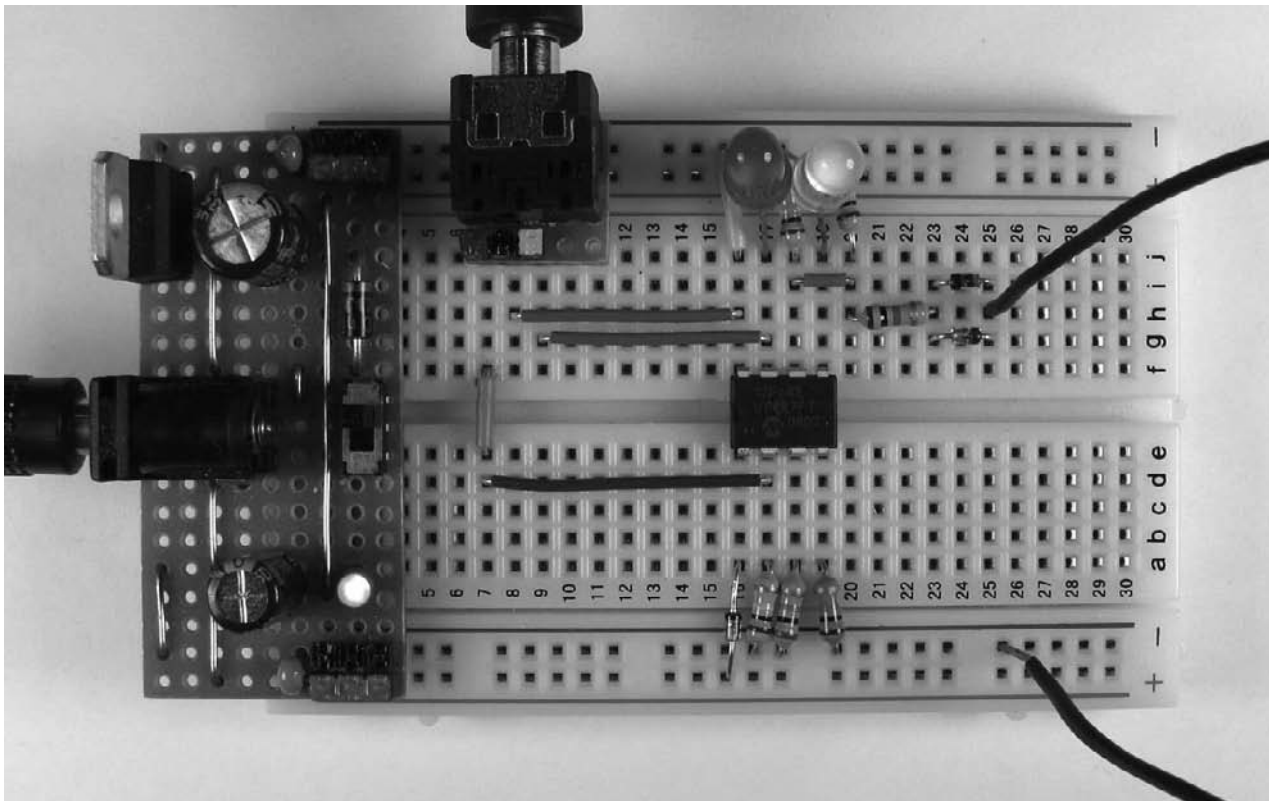


Figure 6-5 Breadboard layout for digital logic probe circuit

breadboard test circuit; just be sure to add them in the final stripboard version.

The software for our logic probe is presented in Listing 6-3. It's really simple: just two statements executing in an infinite *do…loop*. Let's start with the *select case* statement, which is one of my favorites. It performs the same function as a series of *if…then* statements but it's much simpler to write and to understand. Essentially, the *select case* statement looks at the value of the specified variable (*ADCval*) and then executes the first *case* clause that evaluates to true. In our program, if *ADCval* is greater than 190, then the *LEDs* variable is configured to light the red LED. If ADCval is not greater than 190, the next case clause is evaluated. There are two points to keep in mind regarding this process. If none of the *case* clauses is true, the statement executes the *else* clause (if one is stated). If there is no *else* clause, the program advances to the next statement without executing any of the *case* clauses. Also, whenever one of the clauses is true (and therefore executed), none of the subsequent clauses are even evaluated; the program just moves on to the next statement.

The only other aspect of the program that requires explanation is how I arrived at the values

PARTS BIN

Part	Label
Three BAT85 diodes	D1, D2, D3
Two resistors, 330Ω	
Resistor, 1k	—
Two resistors, 10k	—
Three resistors, 100k	—
Capacitor, 0.01μF	C1
LED, red	LED1
LED, green	LED2
PICAXE-08M2 (or 08M)	—

LISTING 6-3

```
' ================================ LogicProbe.bas ================================
' This program runs on a PICAXE-08M2 at 8 MHz.
' It implements a 3-state digital logic probe.
' ================================================================================

' === Constants ==========
  symbol Probe = C.1                          ' probe on C.1
  symbol Red   = %00000001                    ' turn on red LED
  symbol Green = %00000100                    ' turn on grn LED
  symbol Blank = %00000000                    ' both LEDs off

' ADC input boundaries                        ' ADC values (Open=134)
' ____________________________________________________________________________
  symbol Hi = 190                             ' High = 254
  symbol Lo =  70                             ' Low  =  10

' === Variables ===
  symbol ADCval = b0                          ' ADC measurement
  symbol   LEDs = outpinsC

' === Directives ===
  #com 3                                      ' specify serial port
  #picaxe 08M2                                ' specify processor
  #terminal off                               ' disable terminal window

' ============================ Begin Main Program ==============================
  setfreq m8
  dirsC = %00000101                           ' LEDs on outputs 2 & 0

do
  readadc Probe, ADCval
  select case ADCval
    case > Hi
      LEDs = Red
    case < Lo
      LEDs = Green
    else
      LEDs = Blank
  endselect
loop
```

for the *Hi* and *Lo* constants. (As an aside, I couldn't use *High* and *Low* as names for these two constants because the compiler reserves those two words for the *High* and *Low* commands.) In order to understand how the *Hi* and *Lo* values were determined, let's consider the three possibilities: the point in the circuit that the probe is touching is either at a digital *high* level, a digital *low* level, or *open* (i.e., not connected to any other point in the circuit).

1. **Probe is *open*:** In this case, the 1k resistor is effectively disconnected from the circuit, so the two 100k resistors will place the ADC reading near the middle of its range, that is, ADCval = 127. (My actual ADC value in this situation was 134).
2. **Probe is *high*:** In this case, the 1k resistor is in parallel with the 100k resistor that is connected to the supply voltage, so the total resistance on the upper side of the voltage divider is (1k * 100K)/(1K + 100K) = 990Ω, which will raise the ADC reading close to the top of its range. (My actual ADC reading in this situation was 254.)
3. **Probe is *low*:** In this case, the 1k resistor is in parallel with the 100k resistor that is connected to the Ground, so the total resistance on the lower side of the voltage divider is 990Ω, which will lower the ADC reading almost to 0. (My actual ADC reading in this situation was 10.)

The *Hi* constant is the dividing line between a high input and an open input, so I defined *Hi* to be 190, which is approximately halfway between 134 and 254. Similarly, I defined *Lo* to be 70 because that's roughly halfway between 10 and 134.

Download LogicProbe.bas to your breadboard circuit. For test purposes, you can just use an eight- or ten-inch piece of jumper wire for the probe. When you (carefully) touch the jumper wire to a high point in the circuit (e.g., pin 1 of the 08M2), the red LED should light; when you touch a grounded point (e.g., pin 8 of the 08M2), the green LED should light; whenever the jumper is not touching anything (or touching a point on the breadboard that isn't connected to anything in the circuit), neither LED should be lit.

Stripboard Version of Digital Logic Probe

Obviously, a digital logic probe on a breadboard isn't a convenient test instrument. What we need is a device that's small enough to be comfortably held in one hand as we use it to examine our experiments and projects. The "case" I chose for

Figure 6-6 Size comparison of logic probe and mechanical pencil

this purpose is a small plastic test tube about four inches long and two-thirds of an inch in diameter; it turned out to be comfortable to use, but it was a challenge to squeeze the logic probe circuit into such a small space. To give you a sense of the size of the probe, Figure 6-6 is a photo of the completed project next to a mechanical pencil.

The parts list for the stripboard version of the logic probe circuit are listed in the Parts Bin, and the stripboard layout is presented in Figure 6-7. We'll discuss the specifics when we actually construct the probe, but there are a couple of points related to the stripboard layout that I want to clarify before we begin. First, in the layout, you might have noticed that there are no holes in column AD. This is because I included the end of the stripboard (which has no holes in it) in this project because I wanted to be able to round the end of the board to match the curve of the bottom of the test tube, and I preferred not having holes in that area. If you look closely at the photo in Figure 6-6, you can see that the end of the board has been rounded.

PARTS BIN

ID	Part
D1, D2, D3	Three BAT85 diodes
—	Resistor, 1k, 1/6 watt
—	Two resistors, 10k, 1/6 watt
—	Three resistors, 100k, 1/6 watt
C1	Capacitor, 0.01µF
LED1	LED, red, resistorized
LED2	LED, green, resistorized
—	IC socket, 8-pin, machine-pins
—	PICAXE-08M2 (or 08M) processor
H1	Header, male, right angle, 3 pins by 2 rows
—	Ribbon cable, 3 feet, 6 conductors
—	Plastic test tube
—	Finishing nail, 2 inches long
—	Two IDC female connectors, 3 pins by 2 rows
—	Two 5-pin straight male header (0.23″ and 0.32″ mating lengths)

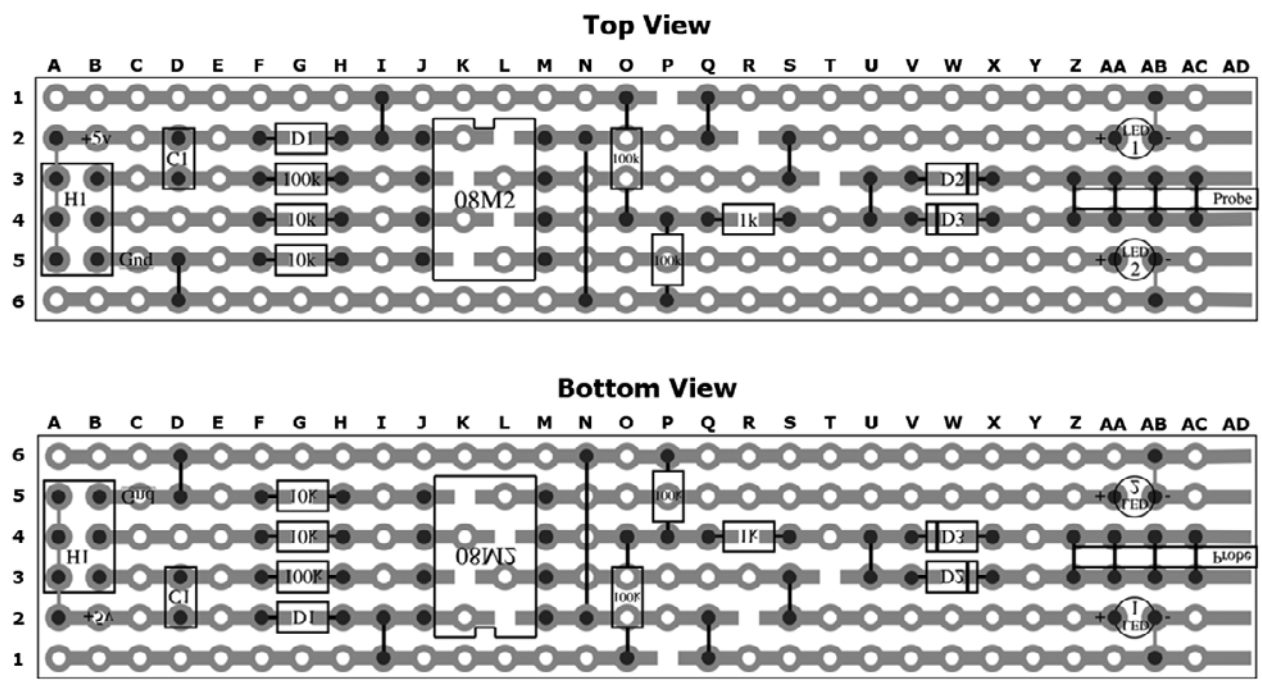


Figure 6-7 Stripboard layout for logic probe project

I also want to clarify what turned out to be the most difficult aspect of the project. The layout shows the board as having six traces, which makes it 0.6 inches (15.2 mm) wide. The inside diameter of the test tube is also approximately 0.6 inches, so when I first began the project I naively thought everything would fit together perfectly. However, I ran into several problems along the way. Figure 6-8 is a close-up photo of the top of the stripboard that finally worked, and Figure 6-9 shows the bottom of the same board. As you can see from the pattern of partial holes along the top and bottom edges, the board ended up to be far less than 0.6 inches wide; in fact, it tapers from about 0.525 inches (13.3 mm) at the connector end down to 0.475 inches (12.1 mm) at the probe end. This is because the test tube tapers along its length—a fact that I failed to notice before I started!

In addition to the unanticipated taper, the board is narrower than I intended because three factors prevented my placing the board at the midline of the test tube. First, I wanted the "probe" (which is actually an ordinary finishing nail) to protrude from the center of the bottom of the test tube. In Figure 6-8, you can see that the nail is soldered on top of the board, so the board has to be a little below the midline. Also, I wouldn't have been able to squeeze the 08M2 into the test tube in that position anyway. For a while, I considered soldering the 08M2 directly to the board, but I finally got it to fit by "lowering" the socket—I'll explain how in the next section when we actually construct the stripboard.

Constructing the Stripboard

The most important and difficult part of constructing the logic probe is getting the board to fit all the way down into the tapered test tube. Of course, you can take the easy way out and skip the test tube altogether—the probe should function perfectly without it. But if you're up for the challenge, I suggest that you read the following procedures fully before you begin; then start by

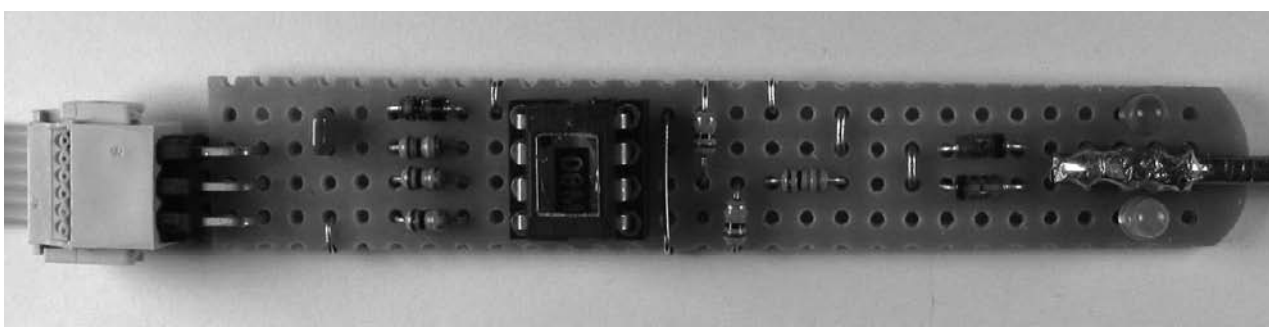


Figure 6-8 Close-up view of the top of the completed logic probe

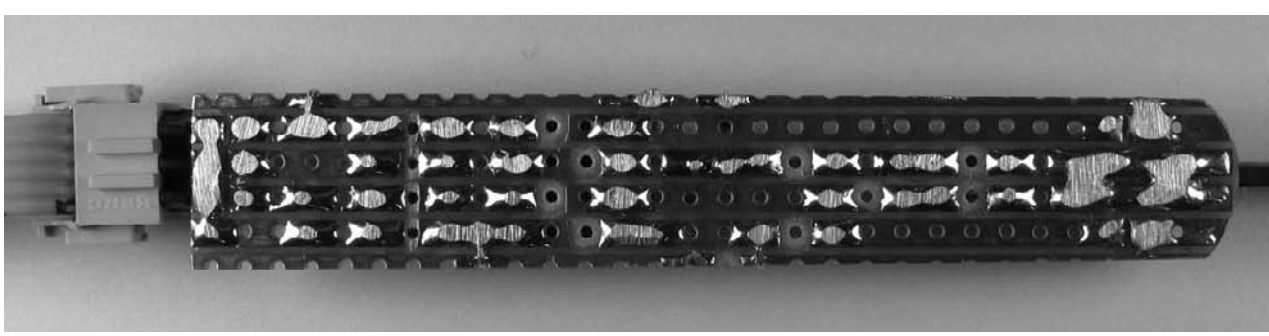


Figure 6-9 Close-up view of the bottom of the completed logic probe

shaping the board to fit in the test tube. As a general guideline, you want to sand or file the taper on the board so that a little more than half the hole remains at positions A1 and A6 and the holes at positions AC1 and AC6 are almost entirely sanded away, as shown earlier in Figure 6-8. Once the board is prepared, you're ready for the assembly process.

1. Sever the traces on the bottom of the board as indicated in the layout in Figure 6-7.
2. Round the probe end of the board by sanding or filing it until it matches the curve of the bottom of the test tube.
3. From the bottom of the board, use a 1/16-inch (1.5-mm) drill bit to enlarge the eight holes for the machine pin socket. This will enable the socket to sit low enough for the 08M2 to fit into the test tube. Solder the socket in place.
4. Solder the nail to the stripboard as follows: Use a pair of diagonal cutters to remove the head of the nail so it lies flat on the stripboard. Cut four bare jumper wires about two inches long, bend each one into a "U" shape, and fit it over the top of the nail and down through the pair of holes on each side of the nail. On the bottom of the board, use a pair of pliers to twist the ends of each jumper together tightly so that the nail is held firmly in place. (Make sure the nail is positioned along the midline of the stripboard.) Solder the jumpers to the *top* of the nail first. (This produces so much heat that it would melt the solder on the bottom of the board if you did it the other way around.) Let the nail cool; then solder the jumpers to the bottom of the board and snip off the excess leads.
5. Drill a 3/32-inch (2.5-mm) hole in the center of the end of the test tube—there's a small dot on the tube at just the right place. Make sure the stripboard fits all the way down the tube and the nail protrudes through the hole. Adjust the width of the board and the size of the hole if necessary.
6. Test-fit the 3 × 2 right angle male header. If necessary, sand or file the end of the stripboard so that the black plastic of the header just overhangs it, rather than sitting on top of the board (see Figure 6-8). Solder in the header and the jumper wire on the bottom of the board that spans from A2 to A5.
7. Insert one of the 3 × 2 IDC female connectors onto the male header pins. Also insert an 08M2 into the socket; then test-fit the board again to be sure everything will fit properly inside the test tube. Make any necessary final adjustments.
8. Solder all the remaining parts (*except* for the two LEDs) in place. At the six places along the edges of the board where a jumper wire or resistor lead needs to be soldered at a location that has less than a complete hole, crimp the lead around the edge of the board to hold it in place. The three 0.1-inch (2.54-mm) jumper wires are easy because in each case, one lead can be bent to touch the other lead. For the two resistors and the longer jumper wire, make sure the lead at the edge of the board does not come in contact with the adjacent trace on the bottom of the board.
9. Figure 6-10 shows the completed probe being used to test a circuit. I happen to be left-handed (as you can see in the photo), which is why I positioned the red and green LEDs as I did—it just makes more sense to me that the *high* LED is above the *low* LED. If you're right-handed and that sort of thing matters to you, you may want to reverse the position of the two LEDs before soldering them in place. (If so, you will also need to reverse constant declarations in the program.) Before soldering each LED in place, bend and snip each

negative lead so that it just contacts the adjacent trace as shown in Figure 6-7 and Figure 6-9.

10. Clean the flux from the bottom of the board and allow it to dry.
11. Inspect the board carefully for accidental solder connections and other problems.

Okay, we're ready for the final step: assembling the ribbon cable and connecting the logic probe. If you don't have experience with ribbon cable and IDC connectors, take a look at the tutorial on my website (www.jrhackett.net/progcable.shtml). When you're ready to proceed, just follow these steps.

1. Drill a 5/16-inch (8-mm) hole in the center of the test tube cap.
2. Attach a 3 × 2 IDC connector to one end of the six-wire ribbon cable.
3. Pass the other end of the cable through the hole in the test tube cap.
4. Attach another 3 × 2 IDC connector to the other end of the cable. Be sure to orient the connector the same way as the first one. (If you stretch the cable out so that it's lying flat on one side, both connectors should either point up or point down.)
5. Attach the connector that is on the interior side of the cap to the stripboard connector so that the cable exits from the bottom of the connector. Slide the stripboard fully into the test tube, and insert the cap into the end of the test tube.

To test the logic probe, first program an 08M2 with the LogicProbe.bas software and then install the 08M2 in the socket. Next, connect the ribbon cable to the supply rails of a breadboard. At this point, you may not know which way is the correct

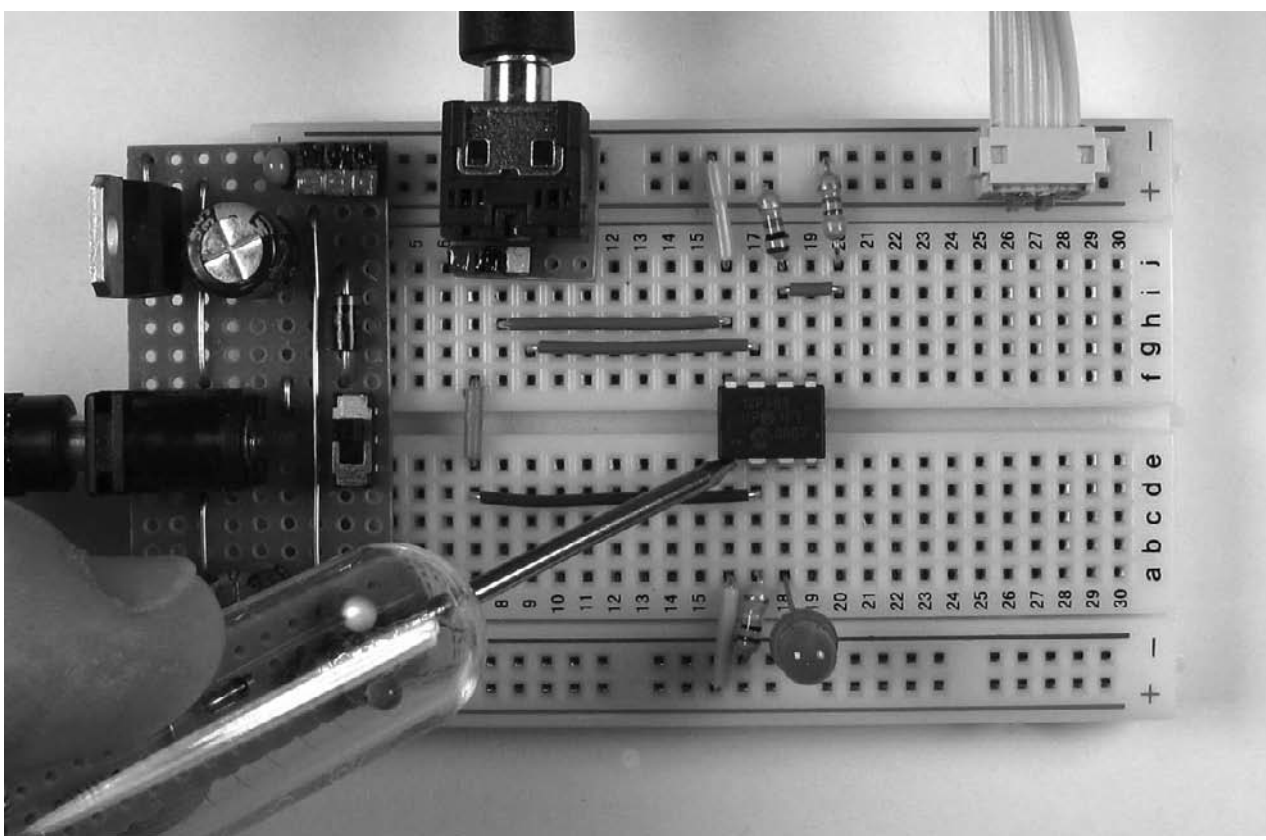


Figure 6-10 Testing a circuit with the completed logic probe

orientation, but the logic probe is protected from any reverse-polarity damage, so we can figure that one out experimentally. Using two 3-pin sections from the two 5-pin male headers, connect the ribbon cable to the power rails of a breadboard. If the power supply is reversed, the LEDs won't light, regardless of which point in the circuit is probed. If that's the case, simply rotate the connector on the power rails 180 degrees and the probe should function properly. Once you know which way is the correct orientation for the connector, you may want to label or paint it to identify +5V and Ground.

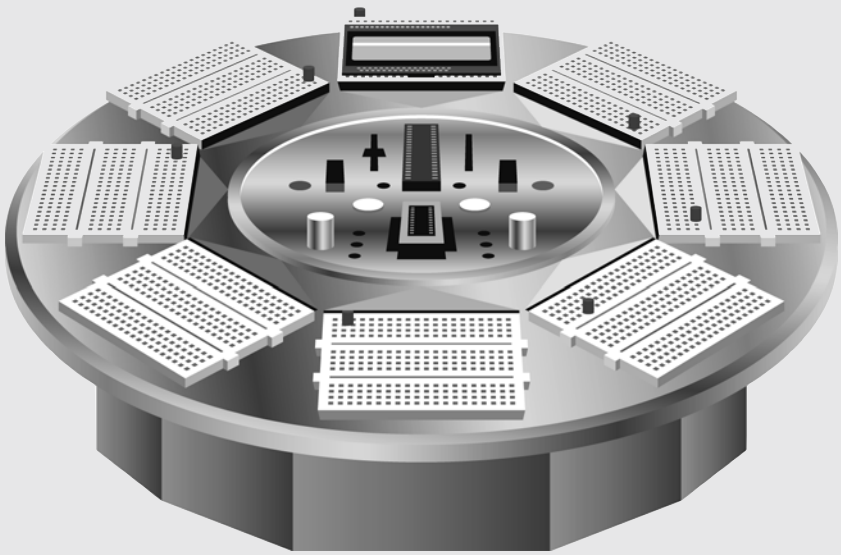
To actually use the logic probe, just attach it to any breadboard circuit you have on hand. When power is applied to the board, the probe should be able to clearly identify the three conditions we discussed: *high, low*, and *open* connections. In addition, the probe can tell you when an output is producing a repetitive waveform. To see how it responds, download the following code snippet to any PICAXE processor, touch the appropriate output pin with the probe, and see what happens:

```
do
  high C.0
  low  C.0
loop
```

This page intentionally left blank

PART TWO

PICAXE Peripheral Projects



This page intentionally left blank

CHAPTER 7

Introduction to the PICAXE-20X2 Processor

Now that we have covered some of the basics of working with the PICAXE M2-class processors, we're almost ready to begin developing our own stand-alone peripheral devices for use in our PICAXE projects. However, we have one more "preparatory" project to complete before we delve into our first I/O peripheral project. In order to be able to develop and test our peripheral projects, we need a "master processor" circuit to function as the host system for our various I/O devices. I have chosen the PICAXE-20X2 processor for this purpose. It's the newest and smallest chip in the X2 class of PICAXE processors, and it supports all the advanced features of the other, more powerful X2-class processors.

Advanced Features of the 20X2 Processor

The PICAXE-20X2 implements an impressive assortment of hardware and software enhancements. A quick look at the pin-out presented in Figure 7-1 will give you some idea of the range of the 20X2's capabilities. It would take more than one volume to fully explore all the possibilities, and by the time you did, there would probably be a 20X3 on the market!

In this introductory chapter, I'm going to briefly present several of the 20X2's advanced features that I have found to be the most useful in my projects. In subsequent chapters in Part Two, we'll explore many of these features in more detail as they are needed. For now, let's take a brief look at some of the 20X2's major hardware and software features in the following areas:

- Supply voltage range
- Operating frequency range
- General-purpose variables
- Flexible I/O pins
- Internal pull-up resistors
- Analog-to-digital conversion
- Serial I/O
- Background timing and interrupt processing

Supply Voltage Range

The 20X2 has the same supply voltage range of the M2-class processors; it can run reliably with a supply voltage anywhere between 1.8V and 5V. This impressive range opens new possibilities for battery-powered X2-class projects.

Operating Frequency Range

Unlike the other X2-class processors, the 20X2 does not use an external ceramic resonator to set its operating frequency. However, it does have the widest range of internal operating frequencies of all the PICAXE processors. The 20X2's *setfreq* command can be used to set its internal resonator to any one of the following frequencies: 31kHz,

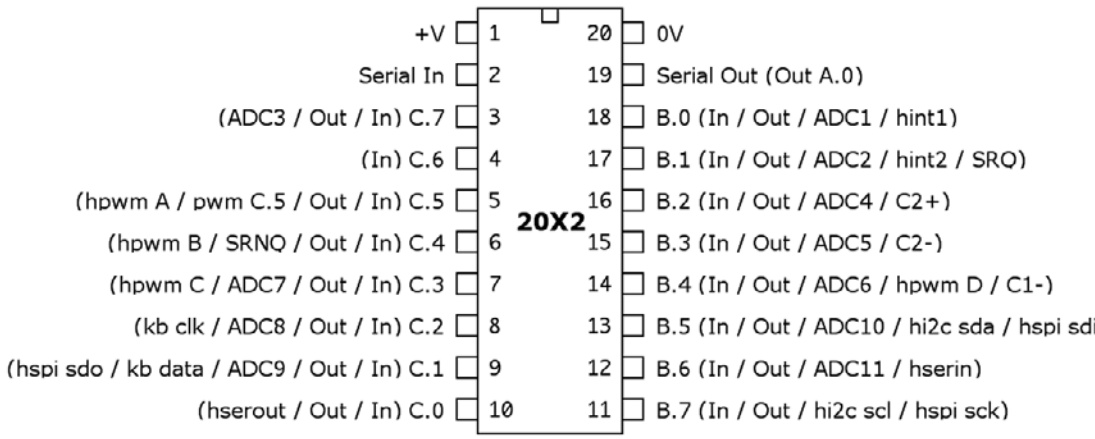


Figure 7-1 PICAXE-20X2 pin-out

250kHz, 500kHz, 1MHz, 2MHz, 4MHz, 8MHz, 16MHz, 32MHz, and 64MHz. An operating frequency of 64MHz is obviously impressive; the fastest speed for the M2-class chips is 32MHz, so the 20X2 can execute a program twice as fast as any M2-class processor.

What's not so obvious is that an operating speed of 31kHz is also impressive, because power consumption is directly related to processor speed. For example, a 20X2 operating at 16MHz consumes about 250 times the power than it would if it were operating at 31kHz. Since many programs can run effectively at 31kHz, this is an impressive power savings. When we combine low operating speed with lower supply voltages, the power savings are even more impressive. A 20X2 operating at 3V and 31kHz can actually draw as little as 16µA, which at 3V is only 48µW—an amazingly small amount of power!

General-Purpose Variables

You may remember that the M2-class processors each have 28 bytes of RAM dedicated to the storage of general-purpose variables. On the 20X2, this figure is doubled to 56 bytes (byte variables b0 to b55, or word variables w0 to w27, or any combination of the two). I've never written a program that needed anywhere near that amount of variable storage, but it's nice to know it's there. In the unlikely event that your program needs more than 56 general-purpose variables, the 20X2 also contains another 72 bytes of storage variables (at locations 56 through 127) that can be accessed with the *peek* and *poke* commands.

Flexible I/O Pins

We have already discussed the M2-class pin naming convention (i.e., the standard *port.pin* format of the pin names, such as C.4, B.0, etc.) and the fact that almost every I/O pin on the M2-class processors can be individually configured to be either an input or an output pin. These powerful and flexible features were originally introduced on the PICAXE X2-class processors, so the 20X2 implements them as well. On the 20X2, the C.6 pin is fixed as an input and the A.0 pin is fixed as an output; all other I/O pins are bidirectional. In addition, the *dirsB* and *dirsC* special-function variables are used in the same manner that we have already discussed.

As a result, the vast majority of your experience in working with the M2-class processors will easily transfer to the X2-class processors. Specifically, if you compare the pin-out of the 20M2 (Figure 4-4) with that of the 20X2 (presented earlier in Figure 7-1), you can see that the majority of the special pin functions on the 20M2 are in exactly the same position on the

20X2. The exception to this rule is the 20M2's touch inputs, which at this point are unique to the M2-class processors. If a project doesn't require touch inputs, you could easily begin with a 20M2 processor and later move up to a 20X2 processor if you needed more memory or a specific advanced function. Other than upgrading the processor, your project's hardware could remain virtually unchanged.

Internal Pull-up Resistors

On each of the X2-class processors, some of the I/O pins have internal "pull-up" resistors. When your program enables one of these resistors, its associated I/O pin is pulled high through an internal resistor inside the processor. This feature makes it possible to omit an external resistor when configuring an I/O pin as an input. For example, if your project doesn't use *pinC.6* (which is fixed as an input), you could enable its internal pull-up resistor rather than adding an external resistor.

The syntax of the *pullup* command is *pullup mask*, where *mask* is an eight-bit variable or constant whose bits determine which pull-up resistors will be enabled. On the 20X2, the following pins have internal pull-up resistors: B.7, B.6, B.5, B.1, B.0, C.7, C.6, and C.0. This list is in the same order as the bits of the *mask* parameter. In other words, bit 7 of the mask controls the resistor for *pinB.7*, bit 6 of the mask controls the resistor for *pinB.6*, etc. For example, consider the following code fragment:

```
dirsB  %11110000   ' configure pins
                     B.7… B.4 as out-
                     puts and B.3… B.0
                     as inputs
dirsC  %00001111   ' configure pins
                     C.7… C.4 as inputs
                     and C.3… C.0 as
                     outputs
pullup %00011110   ' enable the pullups
                     on inputs B.1,
                     B.0, C.7 and C.6
```

Analog-to-Digital Conversion

The X2 processors have more powerful ADC capabilities than the M2-class chips, but the ADC functions on the X2 chips are also somewhat more complex. For example, whenever you use an ADC command on an M2 processor, the referenced pin is automatically configured as an analog input (as we saw in the previous chapter). However, on the X2 chips, you need to first issue an *adcsetup* command to specify which input(s) you want to configure for ADC use. (For details, see the *adcsetup* docs in Part II of the manual.) If you look back at the pin-out presented earlier in Figure 7-1, you will see that the 20X2 can implement as many as 11 ADC inputs. Someday I'm going to figure out what do with that amazing capability.

The X2-class processors also support the *calibadc* and *calibadc10* commands that we discussed in the previous chapter. We will experiment with that capability later in this chapter after we have set up our master processor circuit.

Serial I/O

All M2-class processors are capable of transmitting and receiving serial data. However, there is a limitation of M2-class serial I/O that needs to be taken into account in any program. If serial data is received while an M2 program is busy executing some other task, the data will be missed altogether. There are various ways of dealing with this issue, but they tend to be complicated. On the X2 processors, this problem is completely eliminated. All X2-class chips implement an *hserin* command that enables a program to receive serial data in the background while it tends to other tasks. When the program is ready, it can access the data that has been automatically received and process it accordingly. The *hserin* command can be a little complicated to set up correctly, but it's well worth the effort to do so. We'll tackle that task when we design and construct a serialized LCD display in Chapter 10.

Background Timing and Interrupt Processing

The X2-class *hintsetup* command provides true (i.e., nonpolled) hardware interrupt capability. On the 20X2, pins B.0 and/or B.1 can be configured to trigger an interrupt on either the rising or falling edge of an input pulse, even when the processor is dozing or sleeping. In addition, the *settimer* and *setintflags* commands can be configured to trigger an interrupt whenever the 20X2's internal hardware timer overflows. Among other things, this capability enables the 20X2 to maintain an accurate internal real-time clock. In Chapter 13, we'll take advantage of these features when we develop our countdown timer project.

Project 7
Implementing the 20X2 Master Processor Circuit

Now that we have a basic understanding of some of the major hardware features of the 20X2 processor, we're ready to set up the master processor circuit that we'll be using to develop and test the I/O projects that we'll cover in the remainder of Part Two of this book. All we really need is a fair amount of breadboard space to provide the necessary flexibility for working with the various projects that we will be developing. With that requirement in mind, I opted for the three-breadboard setup shown in Figure 7-2. As

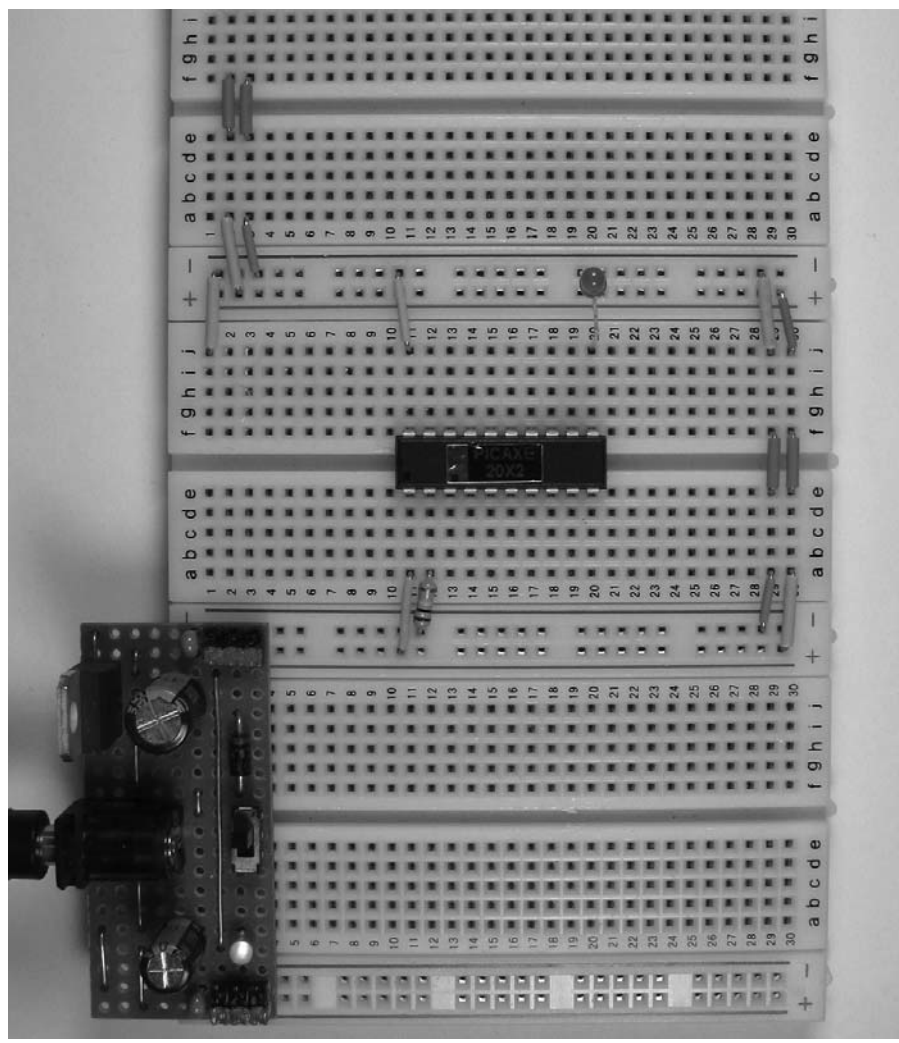


Figure 7-2 The master processor breadboard layout

you can see, this arrangement provides ample space for the possibility of working with more than one peripheral project at the same time. As a general rule, we'll use the lower breadboard to develop our input projects and the upper one for the output projects.

You may have noticed that I didn't include a programming adapter connection in the breadboard layout presented in Figure 7-2. It was an intentional omission because we need to take a certain complication into consideration before we settle on an appropriate programming adapter for our master processor circuit. Since some of our peripheral projects will contain their own processors, we'll need to be able to switch back and forth between programming the peripheral processor and programming the master processor. We could certainly do that with just one programming adapter, but it would be much more convenient to provide the master processor with its own programming adapter and just use our USBS-PA3 adapter for our peripheral processors. Also, since the master processor is in the center of our three-breadboard layout, it would also be more convenient if its programming adapter could be located at the left side of the breadboard rather than at its top or bottom edge. (This is also why I didn't place the power supply to the left of the master processor.)

By now you have probably guessed that we're about to build another programming adapter—if so, you're absolutely correct. For a while, I considered naming it the YAPA-3X2 (for "Yet Another Programming Adapter"), but I finally decided to call it the USBS-PA3X2—you can probably figure out how I arrived at that unwieldy acronym! In any case, let's actually build the USBS-PA3X2 before we go any further.

Constructing the USBS-PA3X2 Programming Adapter

In order to understand the details of constructing the USBS-PA3X2 adapter, I think it will be helpful to have a clear idea of how it will be used in our breadboard circuit. Figure 7-3 shows a close-up

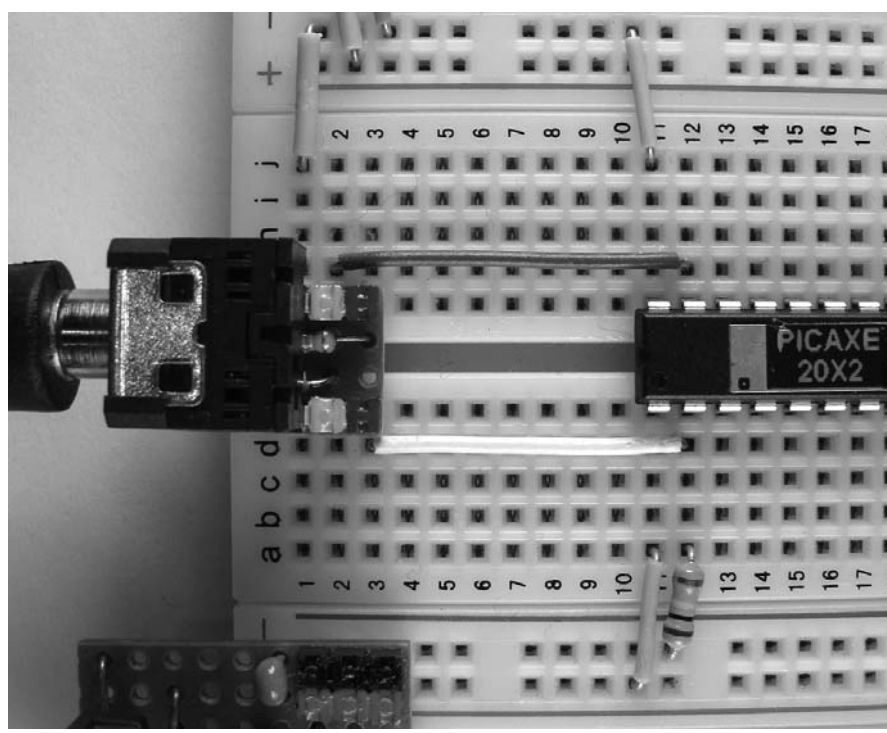


Figure 7-3 Close-up of the USBS-PA3X2 installed on the breadboard

photo of the master processor circuit with a completed USBS-PA3X2 installed on the left end of the middle section of the breadboard so that you can clearly see how the wiring connections are made to the serin and serout pins of the 20X2. In the close-up, you can't see the third pin (the ground connection) on each side of the adapter because it's underneath the overhang of the stereo connector. The jumper wire in the upper-left corner of the photo connects one of the two ground pins to the ground rail on the breadboard. (You can connect whichever ground pin is more convenient for any given project.)

Figure 7-4 presents the stripboard layout for the USBS-PA3X2 adapter. The only unusual feature of the layout is that there is a small "notch" in the stripboard—the section that would have contained holes E5 and E6 has been cut away so that the jumper that runs from the adapter to the serout pin on the 20X2 can sit as close as possible to the chip (see the close-up presented in Figure 7-3). The notch is not required for the adapter to function properly; you can omit it if you prefer.

The required parts are too few to warrant a formal parts list: a small piece of stripboard; a 10k, 1/6W resistor and a 22k, 1/6W resistor; two 3-pin sections of reverse-mountable male headers; and a high-profile stereo connector. If you don't already have them on hand, they're all available on my website. The two 3-pin headers are spaced so that they will straddle the center divider when they are inserted into the breadboard. When the stripboard is assembled, the three pins of the stereo adapter will be inserted in holes A3, C1, and E3 and the stereo adapter will cover the two header pins that provide the Ground connection at A4 and D4. The header pins at A5 and D5 are for the connection to the serout pin of the 20X2, and the pins at A6 and D6 are for the connection to the serin pin. In each case, you will use only one of the two possible connections. However, the availability of the duplicate pins allows the programming adapter to be used either in front of the processor or behind it. Also, the double row of header pins makes the adapter more stable when it's inserted into the breadboard. As usual, read through the entire list

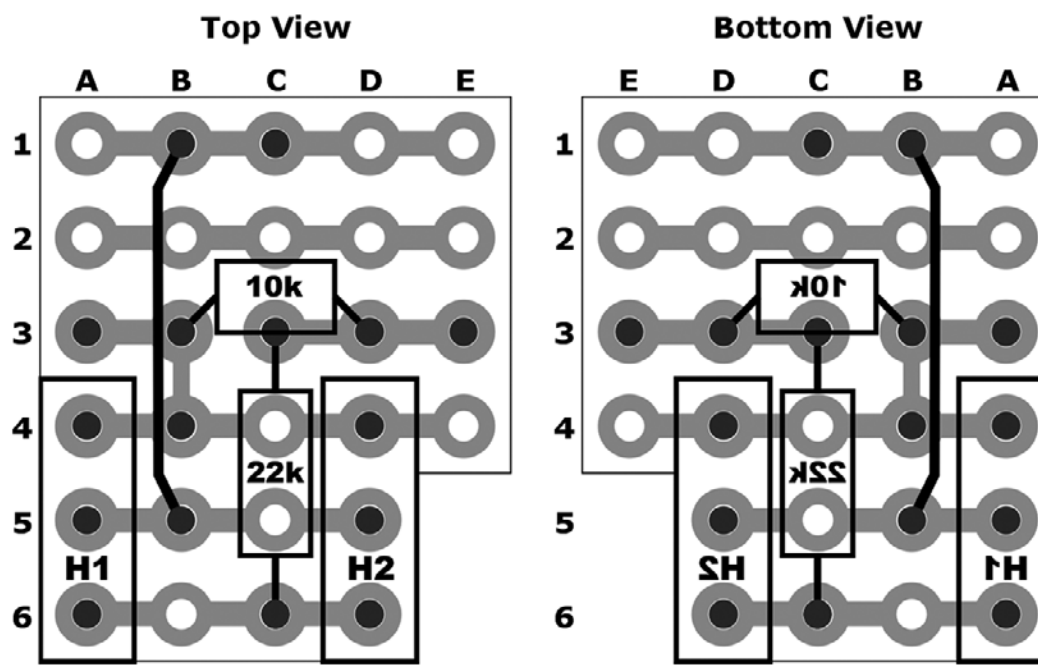


Figure 7-4 USBS-PA3X2 stripboard layout

of assembly instructions that follows to be sure you understand the entire procedure before assembling the board.

1. Cut and sand a piece of stripboard to the required size (six traces with five holes each).
2. (Optional) Cut and sand a small "notch" in the stripboard that removes holes E5 and E6, as shown in the layout presented earlier in Figure 7-4.
3. Using a 3/64-inch (1.2-mm) drill bit, enlarge the holes at A1, A3, C1, E1, and E3. If you can't locate a 3/64-inch drill bit, 1/16 inch (1.5 mm) should also work. (As an aside, Dremel makes a seven-piece set of small drill bits ranging in size from 1/32 inch to 1/8 inch. They are readily available at hardware stores and home improvement centers. If you don't have one already, you might want to take a break and run out to get one. They can be helpful for stripboard work.)
4. Sever the trace on the bottom of the board between holes B3 and C3.
5. On the bottom of the stereo connector, use diagonal cutters to snip off the plastic post that would have sat at hole C3. Also snip off the pin that would have sat at hole B3. Cut both of them as close to the body of the connector as possible.
6. Test-fit the connector into the stripboard. (Its three remaining pins should be inserted in holes A3, C1, and E3.)
7. Insert the 22k resistor leads into holes C3 and C6; solder and snip the leads on the bottom of the board.
8. Insert the 10k resistor leads into holes B3 and D3. Position the resistor so that it sits slightly above hole C3, as shown in the layout (Figure 7-4). On the bottom of the board, bend the lead from B3 to B4 and snip it so that it just reaches B4. Press the lead flat against the bottom of the board; solder the leads at B3, B4, and D3; and snip the lead at D3.
9. Insert the ends of an *insulated* jumper wire at holes B1 and B5. Solder and snip the leads. (In the layout, the wire is offset to make it clear that it has no contact with the resistor lead at B3; since the insulation will accomplish that, the wire can actually run straight between B1 and B5.)
10. Sand or file the bottom of the board to remove any sharp edges. Remove any unnecessary "high spots" so the board will be able to be inserted fully into a breadboard when it's being used in a project.
11. Insert the short ends of two 3-pin pieces of reverse-mountable male headers into a breadboard to support them during the next step. (The two headers should just straddle the gap in the center of the breadboard.)
12. Invert the stripboard and mount it on the long ends of the male headers (which should protrude through holes A4–A6 and D4–D6); solder the six pins in place.
13. Remove the stripboard from the breadboard; snip the short ends off the header pins and file them smooth.
14. From the top of the board, insert the pins of the stereo connector through holes A3, C1, and E3. Invert the board and place the top of the stereo connector on a flat surface. Solder the three pins in place, snip any excess, and file the pins to remove any sharp edges.
15. Clean the flux from the bottom of the board and allow it to dry.
16. Inspect the board carefully for accidental solder connections and other problems.

Testing the USBS-PA3x2 Programming Adapter

When you have completed the programming adapter, use the photo presented earlier in Figure 7-3 as a guide for assembling your master processor breadboard circuit. We'll be using this circuit throughout Part Two of this book as we develop and test our peripheral I/O devices. For now, you may want to set up a simple "Hello World!" circuit to make sure your USBS-PA3X2 adapter is functioning correctly, or you can try the following experiment, which demonstrates the use of the 20X2 *calibadc10* command.

Experiment 1: Determining Battery Supply Voltage

In this experiment, we're going to revert back to the 4.5V battery supply that we used in our original "Hello World!" project. All you need to do is temporarily remove the 5V regulated supply from your master processor setup and replace it with the battery supply. The software for our experiment is presented in Listing 7-1—as usual, it can be downloaded from my website.

The *bintoascii* and *sertxd* commands are the same ones we used in Experiment 2 of the previous chapter, so there are only two questions to answer about the BattMon.bas program: How did I arrive at the mysterious value of 52,378? And, why do we double the value of the *batV* variable before sending it on to the terminal window? To answer those questions, let's start by stating the "English" version of the basic relationship among the variables (presented in the following illustration):

Step 1: $$\frac{Vref}{Vdd} = \frac{ADCval}{ADC\,\max}$$

Step 2: $$\frac{1.024}{Vdd} = \frac{ADCval}{1023}$$

Step 3: $$ADCval \bullet Vdd = 1047.552$$

Step 4: $$Vdd = 1047.552 / ADCval$$

Step 5: $$100Vdd = 104755 / ADCval$$

Step 6: $$50Vdd = 52378 / ADCval$$

Step 7: $$50Vdd = answer$$

Step 8: $$100Vdd = answer \bullet 2$$

The value of the internal reference voltage is to the value of the battery voltage as the result of the *calibadc10* command is to the maximum ADC value. Next, we'll take a look at the algebraic version of the same statement and then simplify the results. In the following paragraph, I'll explain each step of the calculations.

Step 1 is just the algebraic equivalent of the relationship stated earlier. Steps 2, 3, and 4 simply substitute in the values we know and rearrange the terms of the equation to solve for *Vdd*, which is what we want to know. In step 5, we're multiplying both sides of the equation by 100 (and rounding the result to an integer) in preparation for tricking the compiler into working with our decimal value. This is the same approach we took in the previous chapter. Our integer answer will be 100 times the actual value that we want (e.g., if *Vdd* is 4.38V, we'll end up with 438 and then insert the decimal point where we need it in the result).

We would be finished at this point, except for one little problem: The compiler can't deal with an integer as large as 104,755 (the largest allowable word variable is 65,535, or 2^{16}–1). We solve this problem in step 6 by dividing both sides of the equation by 2 and making a mental note that we will need to adjust for that later. In step 7, *answer* is what we actually calculated in the program, and

LISTING 7-1

```
' ================================ BattMon.bas ================================
'   Program runs on a PICAXE-20X2 and measures the
'   analog value of its three-cell battery supply.
' ============================================================================

' === Constants ===
  symbol abit = 500                          ' used in pauses
  symbol LED  = B.7                          ' debugging LED on B.7

' === Variables ===
' Variables b0 through b4 are directly used in
' the bintoascii command - can't use them here.
  symbol ADCval = w3                         ' ADC result
  symbol batV   = w4                         ' supply voltage

' === Directives ===
  #com 3
  #picaxe 20X2
  #no_data                                   ' reduces download time
  #no_table                                  ' reduces download time
  #terminal 9600                             ' open terminal window

' ============================ Begin Main Program ============================
do
  high LED
  pause abit
  calibadc10 ADCval
  batV = 52378 / ADCval                      ' see text
  batV = batV * 2                            ' see text

  low LED
  pause abit
  bintoascii batV, b4, b3, b2, b1, b0
  sertxd (b2,".", b1, b0," volts",cr,lf)
loop
```

step 8 is our reminder that we need to double that result to get back to 100*Vdd* (which is also done in the program). All that remains is to transmit the result to the terminal window, with the decimal point inserted in the correct place to (in effect) divide by 100 and get back to the actual value of *Vdd*. Run the program and measure the battery voltage with a multimeter to confirm the accuracy of the results—they should be pretty close.

As you can see, the *calibadc* and *calibadc10* commands can be helpful when working with a battery-powered project. If these commands weren't available, we would need to include some sort of external voltage reference circuit in order to be able to accomplish what we just did entirely with software. Finally, don't forget that all the M2 processors also support the two ADC calibration commands.

This page intentionally left blank

CHAPTER 8

Infrared Input from a TV Remote Control

In this chapter, we're going to experiment with the commands that enable PICAXE processors to receive and decode the infrared (IR) signals produced by standard TV remote controls. Once we have an understanding of the concepts involved, we'll develop our first peripheral project, an 08M2-based stand-alone device that can recognize the TV-remote signals and send the data on to our 20X2 master processor circuit. In addition to the built-in IR commands, there are two other methods of implementing infrared I/O functions that are based on very different approaches. Before we get into the details of our first experiment, let's take a brief look at PICAXE IR capabilities in all three of these areas.

Reception and Transmission of Standard TV IR Signals

All PICAXE processors support built-in commands that greatly simplify the reception and transmission of infrared signals. These commands are based on the Sony infrared remote control (SIRC) protocol. Even if you don't own a Sony TV, virtually all the inexpensive "universal" remotes on the market today support the SIRC protocol and can be used with the PICAXE IR commands. All you need to do is configure the remote to transmit Sony TV signals. You may need to try more than one of the Sony codes listed in the remote's documentation, but that's relatively easy to do.

SIRC signals are modulated onto a 38kHz carrier wave. Fortunately, several inexpensive IR decoders also function at 38kHz, so the hardware requirements are minimal. The Panasonic PNA4602M decoder that we will be using (available on my website and elsewhere) includes visible-light filtering and automatic demodulation of the SIRC signals. The 4602's output pin can be directly connected to a PICAXE input so the circuitry is quite simple. The PNA4602 pin-out is shown in Figure 8-1, and its datasheet is available on my website (www.jrhackett.net/datasheets/PNA4602M.pdf).

On all current PICAXE processors, the relevant commands are *irin* for IR reception and *irout* for IR transmission. *Irin* can be used with any input pin, and *irout* can be used with any output pin. However, the older M-class processors were limited to a single specific pin for each command. For example, on the 08M, IR input could only be received on input 3 (pin 4). In case you plan to use an 08M processor for the experiments and project in this chapter, I have used that specific pin so that either processor (08M or 08M2) will work with the same hardware setup. (Of course, you will need to download the "M" version of each program.)

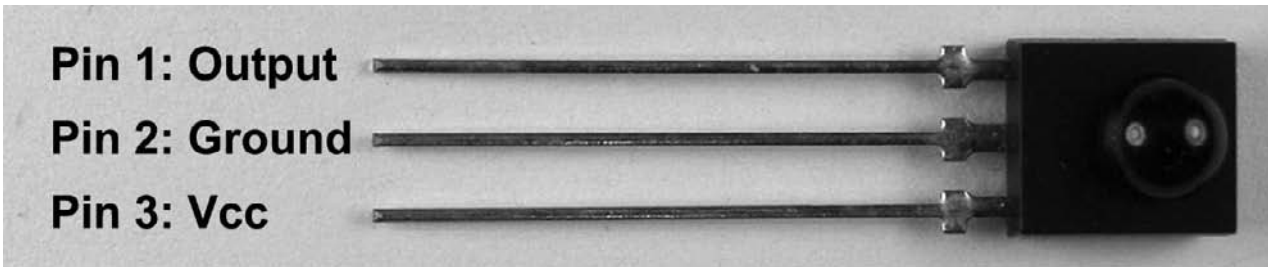


Figure 8-1 PNA4602 pin-out

The *irin* command also includes an optional "timeout" feature that allows your program to continue on to other tasks if a valid IR signal is not received within a specified period. We won't be using that feature in this chapter; if you are interested in more information on it, see the *irin* documentation in Part II of the manual.

IR-Based Serial Communications

All PICAXE processors also support a *pwmout* command that is usually used to implement variable speed control for DC motors. (We'll use *pwmout* for that purpose in Part Three when we implement the DC motor control system for the Octavius robot.) Essentially, the *pwmout* command generates a continuous pulse-width-modulated (PWM) output in the background while a PICAXE program is executing. In other words, *pwmout* produces a rapidly oscillating signal with a specified frequency and duty-cycle. So, *pwmout* is an easy way to produce the 38kHz carrier wave that's received by the PNA4602M. In order to establish an infrared serial communications link between two PICAXE processors, all it takes is a simple one-transistor circuit to modulate a 38kHz PWM signal with a standard serial output signal and use it to drive one or more IR LEDs. On the other end of the communications link, a second processor uses an IR detector to demodulate the incoming signal and convert it back to the standard serial protocol.

If you're interested in learning more about this approach to IR serial communication, three installments of the *PICAXE Primer* column in *Nuts and Volts* (Oct. 2008, Dec. 2008, and Feb. 2009) cover the topic in detail. If you don't already subscribe, you may want to consider doing so, since subscribers have full access to all the online back editions of the magazine.

Simple IR Object-Detection

The third application of the PICAXE IR capability also involves the *pwmout* command. In this case, IR signals are simply used to detect the presence of an object in the path of an infrared beam. There are two different approaches to implementing this capability. The first method involves two separate PICAXE circuits: one for IR transmission and the other for IR reception. For example, an 08M2 circuit that continuously transmits a 38kHz IR "beam" can be placed on one side of a hallway, with another 08M2 "receiver" circuit on the other side. Whenever a person (or other object) moves through the IR beam, the receiver can detect the momentary break in the beam.

In the second approach to object detection, both the IR transmission and the IR reception are implemented on the same processor. This type of circuit frequently functions as an object detector for a robot. Essentially, a short 38kHz IR signal is transmitted and the processor immediately "listens" for an echo. Objects in front of the IR detector within four or five feet can be reliably detected using this approach.

Experiment 1: A Simple TV-IR Input Circuit

For our first experiment, we're going to implement a simple circuit that's capable of receiving and decoding SIRC signals from a TV remote control. The schematic is presented in Figure 8-2, and the necessary parts list is again too brief to warrant a table: PICAXE-08M2, PNA4602M, and a debugging LED (resistorized or standard).

The breadboard circuit is shown in Figure 8-3. As you can see, I set up the circuit on the lower section of our master processor board. While that's not really necessary for this experiment (since it only involves the 08M2), in our second experiment we're going to connect the 08M2 to our master processor, so it's easier to begin in the right place. With that in mind, you may want to duplicate my setup to facilitate making the connections we will soon implement.

Two other points are worth mentioning. First, the programming adapter that I used for the 08M2 is functionally identical to the USBS-PA3X2, but slightly longer. (Actually, it was an early prototype for the USBS-PA3X2.) When we get to Experiment 2, you can either move your USBS-PA3X2 back and forth as you program the two chips or make a second USBS-PA3X2. That's the more convenient approach, especially if you have two USB cables as shown in the photo. That way, when we get to Experiment 2, there's no cable swapping involved as you move back and forth between programming the two processors. Of course, you can also use our original USBS-PA3 adapter as well.

Second, it may not be clear in the photo, but pin 1 of the 08M2 is at the lower-left corner of the chip. As a result, the 08M2's programming adapter is actually behind the processor rather than in front of it. That's one of the advantages of the USBS-PA3X2 that I mentioned earlier—the required connections are just as simple either way. Finally, I haven't included the 08M2's programming interface circuitry in the schematic because it's contained within the USBS-PA3X2 adapter. This will be a standard omission in the remainder of

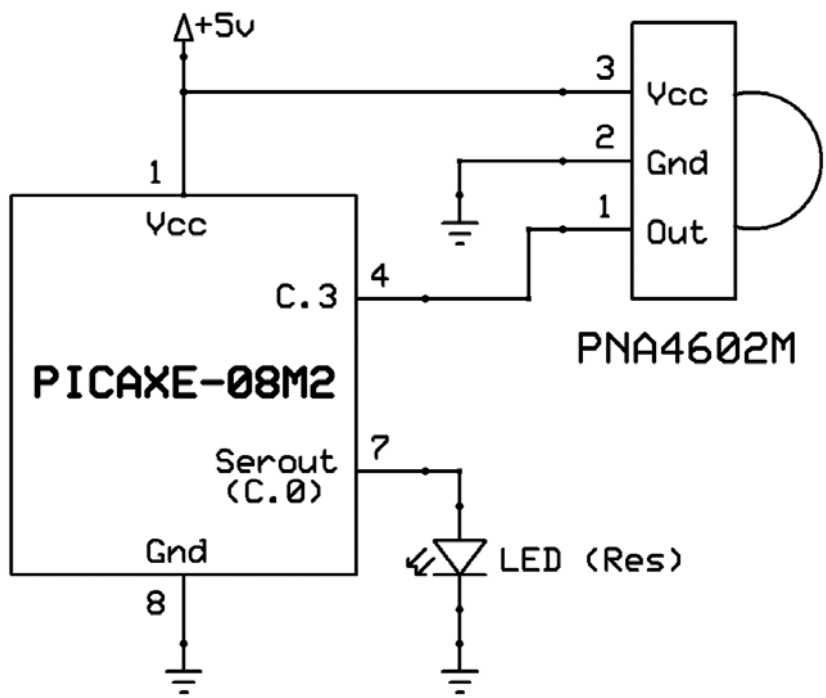


Figure 8-2 Schematic for Experiment 1

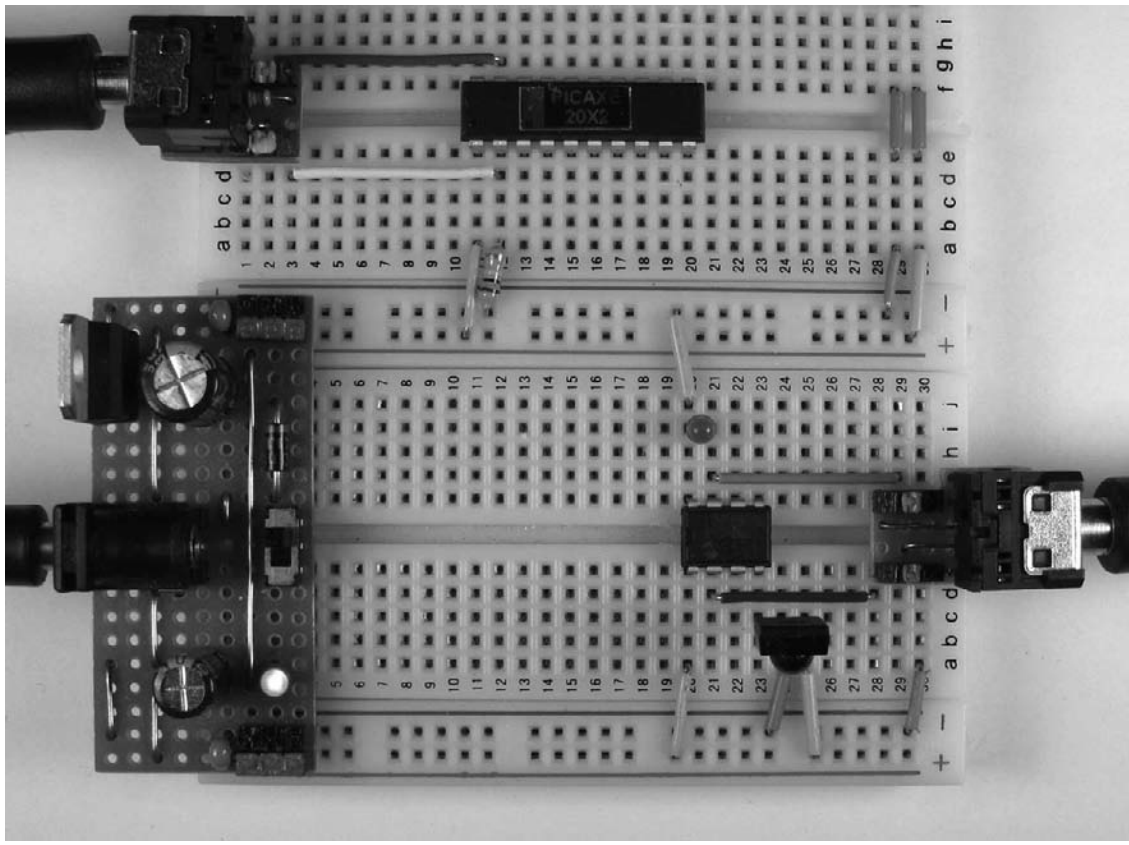


Figure 8-3 Breadboard setup for Experiment 1

schematics in the book, but don't forget that it's actually there!

As you can see in the schematic (Figure 8-2), the output of the PNA4602M (pin 1) is directly connected to input C.3 (pin 4) of the 08M2. The LED that's connected to the serout pin is just in the circuit for debugging purposes. Whenever the 08M2 sends serial data to the terminal window, it does so via the serout pin, so the LED will flicker briefly to let you know that the program is looping correctly.

The syntax for the *irin* command is *irin pin,variable,* where *pin* specifies the IR input pin and *variable* specifies the variable to receive the value associated with a specific keypress on the IR remote. As I have already mentioned, we're using pinC.3 to maintain compatibility with the older 08M processor. The M-class chips also included a built-in variable named *infra*, which was automatically assigned to variable b13. *Infra* also functions correctly with the newer *irin* command, so we will use it to maintain backwards compatibility, but don't forget that you can assign any variable you want to receive the IR data in an *irin* command.

Table 8-1 shows the IR input values for the 16 buttons that are most commonly implemented on inexpensive universal remotes. (The PICAXE Manual includes a more complete listing of the SIRC codes.) In Experiment 1, we're using the raw values that are generated by each button-press. In Experiment 2, we're going to convert them into a more logical order, so you may want to refer back to Table 8-1 when we get to that point.

The software for our first experiment (TV-IRinput.bas) is as simple as the hardware (see Listing 8-1); it requires only a brief explanation. The main loop spends most of its time at the *irin* command waiting for valid data. As soon as it receives a valid data byte, the program transmits it (as individual digits) to the terminal window,

TABLE 8-1 IR Input Values for Experiment 1 and Experiment 2

TV Remote Button	Raw Value (Experiment 1)	Converted Value (Experiment 2)
0	9	0
1	0	1
2	1	2
3	2	3
4	3	4
5	4	5
6	5	6
7	6	7
8	7	8
9	8	9
Channel +	16	10
Channel –	17	11
Volume +	18	12
Volume –	19	13
Mute	20	14
Power	21	15

followed by a *carriage return* and *line feed*. (Every time that happens, the LED will flicker briefly.) The *pause* command simply slows the program down to avoid duplicate data transmissions if the button is held too long.

Download the program to your breadboard setup and test it out. If you don't see the appropriate values appear in the terminal window in response to your button-presses, there are two likely culprits. First, you may need to experiment with a different Sony TV code for your remote—with the dozen or so remotes that I have tried, one of the listed codes has always worked. Second, if the remote you are using is capable of controlling more than one device (e.g., a TV and a DVD), make sure you have pressed the "TV" button. If one of the other devices is currently active, the codes are entirely different and the program will not respond correctly. It's surprisingly easy to inadvertently press one of the other device buttons as you are experimenting. If the program had been working properly and suddenly stops, that's probably what has happened. Simply press the "TV" button again to be sure.

LISTING 8-1

```
' ============================== TV-IRinput.bas ==============================
' This program runs on a PICAXE-08M2 processor at 4 MHz.
' It waits for key-press from a SIRC TV remote control.
' When it's been received, it sends it to the Terminal.
' Note: "infra" is a built-in variable assigned to b13.
' ============================================================================

' === Directives ===
  #com 4                              ' specify com port
  #picaxe 08M2                        ' specify PICAXE processor
  #terminal 4800                      ' specify terminal baud rate

' ========================== Begin Main Program ==========================
do
  irin C.3, infra                     ' wait here for infrared input
  sertxd (#infra,CR,LF)               ' send ASCII digits to Terminal
  pause 500
loop
```

Experiment 2: Interfacing the IR Circuit with the Master Processor

When you have Experiment 1 operating correctly, we're ready to move on to the task of connecting our IR-remote circuit to the master processor. Our breadboard setup for Experiment 2 is shown in Figure 8-4. It's essentially the same as our setup for Experiment 1, with the addition of two connections between the 08M2 and the 20X2 master processor. As you can see in the photo, the 08M2's C.2 pin (which we are going to configure as an output) is connected to the 20X2's C.2 pin (which we will configure as an input). Similarly, the 08M2's C.4 pin (which will be configured as an input) is connected to the 20X2's C.5 pin (configured as an output). Later when we get to our project, you'll see why I chose those particular pins. Also, you may remember from Chapter 5 that the function of the 1k resistor in each connection is to protect the processors from the possibility of both pins in the same connection accidentally being configured as outputs. Finally, you can see that I didn't include debugging LEDs in the circuit. If you have any trouble getting the circuit to function correctly, you may want to add an LED to each of the output pins (C.2 on the 08M2 and C.5 on the 20X2) for debugging the circuit.

Software for the 08M2 IR Peripheral Processor

The major purpose of developing a stand-alone peripheral device for handling IR input is to free up the master processor to go about its business without having to deal with the unpredictable arrival times of the incoming IR data. This task becomes the responsibility of the peripheral processor—since it's the only task it has, there's no problem if it devotes virtually 100 percent of its time to the job. Of course, we also need a flexible method for the 08M2 to transmit the IR data on to the 20X2. There are many ways to accomplish this goal. For example, we could use interrupts to get the job done, or we could turn to the 20X2's powerful *hserin* command to receive the data in

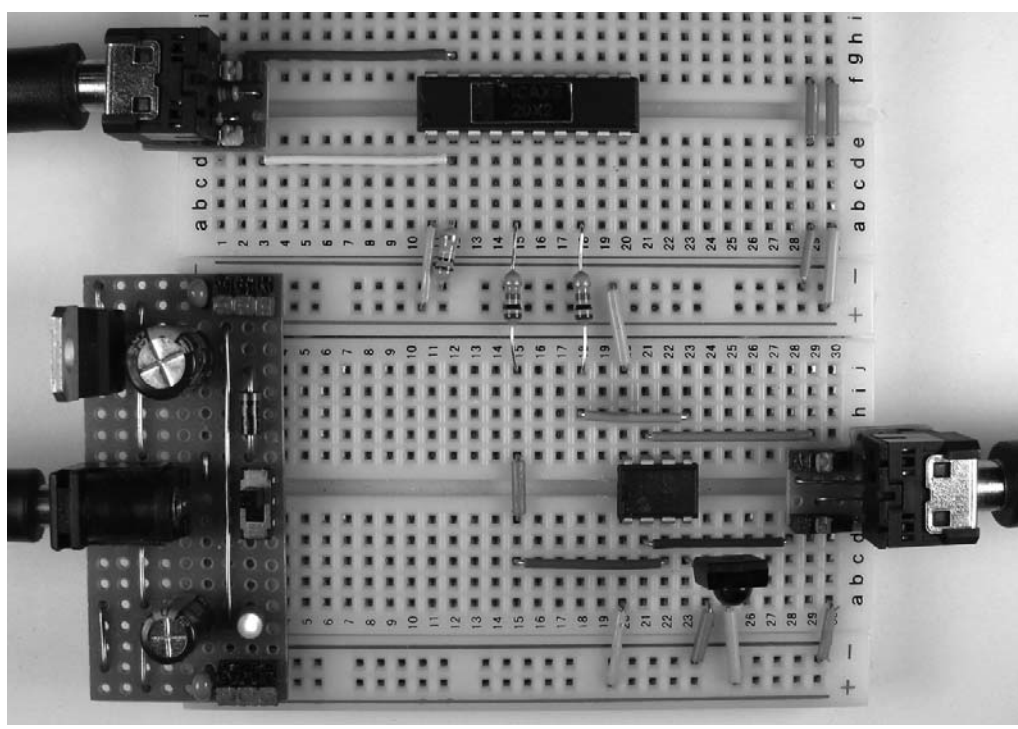


Figure 8-4 Breadboard setup for Experiment 2

the background. However, I opted for a simple "hand-shaking" approach in this project. The 08M2 is going to alert the 20X2 when data is available, and when the 20X2 is ready to receive the data it will tell the 08M2 to send it. The advantage of this approach is that it will also work with any M2 "master" processor.

The necessary software for the 08M2 peripheral (TV-IRtoMP.bas) is presented in Listing 8-2. Whenever the 08M2 receives a valid IR data byte,

LISTING 8-2

```
' ================================ TV-IRtoMP.bas ================================
' This program runs on a PICAXE-08M2 processor at 4 MHz. It waits
' for an IR signal from a SIRC TV remote control. When it's received,
' it encodes & transmits it as single byte to the Master Processor.
' Note: "infra" is a built-in variable assigned to b13.
' ==============================================================================

' === Constants ===
  symbol toMP   = C.2                   ' output line to master processor
  symbol fromMP = C.4                   ' input line from master processor

' === Variables ===
  symbol junk = w0                      ' pulsin requires a word variable

' === Directives ===
  #com 4                                ' specify com port
  #picaxe 08M2                          ' specify processor
  #terminal off                         ' disable the Terminal Window

' ============================ Begin Main Program ============================
do
  irin C.3, infra                       ' wait here for infrared input

  select case infra                     ' implement encoding scheme
    case < 9
      inc infra                         ' make digits 1-9 & data correspond
    case 9
      infra = 0                         ' make 0 = 0
    case < 22
      infra = infra - 6                 ' lower the remaining values
    else
      goto skip                         ' ignore any other keys
  endselect

  high toMP                             ' tell MP that data is available
  pulsin fromMP, 1, junk                ' we're just waiting
  low toMP                              ' prepare to send data
  serout toMP,N2400_4,(infra)           ' send remote key-press to Master
skip:
  pause 500                             ' avoid multiple key-presses
loop
```

it uses the *select case* statement to convert the incoming signal to the value presented earlier in the third column of Table 8-1. I chose those specific values to equate the single-digit buttons with their corresponding codes and to pack the remaining codes sequentially. If you prefer a different encoding scheme, it would be a simple matter to modify the relevant *case* clauses in the *select case* statement.

As soon as the received data byte has been encoded, the 08M2 raises its output line (C.2) to a high level to alert the 20X2 that there's valid data available, and then "listens" on its input line (C.4) for the instruction to send the data. The 20X2 will send a brief "high" pulse on its output pin (C.5) that is connected to the 08M's C.4 input when it's ready to receive the data. (The *pulsin* command that I used to receive the 20X2's "send it" command includes a variable that contains the length of the incoming pulse. I declared that variable as *junk* because we're not interested in the length of the pulse, just the fact that it has arrived.) When the 08M2 receives the "high sign" from the 20X2, it immediately lowers its output pin and sends the data (serially) to the 20X2, which is already waiting to receive it.

Before you actually download the TV-IRtoMP.bas program to the 08M2, let's take a look at its companion program for the 20X2 (SerinFromIR.bas), which is shown in Listing 8-3. The most noteworthy point about the program is that there are two different declarations for the C.2 pin. That's necessary because there are two different ways to refer to an input pin. Most of the time, we simply want to specify which pin a command is to use—for example (in the current program), *serin fromIR*, …. In this situation, C.2 is a constant (i.e., the pin on which we are receiving serial data doesn't change), so the correct declaration is *symbol fromIR = C.2*. However, at other times, we're interested in the real-time value of the input pin; that is, is it currently high or low? In this case, the current state of the input pin is a variable that can change—for example (in the current program), *if IRflag = 1 then*…. As you may remember, the special function variable *pinsC* (for example) is subdivided into eight individual pin (bit) variables named *pinC.7* through *pinC.0* so the correct declaration for the variable is *symbol IRflag = pinC.2*. (Make sure you don't use the same name for both declarations!)

The main program is simple. The initial 100mS pause is just there to simulate a much longer program. The "data available" signal from the IR peripheral will most likely arrive somewhere in the middle of the pause and won't be noticed until the pause has timed out. Fortunately, our communication scheme is specifically set up to deal with such delays. Immediately after the pause, the master processor checks to see if there is any available IR data. If so, the program jumps to the *getData* subroutine, where it signals the IR peripheral to send the data and then receives it and sends it on to the terminal window for display. Once that's done, the main loop simply repeats.

Whenever you are working simultaneously with two programs for two different processors, there are a couple of points to keep in mind that will make things easier. If you are using the terminal window in one of the programs but not in the other (as we are in this experiment), be sure to include the *#terminal off* directive in the program that doesn't need it. This will stop the terminal window from opening when you don't want it to. Also, whenever you need to download both programs, save the one that uses the terminal window until last so that it opens for the correct program. If you need the terminal window and it isn't there, you can open it manually from the PICAXE | Terminal menu option or just press the F8 key.

One last word of caution: There may be times when one of the programs won't download properly. This can happen because the program is busy waiting for something (either an IR command

LISTING 8-3

```
' ============================== SerinFromIR.bas ==============================
' Program runs on a 20X2 & receives data from IR peripheral.
' =============================================================================

' === Constants ===
  symbol abit   = 500                       ' used in pauses
  symbol fromIR = C.2                       ' input pin specification
  symbol toIR   = C.5                       ' output pin specification

' === Variables ===
  symbol IRflag = pinC.2                    ' value of input pin variable
  symbol key = b0                           ' value of key-press

' === Directives ===
  #com 3                                    ' specify com port
  #picaxe 20X2                              ' specify processor
  #no_data                                  ' reduce download time
  #no_table                                 ' reduce download time
  #terminal 9600                            ' open terminal window

' ============================ Begin Main Program ============================
do
  pause 100                                 ' pretend to be busy
  if IRflag = 1 then gosub getData
loop

' ================== End Main Program - Subroutines Follow ==================
getData:
  pulsout toIR,5                            ' 50uS "send it" pulse
  serin fromIR,N2400_8,key                  ' get value of key-press
  sertxd (#key,cr,lf)                       ' send value to Terminal
  return
```

or serial input) and doesn't respond to the download request. Whenever this happens, just turn the power off, restart the download, and turn the power back on—the download should proceed normally. Keeping all that in mind, download TV-IRtoMP.bas to the 08M2 and SerinFromIR.bas to the 20X2. A button-press on the TV remote should produce the corresponding IR code in the terminal window. If it doesn't, you may want to add debugging LEDs to troubleshoot your setup.

Project 8
Constructing the TV-IR Input Module

Now that we have a fully functional design, we're ready to convert our IR input circuit to a self-contained peripheral module. As usual, we'll use a small stripboard circuit; the schematic is presented in Figure 8-5. As you can see, it's similar to the breadboard circuit we just tested, with one important exception—the serin pin is directly tied

to Ground. In the breadboard circuit, the serin pin was tied to Ground by the circuitry in the programming adapter. We won't be programming the 08M2 when it's inserted in the socket because we have already downloaded and tested the program in the breadboard circuit. When the stripboard is completed, we'll simply remove the 08M2 from the breadboard and insert it into the stripboard socket. Therefore, we can tie serin directly to Ground in order to allow the program to run. The only downside is that we won't be able to program the 08M2 when it's in the stripboard circuit, but that doesn't really matter because there's no need to change the program. If you think of a great new feature you would like to add to the program, all you have to do is temporarily move the 08M2 to a breadboard circuit to reprogram it and then back to the stripboard to actually run it.

Constructing the TV-IR Remote Input Module

Constructing the TV-IR input module is straightforward. The parts list is shown in the Parts Bin and the stripboard layout is presented in Figure 8-6.

PARTS BIN

ID	Part
—	Small piece of stripboard, 5 traces with 6 holes each
08M2	DIP socket, 8-pin (machined pins)
08M2	PICAXE-08M2 (or 08M) processor
4602	PNA4602M IR detector
LED	LED, 3 mm, resistorized
H1	Header, female, 3-pin section
H2	Header, male, reverse-mountable, 5-pin section

Three aspects of the layout require clarification. First, the machined-pin socket is required because the jumper from D1 to D5 is on the top of the board so some space is needed underneath the socket. Second, the combination of "X" and a dark center at B5 are intended to indicate that the header pin should be soldered at this point and then snipped off. If you prefer, you could use a four-pin male header instead and leave hole B5 empty. Either way, there should not be a pin at this

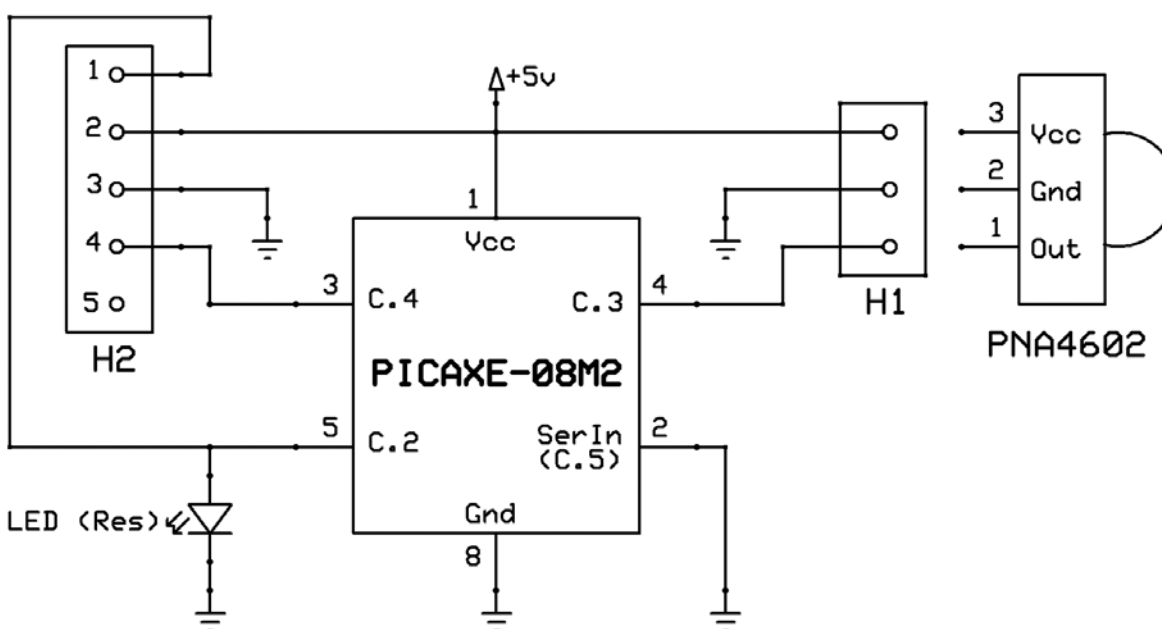


Figure 8-5 Schematic for the TV-IR input module

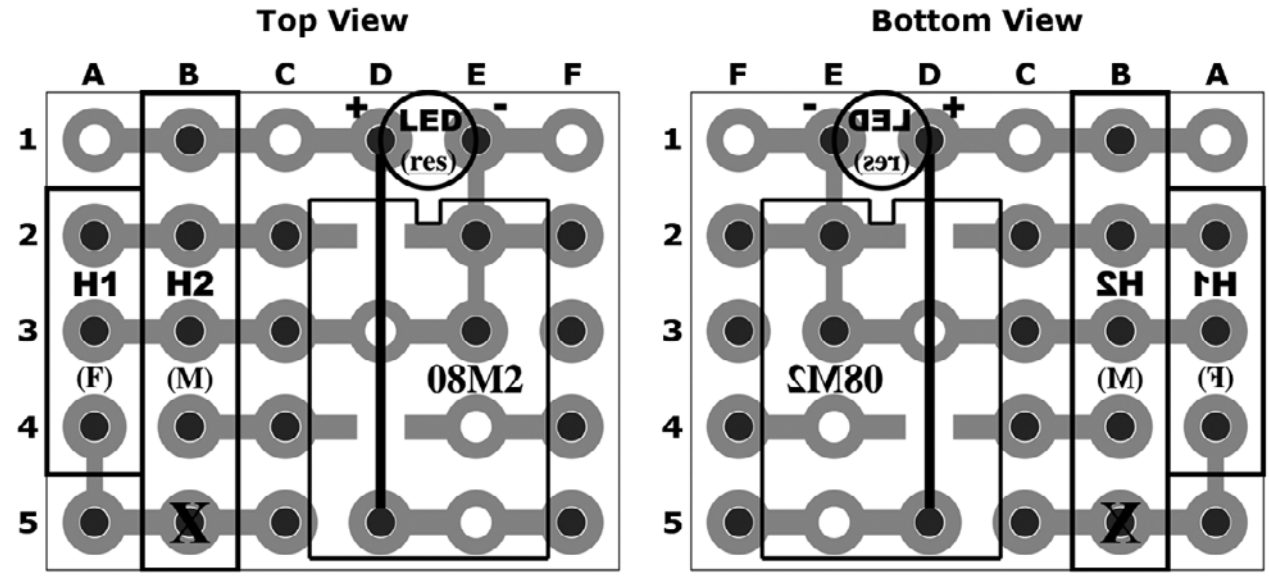


Figure 8-6 Stripboard layout for TV-IR remote input module

location. Finally, the female header that's used for connecting the PNA4602M is included in the circuit because we're going to be using the TV-IR module again when we get to Part Three. At that time, we'll need to bend the 4602's pins at a right angle. I included the female header so that it would be possible to use two different 4602s (one with straight pins and the other with bent pins), rather than rebending the pins every time we needed a different orientation for the 4602—it certainly wouldn't last very long that way.

As usual, read through the complete list of assembly instructions that follows to be sure you understand the entire procedure before assembling the board.

1. Cut and sand a piece of stripboard to the required size (five traces with six holes each).
2. Using a 3/64-inch (1.2-mm) drill bit, enlarge the hole at D1.
3. Sever the traces on the bottom of the board as indicated in Figure 8-7.
4. Clean the bottom of the board with a plastic Scotch-Brite or similar abrasive pad.
5. Using a piece of thin wire (cut from a 1/6-watt resistor or other small part), insert the jumper from D1 to D5 on the top of the board; solder and snip the end at D5 on the bottom of the board. (Do *not* solder the end at D1 yet.)
6. Solder the machined-pin socket in place.
7. Observing the correct polarity, insert the resistorized LED in place; solder and snip the lead at D1 on the bottom of the board. (Do *not* solder the lead at E1 yet.)
8. On the bottom of the board, bend the LED lead from E1 toward E3. Snip the lead so that it just touches the trace at E3.
9. Using a small spring clamp to hold the LED lead tightly against the traces, solder the lead at E1, E2, and E3.
10. File or sand all the cut leads on the bottom of the board.
11. From the top of the board, insert the five-pin reverse-mountable male header through holes B1–B5; invert the board, support the header so that it remains fully inserted, and solder the five pins in place.

12. Snip the header pin on the bottom of the board at B5. Do not snip any of the other four pins! (The short header pins on the top of the board can either be left intact or snipped, whichever you prefer.)
13. From the top of the board, insert the three-pin female header through holes A2–A4; invert the board, support the header so that it remains fully inserted, and solder the pins at A2 and A3. Do *not* solder the pin at A4 yet.
14. On the bottom of the board, bend a piece of *thin* wire jumper around the pin at A4 and snip it so that it just touches the trace at A5; solder the jumper at A4 and A5.
15. File or sand all the cut leads on the bottom of the board.
16. Clean the flux from the bottom of the board and allow it to dry.
17. Inspect the board carefully for accidental solder connections and other problems.

Figure 8-7 is a composite photo of the top and bottom of the completed TV-IR module. In the top view, I have again painted the tops of the male header pins to remind myself of their functions (refer to Figure 8-5 for the header pin-out). If you decide to do that, don't wait until after the female header is soldered in place, as I did—I'm sure it would be a lot easier that way!

Using the TV-IR Remote Input Module

When you have completed the TV-IR module, simply remove the programmed 08M2 from the breadboard and insert it into the socket on the stripboard with pin 1 facing up toward the LED. Also insert the PNA4602M into the three-pin female header so that the rounded portion is facing the 08M2. In the breadboard layout presented in Figure 8-8, you can see that I have removed all the circuitry related to the 08M2 and replaced it with the TV-IR module. If you align pin 1 of the module's male header (at the right edge of the module in the photo) with pin C.2 on the 20X4, pin 4 of the module's header will also be correctly aligned with pin C.5 on the 20X2. The new jumpers in the photo simply connect pin 2 of the header to +5V and pin 3 to Ground.

Both processors have already been programmed with the necessary software, so all that remains is to power up the project and try it out. Since we're not downloading a new program, the terminal window won't open automatically. To make it visible, just press the F8 key or select PICAXE | Terminal. The

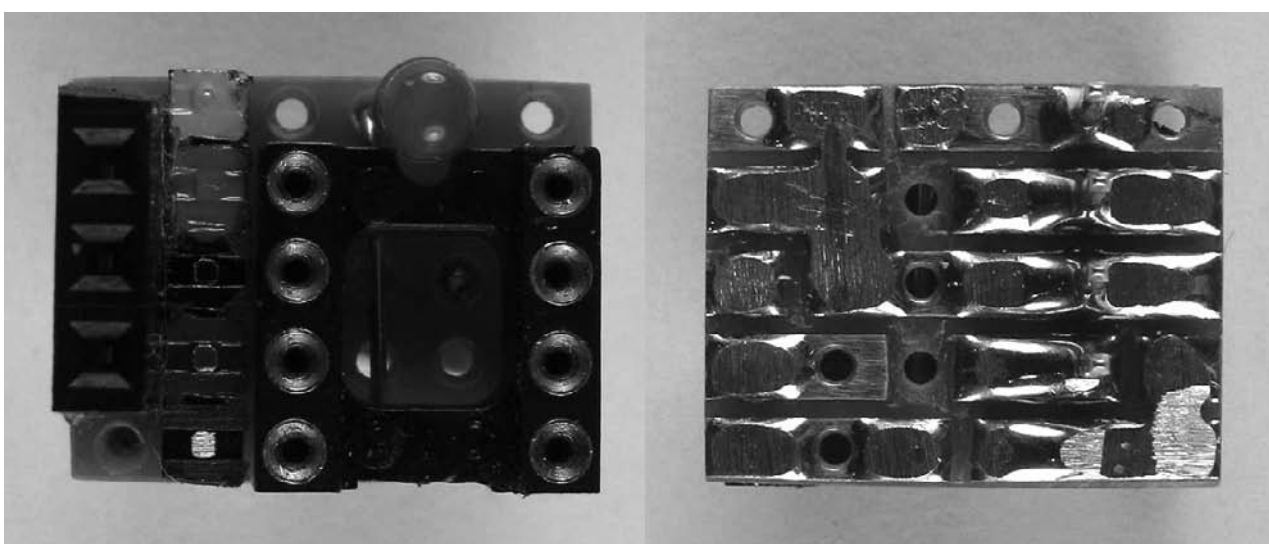


Figure 8-7 Composite photo of completed TV-IR module

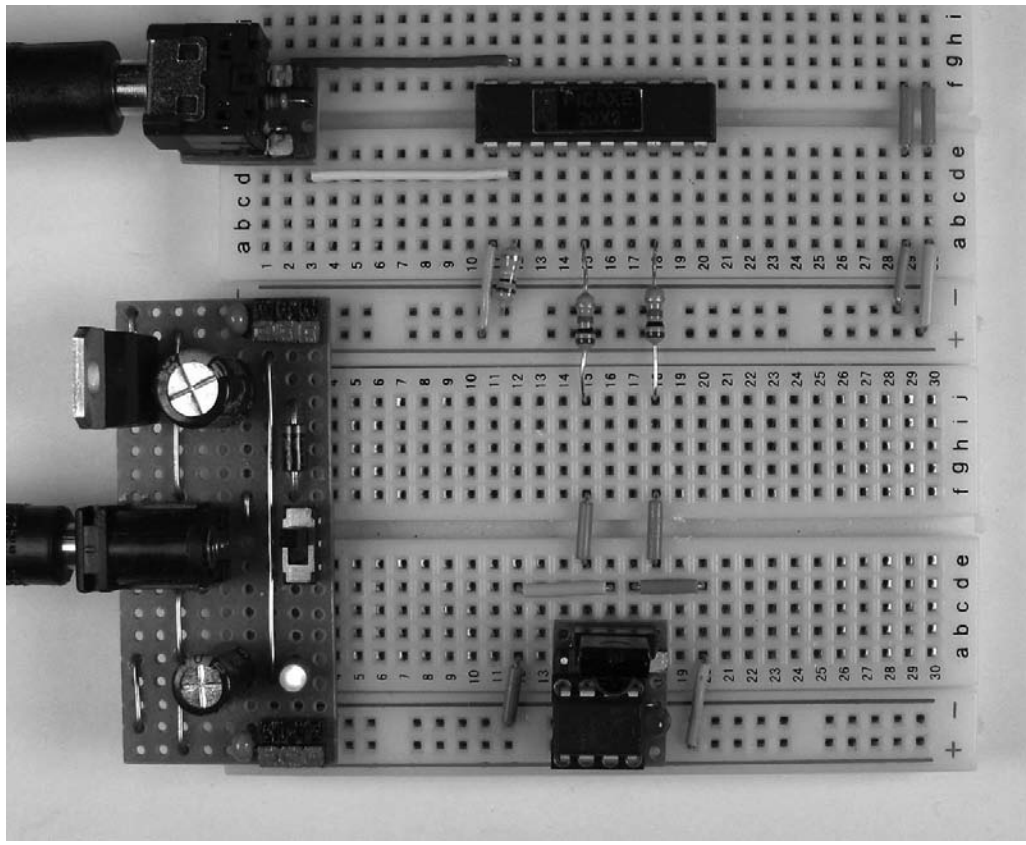


Figure 8-8 Breadboard layout with TV-IR module installed

system should behave the same as it did in Experiment 2. If it doesn't, you will need to recheck the stripboard soldering for possible problems.

The completed TV-IR module can be used as a stand-alone peripheral device in any PICAXE project. All the master processor has to provide is one output pin and one input pin. In return, it will gain the capability of being able to respond to real-time input from the user. As you know, this can be a valuable addition to many projects.

This page intentionally left blank

CHAPTER 9

Interfacing Parallel LCDs

A LIQUID CRYSTAL DISPLAY (LCD) can be a useful addition to many microcontroller projects. In addition to providing detailed feedback to the user, LCDs can be helpful during the debugging phase of software development. We're going to focus on character-based displays rather than graphics displays because they are readily available, relatively inexpensive, and easy to interface. We'll limit our discussion to LCDs based on the Hitachi HD44780 controller chip because they constitute the vast majority of the LCDs that are currently available from surplus and other parts suppliers.

In this chapter, we'll examine the details of interfacing a standard parallel LCD; in the next chapter we'll use what we have learned to develop a stand-alone "serialized" display. Interfacing a microcontroller with a parallel LCD requires a minimum of six output lines and a fair amount of code, so this type of LCD can't be used in any projects that are based on the 08M2 processor, because it doesn't have six output lines. Even with the larger M2 processors, there are times when there aren't sufficient I/O lines to accomplish everything you want to do. Of course, there are many commercially available serial LCDs that only require one I/O line, but they tend to be fairly expensive. By the time you finish both LCD chapters, you'll be able to construct your own serialized LCD for a fraction of the cost of a commercial unit. And the best part is, you'll have a lot of fun doing it!

This isn't my first foray into the world of character LCDs. I wrote a three-part series on the subject in the *Nuts and Volts* "PICAXE Primer" (April, June, and August 2009). At that time, the 20X2 processor hadn't yet arrived, so the projects I developed were based on the 14M. The resulting serialized LCD module works great, but it does require the master processor to slow things down a bit so that the 14M can keep up with the incoming data. Now that the 20X2 is available, it's a whole new ballgame. For one thing, it's capable of operating at 64MHz, which is eight times faster than the 14M's top speed of 8MHz. More importantly, the 20X2 supports the new *hserin* command, which can receive serial data in the background while the processor is attending to other tasks, such as displaying data on an LCD. Therefore, a 20X2-based serial LCD will greatly simplify the timing demands on the master processor that is sending the data to the LCD.

Before we can begin developing our new "super-serial" LCD, we need to become familiar with the basics of interfacing with HD44780-based LCDs. In the following discussion, I'm not going to repeat all the details originally presented in the "PICAXE Primer" series of articles, so you may want to refer to them for more information. In addition, in Part III of the PICAXE Manual ("Microcontroller Interfacing Circuits"), pages 30–41 are devoted to interfacing LCDs. Naturally, there's also considerable information available on

the Web; a quick search for "HD44780" will provide an overwhelming amount of it. An excellent two-part tutorial can be found at www.epemag.com/lcd1.pdf and www.epemag.com/lcd2.pdf, and a valuable PICAXE-specific LCD resource is available at www.hippy.freeserve.co.uk/picaxelc.htm.

Understanding the Basics of HD44780-based LCDs

Character-based LCDs are commonly available in sizes of 8, 12, 16, 20, 24, 32, or 40 characters by 1, 2, or 4 lines. We're going to be using a 16X2 display because that's by far the most readily available and inexpensive variety. New displays are available for about $10 (see www.mouser.com) and for considerably less on the surplus market (e.g., www.allelectronics.com or www.goldmine-elec.com) and eBay. Any 16-character by 2-line HD44780-compatible LCD will work for the experiments and project in this chapter, as long as it meets the requirements we are about to discuss.

All HD44780-based LCDs share a standard pin-out, which is presented in Table 9-1. Pins 15 and 16 are optional; their function is to power the display's backlight, if it has one. (We'll discuss each of the pins in more detail later.) For our purposes in this chapter, the most important consideration is the position of the I/O connector on the LCD board. If you jump ahead for a moment and take a look at Figure 9-3, you'll see the solder connections for the LCD's 16-pin connector in a single row near the left end of the top edge of the display. This is the configuration that we will need when we construct our projects in this chapter and the next. The majority of HD44780-compatible LCDs have their connector in this position, so you shouldn't have any difficulty finding one to use in our experiments.

If you decide to use a backlit display, and especially if you intend to experiment with more than one display, there's an important issue that you need to keep in mind. The LCD backlighting is LED-based, so a current-limiting resistor is usually required. However, the correct value is rarely stated in the display's datasheet, so you may need to do a simple calculation to determine the correct size for the current-limiting resistor. You need to know the typical forward voltage (V_f) across the backlight and the maximum forward current (I_f) through the backlight. For example, the datasheet for one of my LCDs specifies a V_f of 4.0V and an I_f of 120mA. Since we're using a 5V supply, a V_f of 4V leaves 1V to be dropped across the resistor, and 1V / 120mA = 8.3Ω, so that's the minimum resistor size we need. (Actually, I would probably double that to be safe.)

TABLE 9-1 HD44780 LCD Pin-out

Pin	Name	Function
1	Vss	Ground
2	Vcc	+V (+3 to +5 volts)
3	Vee	Contrast control
4	RS	Register select (H = data / L = cmd)
5	R/W	Read/write (H = read / L = write)
6	En	Enable
7	DB0	Data bit 0
8	DB1	Data bit 1
9	DB2	Data bit 2
10	DB3	Data bit 3
11	DB4	Data bit 4
12	DB5	Data bit 5
13	DB6	Data bit 6
14	DB7	Data bit 7
15	A	Backlight anode (optional)
16	K	Backlight cathode (optional)

Unfortunately, you can't always find the datasheet you need, and even if you do, some of them don't include the necessary data (go figure!), so you're left on our own to determine the correct size for the current-limiting resistor. The easiest way to do that is to start with a safe value (e.g., 330Ω) and see if the display seems bright enough. If not, lower the value a bit and try it again until the display is reasonably bright. Once I have found a value that I like, I write it on the back of the display with a Magic Marker. That way I don't have to repeat the process every time I experiment with a different backlit display.

The HD44780 Instruction Set

All HD44780-compatible LCDs also share the same set of instructions or commands that accomplish a variety of tasks. For example, you can send a command to the LCD to show the cursor or hide it, or make it blink. It's even possible to scroll the contents of the display window to create a "moving message" effect. However, we have more than enough information to deal with in this chapter, so we're going to limit ourselves to the basic commands presented in Table 9-2. If you're interested in some of the more advanced commands, the datasheet for your LCD should include a complete reference.

TABLE 9-2 Selected HD44780 Commands

Command	Function
56	Initialize display as follows: 8 data bits, 2-line LCD, 5 by 7 dots
1	Clear display and cursor "home"
12	Hide cursor
14	Show cursor
128	Move cursor to start of line 1
192	Move cursor to start of line 2

The first command in Table 9-2 requires a brief explanation. First, the manner in which the LCD is initialized is determined by a command called "Function Set." Rather than explaining all the options of the *Function Set* command, I have simply included the correct value (56) for the way we are going to set up the interface: a two-line LCD with 5 × 7 dot characters and an eight-bit data interface. If your display requires something different (e.g., if it's a one-line display), see the documentation for the *Function Set* command in the display's datasheet.

In addition to being able to move the cursor to the beginning of either line (the last two commands in Table 9-2), you can place it at any specific character location by simply sending the appropriate command. The display locations are numbered sequentially from the beginning of each line, so you just need to know the value that's associated with the position at which you want to print the character. For example, if you wanted to print a character at the fourth position of line 2, you would simply send a command of 195 (i.e., 192 +3). Also, when a character is printed to the display, there's no need to specify the location for the next character—it's automatically incremented by one unless you specify otherwise. As we'll see in our first experiment, this makes it a simple matter to send a string like "Hello World!" to the LCD.

LCD Interface Requirements

HD44780-based LCDs can be configured to operate in either an eight-bit or four-bit data transfer mode. Since the four-bit mode requires two data transfers for each byte that is sent to the LCD, it takes approximately twice as long to display the same data as it would in the eight-bit mode. When I developed the PICAXE Primer 14M-based LCD projects, I used the four-bit mode out of necessity—the 14M doesn't have sufficient output pins to implement the eight-bit mode. (In case you were wondering, neither does the older 20M.)

Fortunately, we can now take advantage of the numerous output pins available on the 20X2 and configure our interface for eight bits, which turns out to be much simpler than the four-bit mode. Of course, it's also twice as fast.

At this point, we're ready to take a more detailed look at the requirements for interfacing the 20X2 with the HD44780's 16-pin connector (refer back to Table 9-1):

- **LCD power supply (pins 1 and 2):** Nothing special here—pin 1 is connected to Ground and pin 2 is connected to +5V.
- **Contrast adjustment (pin 3):** This pin simply requires a 10k potentiometer with one end connected to +5V, the other end grounded, and the wiper connected to the pin 3 input.
- **Register select (pin 4):** Prior to sending a data byte to the LCD, the *Register Select* pin must be set high; prior to sending a command byte, it must be in a low state.
- **Read/write input (pin 5):** This input needs to be at a low level whenever we want to write to the LCD, and at a high level when we want to read data back from it. We won't be doing any data reading, so we'll just tie pin 5 to Ground. Because of this, our data interface is unidirectional. That is, all the PICAXE connections are outputs and all the LCD connections are inputs. Therefore, we don't have to bother with any current-limiting resistors on the interface lines (LCD pins 4–14).
- **Enable (pin 6):** In order to actually send a data byte to the LCD, a 10µS "high" pulse must be sent on the *Enable* line.
- **Data (pins 7–14):** These are all simply direct connections.
- **Backlight power supply (pins 15 and 16):** We've already discussed these connections, but a little emphasis never hurts: *Be extremely cautious when choosing the current-limiting resistor for a backlit display.* If the display isn't backlit, pins 15 and 16 probably won't even be there; even if they are, they won't be connected to anything.

Experiment 1: Interfacing an HD44780-based Parallel LCD

Figure 9-1 presents the schematic for our first experiment. (As usual, I haven't included the programming adapter circuitry.) You're probably wondering why I chose the specific 20X2 I/O pins that I did, so I'll briefly explain. Since we need eight data lines, portB is the simplest choice because pin C.6 is fixed as an input-only line. I could have chosen any of the other portC lines for register select and input (except for C.6, which is fixed as an input), but I wanted to avoid C.5 and C.2 in case you want to experiment with sending data from the TV-IR module to the LCD. If your TV-IR circuitry is still on your breadboard, you won't need to change any of the connections that we set up in the previous chapter. Aside from the LCD itself (which is *not* available on my website), the only other required parts are the 10k potentiometer and the 16-pin headers (which *are* available on my website). As I mentioned earlier, the current-limiting resistor is only needed if you are using a backlit display, and you may need to determine its value experimentally.

Figure 9-2 is a close-up photo of my breadboard setup before installing the LCD. The two short jumpers indicated by the arrows have no electrical function. I added them after I accidentally inserted the LCD's connector into the breadboard incorrectly. Fortunately, I didn't blow anything up, but the two short jumpers, which have exactly 16 breadboard rows between them, make it impossible for me to make the same mistake again (I hope).

Figure 9-3 is the same setup with the LCD inserted between the short jumpers and running the software we're about to discuss. In the photo, you

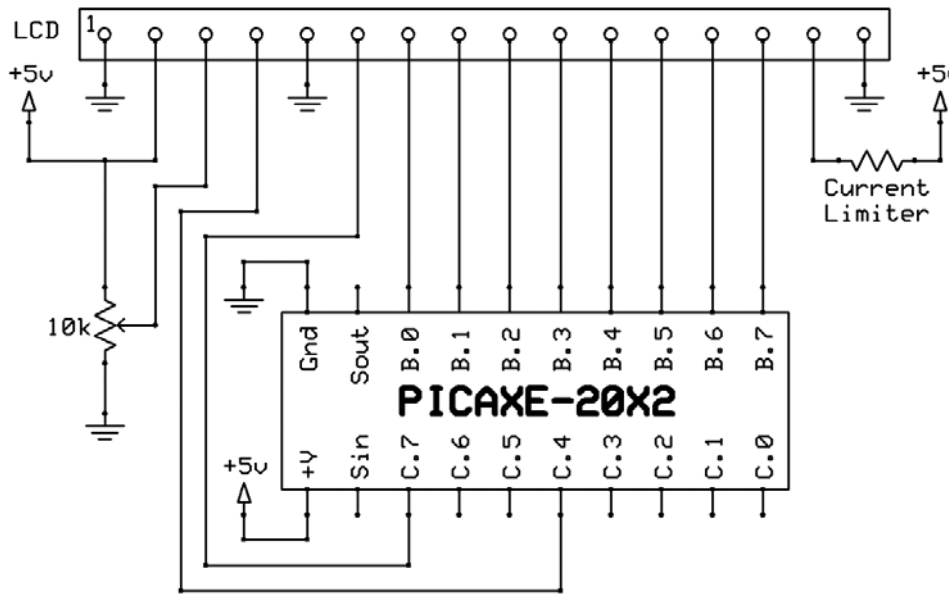


Figure 9-1 Schematic for Experiment 1

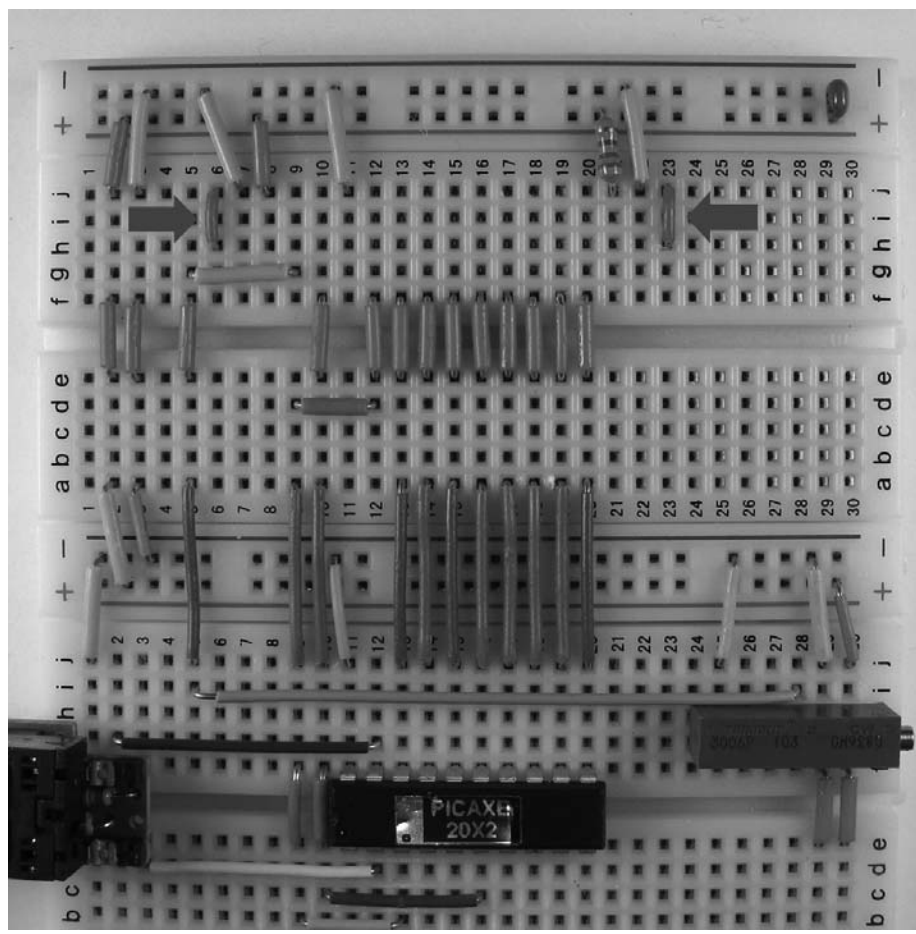


Figure 9-2 Breadboard setup (without the LCD) for Experiment 1

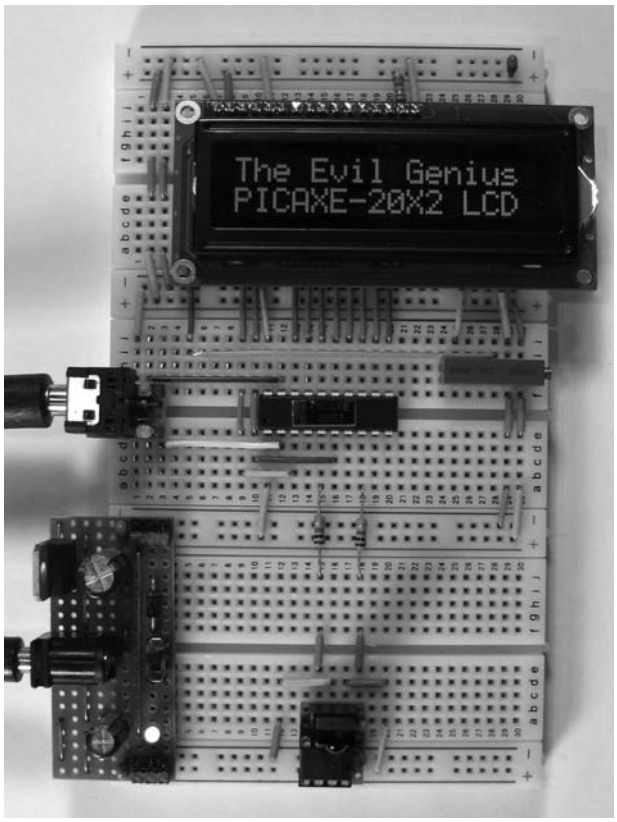


Figure 9-3 Same setup, with the LCD installed

can see that I have left my TV-IR circuitry intact—it won't interfere with the operation of the LCD.

The software for Experiment 1 (LCDparallel.bas) is presented in Listing 9-1. It's more involved than that of our previous projects, so I commented it as thoroughly as I could.

Before you download the software to your 20X2, let's briefly run through the salient details. The *table* command that stores the data to be displayed on the LCD is a convenient alternative to the older *data* command. The significant difference is that *data* storage is in Electrically Erasable Programmable Read-Only Memory (EEPROM), while *table* storage is in flash memory, which has a much faster access time. Otherwise, the two commands are very similar.

At the beginning of the main program, we specify the directionality of all the 20X2 I/O pins (*dirsB = %11111111* and *dirsC = %10111011*). As you can see, I configured C.6 and C.2 as inputs. However, C.6 is fixed as an input anyway and

LISTING 9-1

```
' ============================== LCDparallel.bas ================================
' Program runs on a PICAXE-20X2; sends serial data to an HD44780 16x2 LCD


' === Constants ===
  symbol enable = C.7                   ' LCD enable pin connected to C.7
  symbol RegSel = C.4                   ' LCD RegSel pin connected to C.4


' === Variables ===
  symbol  char = b0                     ' character to be sent to LCD
  symbol index = b1                     ' used as counter in For-Next loops


' === Directives ===
  #com 3                                ' specify download port
  #picaxe 20X2                          ' specify processor
  #no_data                              ' save time downloading
  #terminal off                         ' disable terminal


' === Table ======================
  Table  0, ("The Evil Genius ")        ' data to be displayed on line 1
  Table 16, ("PICAXE-20X2 LCD ")        ' data to be displayed on line 2
```

LISTING 9-1 (continued)

```
' ============================== Begin Main Program ==============================

  dirsB = %11111111                     ' set all portB pins as outputs
  dirsC = %10111011                     ' set C.6 and C.2 as inputs
  pullup  %00000010                     ' enable C.6 internal pullup resistor


' === Initialize the LDC ===
  pause 200                             ' pause 200 mS for LCD initialization
  char = 56                             ' setup for 8-bits, 2 lines, 5X7 dots
  gosub OutCmd                          ' send instruction to LCD
  char = 12                             ' display on, cursor off
  gosub OutCmd                          ' send instruction to LCD


' === Main Program Loop - Send data to the LCD ===
do
  char = 1                              ' clear display & go home
  gosub OutCmd                          ' send instruction to LCD
  wait 1

  for index = 0 to 15
    readtable index, char               ' send line 1 to LCD
    gosub OutTxt
  next index
  wait 1

  char = 192                            ' move cursor to start of line 2
  gosub OutCmd                          ' send instruction to LCD
  for index = 16 to 31                  ' send line 2 to LCD
    readtable index, char
    gosub OutTxt
  next index
  wait 4
loop


' =================== End Main Program - Subroutines Follow ===================

OutCmd:
  low RegSel                            ' set up for command byte
  goto Doit                             ' do it
OutTxt:
  high RegSel                           ' set up for text byte
Doit:
  outpinsB = char                       ' load byte onto outpinsB
  pulsout enable,1                      ' send data
  return
```

we're not using C.2, but I made it an input to be consistent with the connections to the IR-TV module in case you still have it connected. The next statement (*pullup %00000010*) enables the C.6 internal pullup resistor (see the *pullup* documentation in the manual for details). This convenient feature allows us to omit an external resistor on the unused C.6 line.

In the initialization portion of the main program, we need to pause briefly as the LCD begins its internal initialization, and then send two commands to the LCD: "56" to accomplish the initialization mode that we discusser earlier, and "12" to hide the cursor. (You could certainly show it, if you prefer.) After that, the main *do…loop* simply reads data from the table and sends each character to the LCD.

The *OutCmd* and *OutTxt* subroutines are really the heart of the program. They are unusual in that they are actually the same subroutine with two different entry points. This approach simply ensures that *RegSel* will be correctly configured for the byte that is being sent to the LCD (*RegSel* is set *low* for commands and *high* for text). In the main portion of the subroutine (at address *Doit*), the outgoing character is simply placed on the output pins (*outpinsB = char*), and then the *Enable* line is pulsed to actually send the character to the LCD.

That's all there is to it; download LCDparallel.bas to your breadboard setup and run it. At first, you probably won't see anything at all on the LCD because the contrast will most likely need adjustment. While the program is running, adjust the potentiometer throughout its entire range until the text is properly displayed. If nothing appears, you will need to check your breadboard setup for possible wiring problems.

Project 9
Constructing an Eight-bit Parallel 16 x 2 LCD Board

Now that we have a fully functional design, we're ready to convert our breadboard circuit into a self-contained peripheral module. As usual, we'll use a stripboard circuit for that purpose; the parts list is presented below (all the parts are available on my website). We'll skip the schematic because it's essentially the same as the one we just used for our breadboard circuit (Figure 9-1), with the addition of the necessary headers to connect to the LCD and breadboard.

PARTS BIN

ID	Part
—	Stripboard, 25 traces (or 32—see text) with 14 holes each
—	Capacitor, .01µF
—	Potentiometer, 10k
H1	Female header, straight, 16 pins
H2	Female header, right-angle, 4 pins
H3	Female header, right-angle, 4 pins
H4	Female header, right-angle, 8 pins (or two 4-pin headers)
H5	Female header, straight, 4 pins

The layout for our parallel LCD module is presented in Figure 9-4. The first thing we need to discuss is the size of the stripboard. As you can see in the layout, it contains 25 traces. However, a typical 16 × 2 LCD is about 3.2 inches wide, which is equivalent to 32 traces. Therefore, if you want the stripboard footprint to match that of the LCD, you will need to make it 32 traces wide. I didn't want to include the extra seven traces in the layout because I was concerned that it would be too small to see clearly. If you want to use a "full-

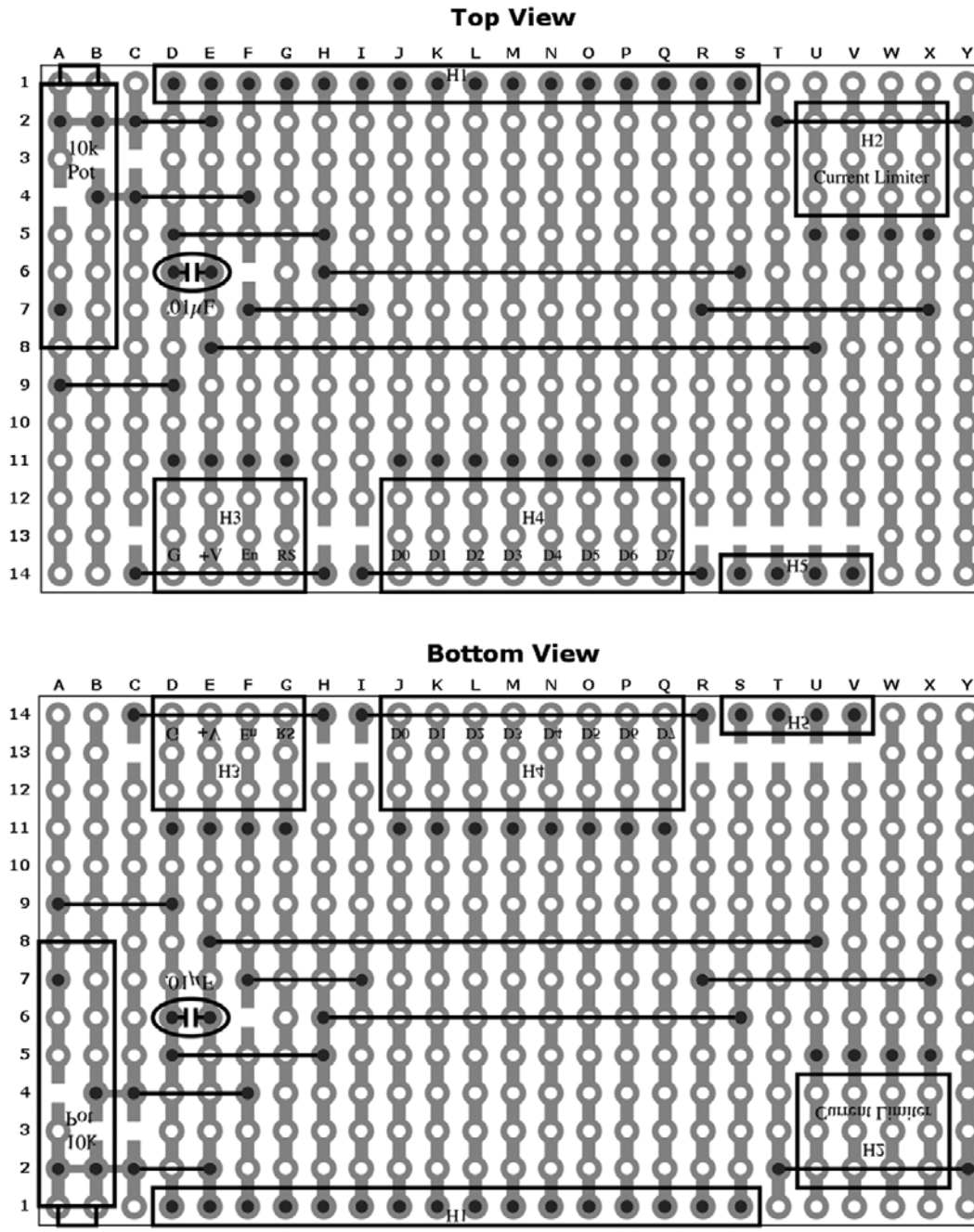


Figure 9-4 Stripboard layout for parallel LCD module

size" stripboard, just include the extra traces—that section of the stripboard will be empty anyway.

Okay, let's discuss the headers. H1 is where the LCD connector gets inserted. If your LCD didn't come with a 16-pin male header already installed, you will need to add one so that the pins protrude from the *bottom* of the LCD. The purpose of H2 is to make it easy to swap out the current-limiting resistor whenever it's necessary—just insert the resistor leads in the first and last sockets of the female header, which is recessed by one row so that the resistor won't protrude above the top of the LCD. H3 and H4 are intended to accept double-ended male headers (available on my

website) that are long enough to also insert into a breadboard (and to bend if you want to adjust the angle at which the LCD sits). H5 is a straight female header that has no electrical function. It's included so that you can insert a four-pin section of a male header and snip off the short ends of the pins. That combination will be the same height as the 16-pin header with the LCD inserted, so H5 will support the LCD so that it sits parallel to the stripboard. (This is a parallel LCD, right?) Finally, the jumpers that run across the tops of each of the right-angle female headers also have no electrical function; they simply "clamp" the headers tightly to the board so that they don't bend and weaken from repeated insertions into and extractions from the breadboard.

As usual, read through the complete list of assembly instructions that follows to be sure that you understand the entire procedure before assembling the board.

1. Cut and sand a piece of stripboard to size (25 or 32 traces with 14 holes each).
2. Sever the traces on the bottom of the board as indicated in Figure 9-5.
3. Clean the bottom of the board with a plastic Scotch-Brite or similar abrasive pad.
4. Insert the jumper from C2 to E2. Bend the lead flat against the board from C2 to A2 on the bottom of the board and snip it so that it just reaches A2. Solder the jumper at E2, but *not yet* at C2, B2, or A2.
5. Insert the jumper from C4 to F4. Bend the lead flat against the board from C4 to B4 on the bottom of the board and snip it so that it just reaches B4. Solder the jumper at F4, but *not yet* at C4 or B4.
6. Insert all the remaining jumpers *except* the three that go over headers H2, H3, and H4; solder and snip the leads.
7. Insert all the right-angle female headers. Flip the board over with the headers in place, set it on a flat surface, and solder all the pins.
8. For each of the right-angle female headers, bend and insert a jumper so that it spans the top of the header as shown in Figure 9-5. Using a small clamp to hold the header and the jumper tightly against the top of the board, solder and snip the jumper leads on the bottom of the board.
9. Insert the capacitor; solder and snip its leads.
10. Insert the 10k potentiometer. Flip the board over and support it on a flat surface.
11. Make sure the unsoldered jumper leads are making contact with the appropriate potentiometer pins; solder one lead at A2, B2, and C2 and the other lead at B4 and C4.
12. Solder the remaining potentiometer pin at A7.
13. Insert the 16-pin and 4-pin straight female headers. Flip the board over and support the headers on a flat surface; solder the headers in place.
14. File or sand all the cut leads on the bottom of the board.
15. Clean the flux from the bottom of the board and allow it to dry.
16. Inspect the board carefully for accidental solder connections or other problems.

Figure 9-5 is a photo of the completed parallel LCD module. As you can see, I chose to make the stripboard the same size as the LCD. (I'll bet you're not surprised.) In addition, you can see the current-limiting resistor that I have installed in the H2 header. What you can't see is that I have filed the cut ends of the four-pin male header that I inserted into H5 and painted them to be sure they don't make any electrical connection with the bottom of the LCD board. Also, I have bent and inserted double-ended male headers in preparation for mounting the module on my breadboard.

In Figure 9-6, I have removed most of the jumpers from Experiment 1 and added the necessary jumpers for the power and Ground connections, which you can't see behind the LCD.

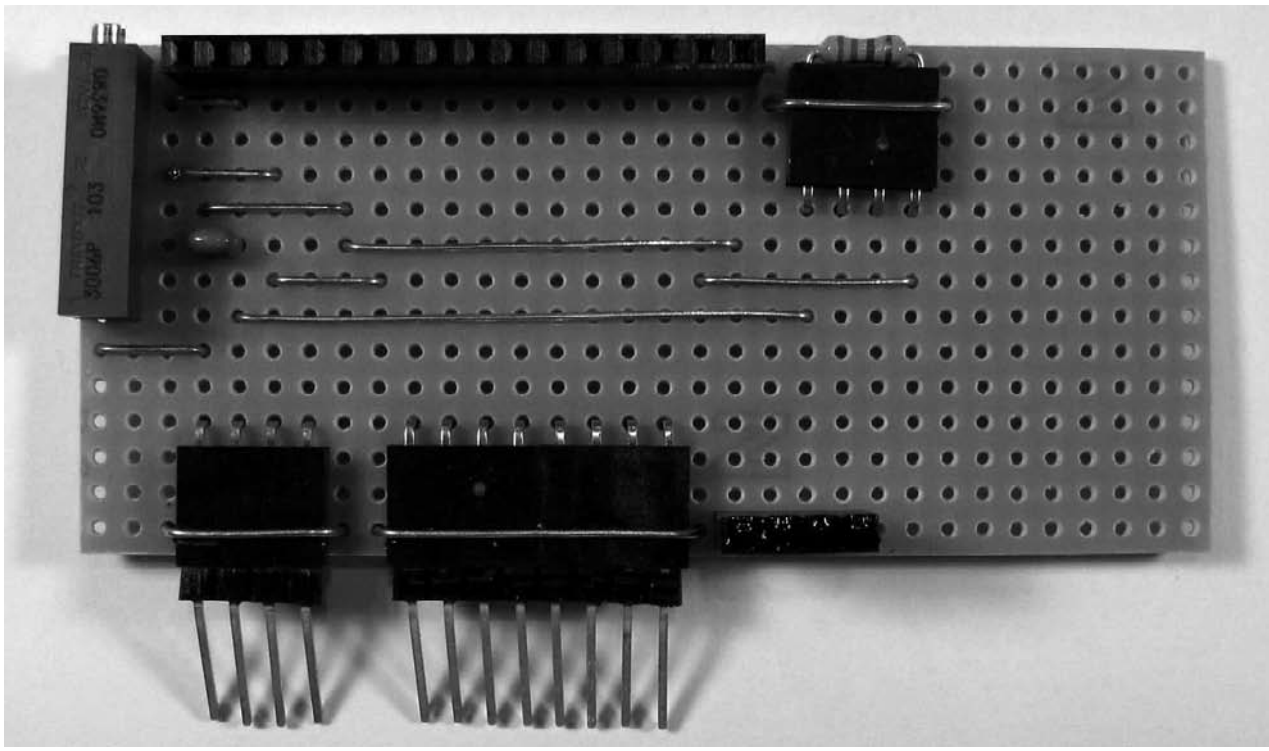


Figure 9-5 Completed parallel LCD module

Figure 9-6 Parallel LCD module in operation

(The jumpers to connect the LCD's *Register Select* and *Enable* lines to the 20X2 were already in place.) As you can see, all the connections are the same as the ones we used in Experiment 1, so you can test the module without needing to change the software in the 20X2. (Obviously, I changed my message for a little variety!) Don't forget that you will probably need to adjust the contrast again, unless you used the same potentiometer from Experiment 1.

Programming Challenge

When you are sure the LCD module is functioning correctly, you're ready to take the LCD programming challenge. I'm assuming you have left your TV-IR module connected as it was in the previous chapter. (If not, just reinstall it with the same connections to the 20X2 pins C.2 and C.5.) The challenge is to create a 20X2 program that receives a keypress from the TV remote and displays its value on the LCD. Have fun!

The parallel LCD module is a great addition to any breadboard circuit that uses the 20X2 processor, or any X1 or X2 chip for that matter. All X1 and X2 processors have more than enough memory and I/O pins to handle the parallel LCD interface and still have considerable resources available for other project tasks. The 20M2 would also be suitable because its pin-out is essentially the same as that of the 20X2. However, as I mentioned earlier, the 08M2 definitely wouldn't work, and the 14M2 might not be quite adequate for a project that includes the LCD and two or three additional I/O devices. The solution, of course, is to use a 20X2 to "serialize" our LCD module so that it can be used by any PICAXE processor, including the little 08M2, which is exactly what we're going to do in the next chapter.

CHAPTER 10

Serializing a Parallel LCD

In this chapter, as you already know, we're going to serialize a parallel LCD. The obvious advantage of doing so is that a serialized display will be usable in any project, regardless of which processor is used, because a serial LCD only requires one output line for interfacing. In addition, the 20X2's capability of receiving serial data in the background will greatly simplify the software requirements on the part of the master processor. If we were to use the older *serin* command (as I did in the earlier "PICAXE Primer" articles that I mentioned in the previous chapter), we would be restricted to sending serial strings of one specified length because the *serin* command either "hangs" if fewer than the expected number of bytes are transmitted or "loses" the extra bytes if too many are sent. Receiving the serial data in the background completely eliminates these issues and allows for a simple and flexible programming interface.

Receiving Serial Data in the Background

In order to implement serial data reception in the background, the 20X2's *hsersetup* command must be properly configured. The complete syntax for the command is *hsersetup baud_setup, mode,* where *baud_setup* specifies the baud rate and *mode* configures the available special functions.

Since we want the LCD to respond as quickly as possible to serial input, we're going to run the 20X2 at its fastest clock speed (64MHz) and receive the data at 4800 baud, which can be easily implemented on all PICAXE processors. Therefore, we'll set *baud_setup* to *B4800_64*. (See the manual for details.) The *mode* parameter is a three-bit variable or constant whose bits specify the following special functions:

- **bit2:** Invert the serial input data (1 = "N"; 0 = "T"). This option only applies to X2-class processors. We'll set bit2 = 1 so that we can have the option of using a master processor's *sertxd* command (which is always an "N" transmission) in conjunction with the *serout* pin on the M2-class processors.
- **bit1:** Invert the serial output data (1 = "N"; 0 = "T"). For the sake of consistency, we'll also set bit1 = 1, but it really doesn't matter because we won't be transmitting any serial data from our LCD. However, it's worth noting that whenever *hsersetup* is configured, both the *hserin* pin (B.6 on the 20X2) and the *hserout* pin (C.0 on the 20X2) are affected. In other words, since we're going to be using B.6 to receive the serial data, C.0 can't be used as an I/O pin in our project—it's automatically configured as the *hserout* pin even though we don't need that function in our project.

- **bit0:** Receive serial data to the scratchpad in the background (1 = automatic serial reception in the background/*hserin* command not used; 0 = reception via *hserin* command *only*/no automatic serial reception).

Taking all this information into consideration, the *hsersetup* statement that we need is *hsersetup, B4800_64, %111.*

In order to understand how the background reception of serial data actually occurs, we need to discuss the scratchpad memory area of the 20X2 and the built-in variables associated with its use. (The scratchpad area is in addition to the variable storage memory area that we have already discussed. All current PICAXE processors have a variable storage area, but the scratchpad only exists on X1 and X2 processors.) On the 20X2, the scratchpad consists of 128 bytes of memory with addresses from 0 to 127. The contents of the scratchpad can be accessed directly by using the *get* and *put* commands (see the manual), and indirectly by using special-function variables that provide the power of indirect addressing that we discussed earlier in reference to the variable storage area. The X1 and X2 processors support the following built-in pointer variables and a system flag that can be used to access the data stored in the scratchpad:

- **hserptr:** This is the "data write pointer." It contains the address of the scratchpad location into which the next incoming serial data byte will be written. As soon as *hsersetup* has been properly configured, *hserptr* is automatically set to 0 so that the first incoming data byte will be stored in scratchpad location 0; next, *hserptr* is automatically incremented so that the next incoming byte is stored in location 1, and so on.
- **ptr:** This is the "data read pointer." It contains the address of the scratchpad location from which the next stored data byte will be read. It functions similarly to the *bptr* variable that is used to indirectly access data in the variable storage area.
- **hserinflag:** This is one of the flags contained in the special-function variable *flags.* It's automatically initialized to 0 when the *hsersetup* statement is executed, and automatically set to 1 as soon as a serial data byte has been received in the background. When our program has finished processing the serial data (i.e., sending it to the LCD), it must reset *hserinflag* to 0 in preparation for the next incoming string of data.

Perhaps the best way to clarify these concepts is with a concrete example. Let's assume that our program has already executed the *hsersetup, B4800_64, %111* statement that we discussed earlier and, subsequently, the master processor has sent the serial string "Hello" to the serialized LCD. Therefore, the following has automatically occurred in the background: "H" has been stored in scratchpad location 0, "e" has been stored in location 1, etc., and *hserptr* now equals 5 (data has been automatically stored in scratchpad locations 0 through 4, so the next incoming byte will be stored at location 5). Also, *hserinflag* now equals 1 because a background serial-receive has occurred. At this point, the following code snippet could be used to access this data and send it to the LCD:

```
if hserinflag = 1 then
  LstChr = hserptr - 1    ' get location
                            of last
                            received
                            byte
  for ptr = 0 to LstChr   ' for each
                            received
                            byte
    char = @ptr           ' fetch the
                            character
                            (see below)
    gosub OutByte         ' and send it
                            to the LCD
  next ptr
  hserinflag = 0          ' reset the
                            new data
                            flag
```

```
   hserptr = 0              ' reset the
                              data write
                              pointer
endif
```

As I have already mentioned, indirect addressing may seem to be an unnecessarily complicated way of doing things, but it's much faster than the simpler direct addressing approach (i.e., *get ptr, char*). Since we want our LCD to be as fast as possible, we'll use pointers and indirect addressing.

At this point, I'm sure you've had your fill of theory, so let's take a break from it and construct our serialized LCD board. Once we have a fully functional board, we'll be able to implement our indirect addressing approach.

Project 10
Constructing a Serialized 16 x 2 LCD

The parts list for our serialized LCD project is shown here; as usual, all the parts are available on my website. The schematic is presented in Figure 10-1; most of it will be familiar by now. However, there is one important difference that we need to discuss. Since we need pin B.6 to function as the *hserin* pin, it's not available as a data output pin to the LCD. In the schematic, you can see that I have substituted pin C.1 in its place. The software adjustment required to handle this substitution is surprisingly simple, as we'll see when we get to that point in the project.

Header H2 is another aspect of Figure 10-1 that requires clarification. Its purpose is to enable us to insert our USBS-PA3 programming adapter so that the 20X2 can be programmed directly on the stripboard. I'm sure you will want to modify and improve the program to add features along the way; in-circuit programming is certainly the most convenient way to accomplish that goal. (When we discuss the software for the project, I'll describe

PARTS BIN

ID	Part
—	Stripboard, 32 traces with 14 holes each
—	Capacitor, .01μF
—	Resistor, 4.7k, 1/4 or 1/6 Ω
—	Resistor, 100k, 1/6 Ω
—	Potentiometer, 10k
—	IC socket, 20 machined pins
—	PICAXE-20X2
H1	Female header, straight, 16 pins
H2	Female header, right-angle, 3 pins (see text)
H3	Female header, right-angle, 4 pins
H4, H6	Female header, right-angle, 2 pins
H5	Female header, straight, 4 pins
—	Male header, double-ended, two 2-pin pieces

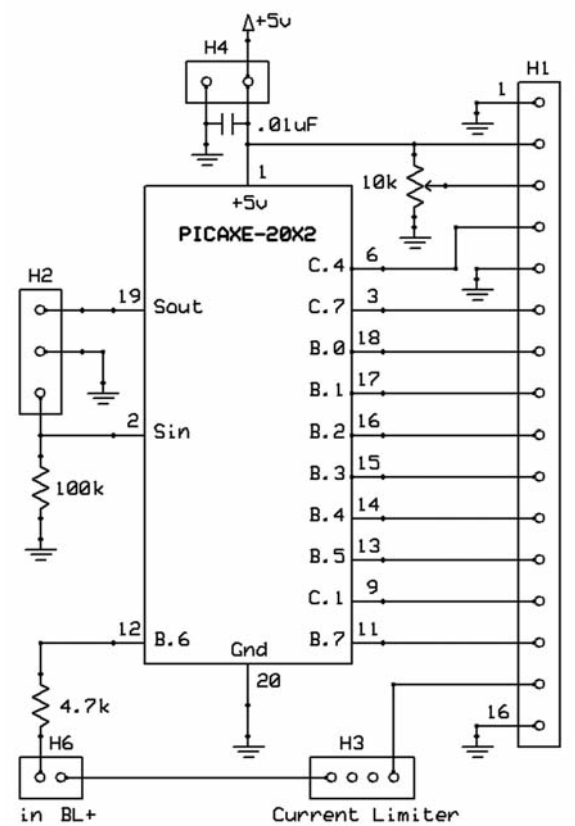


Figure 10-1 Schematic for the serialized LCD project

how to use the programming adapter header.) One other minor point with respect to header H2: I don't carry three-pin, right-angle female headers, but it's a simple matter to start with a four-pin header, pull out one pin (from the bottom) from one end of the header, and snip and sand or file the excess plastic.

Figure 10-2 is the stripboard layout for the project. As you can see, all the other headers serve the same functions that we implemented in our previous project. H1 is where the LCD connector gets inserted, H3 is for the current-limiting resistor, H4 and H6 are intended to accept double-ended male headers (available on my website) that are long enough to also insert into the breadboard (and to bend if you want to adjust the angle at which the LCD sits), and H5 is a straight female header that has no electrical function—it just supports the LCD so that it sits parallel to the stripboard, even though this is definitely not a parallel LCD!

As usual, read through the complete list of assembly instructions that follows to be sure you understand the entire procedure before assembling the board:

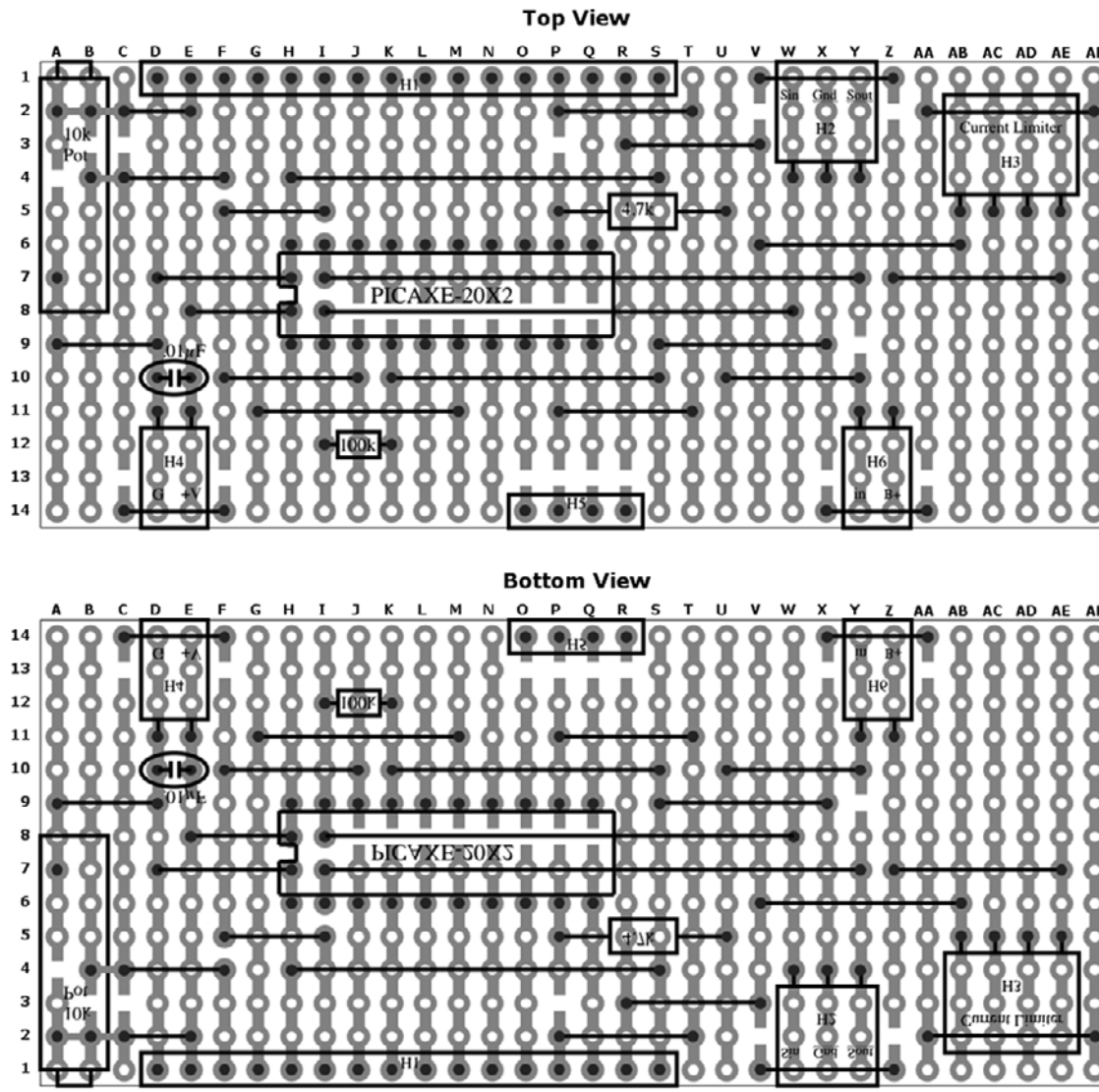


Figure 10-2 Stripboard layout for serialized LCD project

1. Cut and sand a piece of stripboard to the required size (32 traces with 14 holes each).
2. Sever the traces on the bottom of the board as indicated in Figure 10-2.
3. Clean the bottom of the board with a plastic Scotch-Brite or similar abrasive pad.
4. Insert the jumper from C2 to E2. Bend the lead flat against the board from C2 to A2 on the bottom of the board and snip it so that it just reaches A2. Solder the jumper at E2, but *not yet* at C2, B2, or A2.
5. Insert the jumper from C4 to F4. Bend the lead flat against the board from C4 to B4 on the bottom of the board and snip it so that it just reaches B4. Solder the jumper at F4, but *not yet* at C4 or B4.
6. Insert all the remaining jumpers, except the four that go over the tops of the right-angle female headers; solder and snip the leads.
7. Insert the two resistors; solder and snip their leads.
8. Insert all the right-angle female headers. Flip the board over with the headers in place, set it on a flat surface, and solder all the pins.
9. For each of the four right-angle female headers, bend and insert a jumper so that it spans the top of the header as shown in Figure 10-2. Using a small clamp to hold the header and the jumper tightly against the top of the board, solder and snip the jumper leads on the bottom of the board.
10. Insert the capacitor; solder and snip its leads.
11. Insert and solder the 20-pin socket in place (make sure pin 1 is at H9).
12. Insert the 10k potentiometer. Flip the board over and support it on a flat surface.
13. Make sure the unsoldered jumper leads are making contact with the appropriate potentiometer pins; solder one lead at A2, B2, and C2 and the other lead at B4 and C4.
14. Solder the remaining potentiometer pin at A7.
15. Insert the 16-pin and 4-pin straight female headers. Flip the board over, support the headers on a flat surface, and solder the headers in place.
16. File or sand all the cut leads on the bottom of the board.
17. Clean the flux from the bottom of the board and allow it to dry.
18. Inspect the board carefully for accidental solder connections or other problems.

Testing the Completed LCD Board

Before using any stripboard circuit that includes an on-board processor, it's always a good idea to test the board for any wiring problems (especially in the power connections) before inserting the processor into the board. To test the serial LCD board, insert any size resistor into H3 and use two 2-pin pieces of a double-ended male header to attach the completed stripboard to a powered breadboard via H4 and H6. Using the header labels shown earlier in Figure 10-2, connect +V and B+ to the +5V power rail and connect G to the ground rail (see Figure 10-3). Turn on the breadboard power supply and use a multimeter to test for the presence of +5V at pin 1 of the IC (integrated circuit) socket, and pins 2 and 15 of H1. Also test all the other pins of the IC socket and H1 to be sure that +5V is *not* present on any other pin.

Testing the Programming Connection to the LCD Board

Next, power-down the breadboard, insert a 20X2 into the IC socket so that pin 1 is at the lower-left corner, and insert a resistorized LED into H1 so

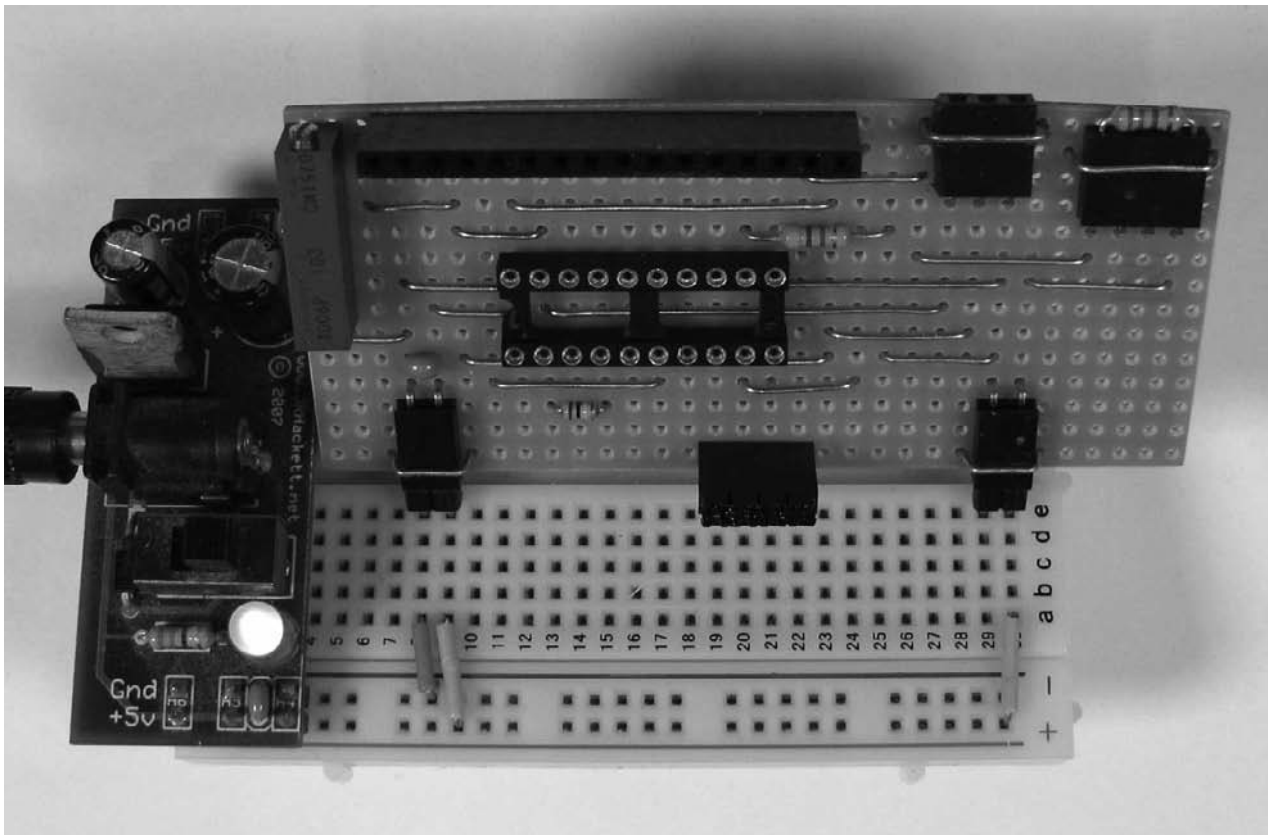


Figure 10-3 Breadboard setup for testing the completed LCD board

that its +5V lead is inserted into pin 14 of H1 and its Ground lead is inserted into pin 16 of H1. (This arrangement connects the LED to pin B.7 of the 20X2.) Finally, insert the AXE027 USB cable into the USBS-PA3 programming adapter and insert the adapter into H2 so that the USB cable extends behind the stripboard (see Figure 10-4). When the adapter is inserted correctly, the *Serin* connection will be on the left (looking from the front of the stripboard) and *Serout* will be on the right.

The following "bare-bones" code snippet is a simple "Hello World" program to test the programming connection to the LCD board. Type it into the Programming Editor and download it to the 20X2 on the LCD board. The LED should blink; if not, you will need to troubleshoot the wiring connections on your stripboard.

```
'=== HelloLCD.bas ===
  #com 4
  #picaxe 20X2
  #no_data
  #no_table
do
  toggle B.7
  wait 1
loop
```

Testing the LCD Interface

When you're sure that the programming interface is functioning correctly, we're ready to test the interface between the on-board 20X2 and the LCD. The only difference between this interface and the one we used in the previous chapter is that we can no longer use the 20X2's B.6 pin for output to the LCD. The software that implements this change

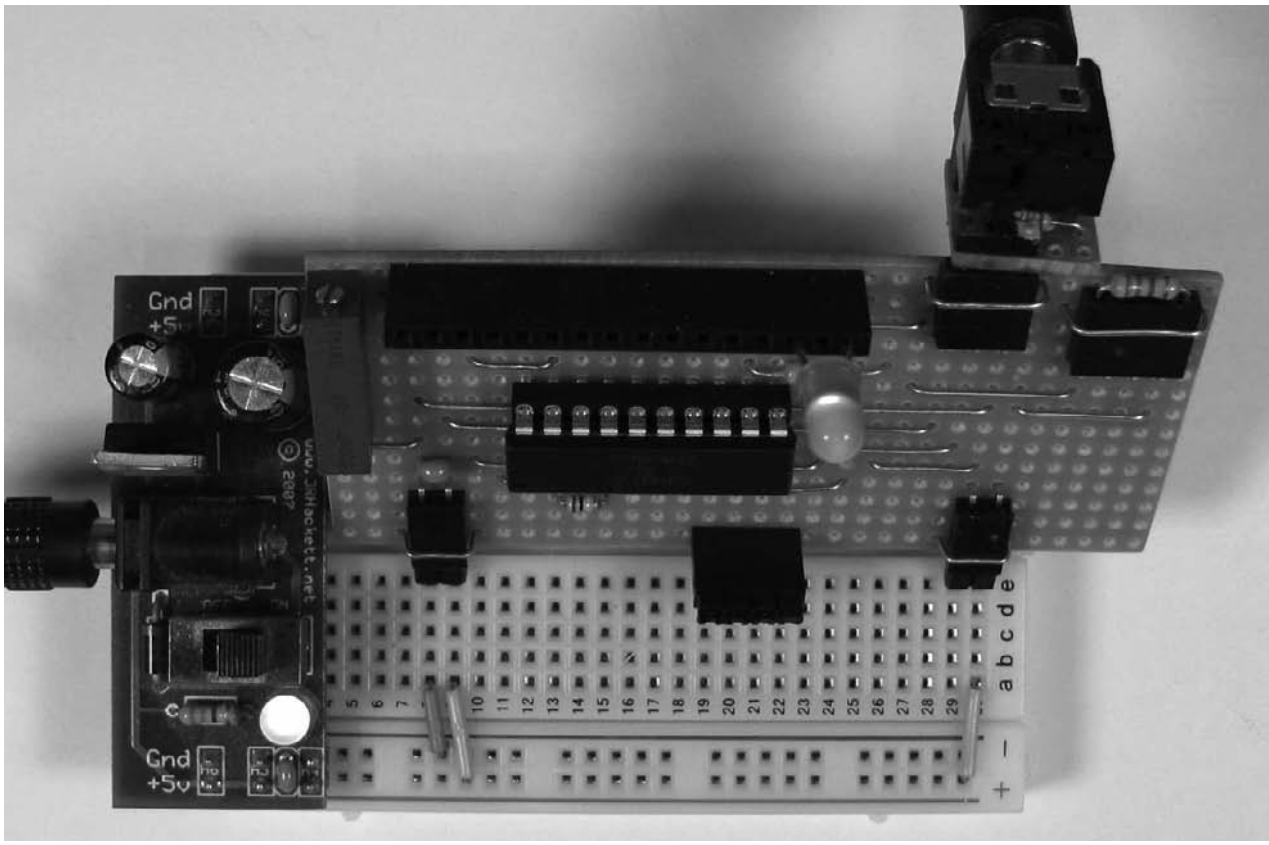


Figure 10-4 Breadboard setup for testing the programming connection to the LCD board

(LCDtest.bas) is presented in Listing 10-1. It's essentially the same as the program we worked with in Chapter 9, with three simple changes that handle the necessary pin substitution. First, in the "Variables" section, I have added a declaration for the replacement pin (*symbol newB.6 = outpinC.1*). Second, at the beginning of the main program, the *dirsB* statement (*dirsB = %10111111*) has been changed to reflect the fact that pin B.6 will now be an input. Third, near the end of the "OutCmd & OutTxt" subroutine (right after the *outpinsB = char* statement), I added the following statement: *newB.6 = bit6*, which probably requires a quick explanation. The *newB.6 = bit6* statement works because the *char* variable has been assigned to variable b0 and, as you remember, the bits of b0 are individually accessible. If B.6 were still an output, the *outpinsB = char* statement would have assigned bit6 of *char* to B.6. However, we have now defined B.6 as an input, so nothing is assigned to it by the *outpinsB = char* statement. The additional statement (*newB.6 = bit6*) simply corrects this omission by assigning bit6 of *char* to *newB.6,* which, of course, is actually *pinC.1*. Download LCDtest.bas from my website and use the Programming Editor to download it to the 20X2 on the LCD board. You should see the two-line message repetitively appearing on the LCD; if not, you will need to troubleshoot the wiring connections on your stripboard.

LISTING 10-1

```
' ================================ LCDtest.bas ==================================
' Program runs on a PICAXE-20X2 at 8MHz & transmits 8-bit data
' to an HD44780 16x2 LCD display.


' === Constants ===
  symbol enable = C.7                    ' LCD enable pin connected to C.7
  symbol RegSel = C.4                    ' LCD RegSel pin connected to C.4


' === Variables ===
  symbol   char = b0                     ' character to be sent to LCD
  symbol  index = b1                     ' used as counter in For-Next loops
  symbol newB.6 = outpinC.1              ' replacement pin for B.6 (hserin)
' Note:  newB.6 is a variable because its value can change


' === Directives ===
  #com 3                                 ' specify download port
  #picaxe 20X2                           ' specify processor
  #no_data                               ' save time downloading
  #terminal off                          ' make sure terminal is off


' === Table ======================
  Table  0, ("Evil Genius LCD ")
  Table 16, ("Serial Display! ")
' ============================ Begin Main Program ===============================

  dirsB = %10111111                      ' set portB as outputs (except B.6)
  dirsC = %10111111                      ' set portC as outputs (except C.6)
                                         ' *** Initialize the LCD ***
  pause 200                              ' pause 200 mS for LCD initialization
  char = 56                              ' setup for 8 bits, 2 lines, 5X7 dots
  gosub OutCmd                           ' send instruction to LCD
  char = 12                              ' display on, cursor off
  gosub OutCmd                           ' send instruction to LCD

' *** Main Program Loop - Send data to the LCD ***
do
  char = 1                               ' clear display & go home
  gosub OutCmd                           ' send instruction to LCD
  wait 1
  for index = 0 to 15
    readtable index, char                ' send line one to LCD
    gosub OutTxt
  next index
```

LISTING 10-1 (continued)

```
  wait 1
  char = 192                              ' move cursor to start of line two
  gosub OutCmd                            ' send instruction to LCD
  for index = 16 to 31                    ' send line two to LCD
    readtable index, char
    gosub OutTxt
  next index
  wait 4
loop


' ==================== End Main Program - Subroutines Follow ====================

OutCmd:
  Low RegSel                              ' set up for command byte
  goto Doit                               ' do it
OutTxt:
  High RegSel                             ' set up for text byte
Doit:
  outpinsB = char                         ' load byte onto outpinsB
  newB.6 = bit6                           ' load bit6 onto pinC.1
  pulsout enable,1                        ' send data
  return
```

Installing the LCD Driver Software

When you are sure the LCD interface is functioning properly, we're ready to take a look at the driver software we need to transform the LCD board into a stand-alone serial peripheral that can be used in any PICAXE project. The following software (LCDhserinDriver.bas) may look a little intimidating simply due to its length (Listing 10-2). However, if we break it down to its functional parts, it's really not that difficult.

Let's begin with the part that I'm not going to explain. (How oxymoronic is that?) In the September 2009 installment of the "PICAXE Primer" in *Nuts and Volts* magazine, I went on a little bit of a rant. I'll spare you the details, but let me summarize: There are five lowercase letters (g, j, p, q, and y) in the modern English alphabet that should have "descenders," which is a term that refers to the portion of the letter that should be written below the main line of the text. Unfortunately, typical LCD character sets ignore this fact and print those letters without any descenders. In my opinion, this practice greatly detracts from the readability of the display; to see what I mean, look back at the letter "g" in the word "again" that was shown in Figure 9-6 of the previous chapter.

Fortunately, all HD44780 LCDs include on-board memory space for eight "custom characters." In the "PICAXE Primer" article, I detailed the process of creating custom characters for the purpose of improving the display of the five lowercase letters with descenders. If you are interested in the details, you may want to obtain a copy of that article. I'll just briefly summarize how the program implements the replacement of the "offending" characters. First, the data statements

LISTING 10-2

```
' ============================== LCDhserinDriver.bas ==============================
' Program runs on a PICAXE-20X2. It receives serial data from a
' master processor & displays it on the EG serialized 16x2 LCD.



' === Constants ===
  symbol enable = C.7                     ' LCD enable pin
  symbol RegSel = C.4                     ' LCD RegSel pin connected to C.4


' === Variables ===
  symbol char   = b0                      ' character to be sent to LCD
  symbol LstChr = b1                      ' used to access hserin data
  symbol index  = b2                      ' used in for/next loop
  symbol newB.6 = pinC.1                  ' replacement pin for B.6 (hserin)
' Note:  newB.6 is a variable because its value can change



' === Directives ===
  #com 4                                    ' specify COM port
  #picaxe 20X2                              ' specify compiler mode
  #no_data                                  ' save time downloading
  #terminal off                             ' make sure terminal is off


' ================================================================================
' The following are custom characters with true descenders
' that we will use in place of the built-in versions.
' Note: these cannot be deleted without major program changes.
' ================================================================================


' === Data ===
' Lowercase "g"
  Data (%00000)
  Data (%00000)
  Data (%01111)          '  ****
  Data (%10001)          ' *    *
  Data (%10001)          ' *    *
  Data (%01111)          '  ****
  Data (%00001)          '      *
  Data (%01110)          '  ***

' Lowercase "p"
  Data (%00000)
```

LISTING 10-2 (continued)

```
    Data (%00000)
    Data (%10110)          ' * **
    Data (%11001)          ' **  *
    Data (%10001)          ' *   *
    Data (%11110)          ' ****
    Data (%10000)          ' *
    Data (%10000)          ' *

  ' Lowercase "q"
    Data (%00000)
    Data (%00000)
    Data (%01101)          '  ** *
    Data (%10011)          ' *  **
    Data (%10001)          ' *   *
    Data (%01111)          '  ****
    Data (%00001)          '     *
    Data (%00001)          '     *

  ' Lowercase "j"
    Data (%00010)          '    *
    Data (%00000)          '
    Data (%00110)          '   **
    Data (%00010)          '    *
    Data (%00010)          '    *
    Data (%00010)          '    *
    Data (%10010)          ' *  *
    Data (%01100)          '  **

  ' Lowercase "y"
    Data (%00000)
    Data (%00000)
    Data (%10001)          ' *   *
    Data (%10001)          ' *   *
    Data (%10001)          ' *   *
    Data (%01111)          '  ****
    Data (%00001)          '     *
    Data (%01110)          '  ***

  ' Bullet
    Data (%00000)
    Data (%01110)          '  ***
    Data (%11111)          ' *****
    Data (%11111)          ' *****
    Data (%11111)          ' *****
    Data (%01110)          '  ***
    Data (%00000)
    Data (%00000)
```

(continued)

LISTING 10-2 (continued)

```
' Up-Arrow
  Data (%00100)          '    *
  Data (%01110)          '   ***
  Data (%10101)          ' * * * *
  Data (%00100)          '    *
  Data (%00100)          '    *
  Data (%00100)          '    *
  Data (%00100)          '    *
  Data (%00000)

' Down-Arrow
  Data (%00100)          '    *
  Data (%00100)          '    *
  Data (%00100)          '    *
  Data (%00100)          '    *
  Data (%10101)          ' * * * *
  Data (%01110)          '   ***
  Data (%00100)          '    *
  Data (%00000)


' ============================== Begin Main Program ================================

  dirsB = %10111111                       ' configure portB (B.6 is hserin pin)
  dirsC = %10111111                       ' configure portC (C.6 is input only)
  hsersetup B4800_64, %111                ' setup for background serial receive


' === Initialize the LCD ===
  pause 200                               ' pause 200 mS for LCD initialization
  pinsB = 56                              ' setup for 8 bits, 2 lines, 5X8 dots
  pulsout enable,1                        ' send command
  char  = 12                              ' display on, cursor off
  gosub OutByte                           ' send instruction to LCD

' === Load custom chars into CGRAM ===
  char = 64                               ' command: define custom characters
  low RegSel                              ' set up for command
  gosub OutByte2
  high RegSel                             ' set up for text
  for index = 0 to 63                     ' install 8 custom characters
    read index, char
    gosub OutByte2
  next index
  char   = 1                              ' clear display and go home
  low RegSel                              ' this must be located here
  gosub OutByte2                          ' to "turn off" load custom chars
```

LISTING 10-2 (continued)

```
' ==== Main Program Loop - Receive data and display it on LCD ====
  setfreq m64
do
  if hserinflag = 1 then                    ' if background receive has occurred
    pause 320                               ' allow 40mS for max of 20 characters
    LstChr = hserptr - 1                    ' get address of last byte received
    for ptr = 0 to LstChr                   ' for each scratchpad address,
      char = @ptr                           ' fetch character (indirect addressing)
      gosub OutByte                         ' process char and send it to LCD
    next ptr
    hserinflag = 0                          ' reset the new background data flag
    hserptr = 0                             ' reset the data write pointer
  endif
loop


' ==================== End Main Program - Subroutines Follow ======================

OutByte:
  select case char
    case 0 to 31                            ' commands
      low RegSel                            ' set up for command
    case 128 to 207                         ' provides capability to move the
      low RegSel                            ' cursor to any on-screen location
    case 103,106,112,113 ' "g", "j", "p", & "q"
      char = char - 103
      high RegSel                           ' set up for text
    case 121                                ' "y"
      char = 4
      high RegSel                           ' set up for text
    else                                    ' all other text
      high RegSel                           ' set up for text
  endselect
Outbyte2:
  outpinsB = char                           ' load character onto outpinsB
  newB.6 = bit6                             ' substitute pinC.1 for pinB.6
  pulsout enable,1                          ' send data
  return
```

define the improved dot patterns for each of the five characters and, just because there was room, I included three others: a bullet, an up arrow, and a down arrow. (If you want to display one of those three characters, you can do so by sending a "5," "6," or "7" character to the LCD.) The section labeled "Load custom chars into CGRAM" actually installs those data patterns in the LCD's eight bytes of custom character memory.

In the *OutByte* subroutine, the *select case* statement "intercepts" the ASCII codes for the offending characters and substitutes the correct code for the new and improved characters. The same *select case* statement also determines whether each incoming character is a command byte or a text byte and configures *RegSel* accordingly. The remainder of the subroutine simply sends the character out to the LCD for display, a process we discussed earlier in the "Testing the LCD Interface" section.

At this point, we have covered all the significant aspects of the program except for the main program loop, so let's turn our attention to that. Just before entering that loop, we boost the speed of the 20X2 up to its maximum (*setfreq m64*) so that the transfer of data to the LCD can occur as rapidly as possible. The loop itself consists of one *if* statement that only executes if a serial character has been received in the background; in other words, if *hserinflag* is 0, nothing has been received and the loop just continually repeats without really doing anything. As soon as a character has been received, *hserinflag* is automatically raised to 1 so the *if* statement executes the following tasks:

- **pause 320:** The *hserinflag* is automatically raised as soon as the first character in a string has been received, so the first thing we need to do is pause long enough to allow all the remaining characters to be received. Let's do a little math to see how I arrived at the *pause 320*. We're receiving data at 4800 baud; let's call it 5000 baud to simplify the math. Baud is defined as bits per second and an eight-bit character with its start and stop bits totals ten bits, so 5000 baud equals 500 characters per second. That means it takes about 2mS to transmit one serial character. I have placed an arbitrary length limit of 20 on my serial strings (16 characters with a couple of commands thrown in), so the maximum time it should take to transmit the longest string would be 20 * 2 = 40mS. However, we can't just say *pause 40* because we're running the 20X2 at 64MHz, which is eight times its default speed of 8MHz. Therefore, pauses occur eight times more quickly than they would at 8MHz. To compensate for that, we need to multiply the 40 by 8 to get the correct value (320) for our *pause* statement.
- **LstChr = hserptr – 1:** At this point, we're about to enter a *for…next* loop that sends the received data on to the LCD, so we need to know the address of the last character that was received. Since *hserptr* is automatically incremented to prepare for the reception of the next character, the address of the last character received is simply *hserptr – 1*.
- **for…next loop:** In this loop, we're sending the serial string (one character at a time) to the LCD. The only tricky part is that we're using indirect addressing: Each time through the loop, the *char* variable is assigned the value that is located at the address that is "pointed to" by the *ptr* variable. Once that value has been assigned to *char*, the *OutByte* subroutine is called to send the value to the LCD.
- **hserinflag = 0:** Now that the string has been sent to the LCD, we need to reset *hserinflag* to 0 in preparation for waiting for the next string to begin to arrive.
- **hserptr = 0:** Similarly, we need to reset the data-write pointer so that the first character of the next string will be automatically stored in scratchpad location 0.

Testing the LCD Driver Software

In order to test the LCD driver software, we need to program our 20X2 master processor (not the on-board LCD processor) to send some data to the LCD for display. The following program (Listing 10-3) will certainly get the job done, but you may want to modify it to suit your purposes. The

LISTING 10-3

```
' ============================== SeroutToLCD.bas ================================

' This program runs on a PICAXE-20X2 processor at 8 MHz.
' It sends serial data to the Evil Genius LCD display.
' Note 1: The "Display Clear" command (DC) must always
' be sent on a line by itself (or next character is lost).
' Note 2: A minimum delay of 40mS is required between serial
' transmissions to allow time for the LCD to display the data.


' === Constants ===
  symbol abit  = 40
  symbol toLCD = B.7

' LCD Commands
  symbol DC =   1                          ' display clear & cursor home
  symbol C0 =  12                          ' cursor off
  symbol C1 =  14                          ' cursor on (no blink)
  symbol CB =  15                          ' cursor blink
  symbol L1 = 128                          ' cursor at position 1 of line 1
  symbol L2 = 192                          ' cursor at position 1 of line 2


' === Directives ===
  #com 3                                   ' specify COM port
  #picaxe 20X2                             ' specify compiler mode
  #no_data                                 ' save time downloading
  #no_table                                ' save time downloading
  #terminal off                            ' make sure terminal is off


' ============================== Begin Main Program ================================

  wait 1                                   ' allow time for LCD to initialize
do
  serout toLCD, N4800_8, (DC)
  wait 1
  serout toLCD, N4800_8, (L1,"PICAXE-20X2 LCD ")
  pause abit
  serout toLCD, N4800_8, (L2,"Serial Display ")
  wait 2
  serout toLCD, N4800_8, (L1,"The Evil Genius ")
  pause abit
  serout toLCD, N4800_8, (L2," strikes again! ")
  wait 2
loop
```

program is simple; it just repetitively sends two sets of strings to the serial LCD module. As you can see, I have included six LCD commands in the "Constants" section to get you started, but after reading the HD44780 documentation, you certainly can add any commands that you want to use.

NOTE **Don't forget that you can also send a command to place the cursor at any position on a line, as we discussed in Chapter 9.**

Two important points to keep in mind are noted in the initial comments. First, the "Display Clear" command must be sent by itself, not as part of a longer string, because the display requires a relatively long time to execute this command, so a couple of characters that immediately follow it would be lost if we didn't allow the necessary time. Second, whenever you send two strings "back to back," it's necessary to intersperse a brief delay to allow sufficient time for the LCD module to display the first string before the second one begins to arrive. So far, 40mS has worked fine for me, but you may want to experiment with that value; if the delay is too short, all that will happen is some characters will fail to be displayed on the LCD.

My breadboard setup for the completed project is shown in Figure 10-5. I chose pin B.7 as the serial output pin for the master processor, but you could easily redefine it to whichever pin you prefer to use. First download LCDhserinDriver.bas to the LCD module, and then download SeroutToLCD.bas to your master processor on the breadboard—you should see the two messages repetitively alternating

Figure 10-5 Breadboard setup for completed serial LCD project

Figure 10-6 Comparison of LCD standard and custom characters

on the display. If not, the usual troubleshooting session is in order!

To see the difference that the custom characters make, compare the display in Figure 10-5 with that of Figure 9-6 in Chapter 9. The difference may seem slight, but when there are three or four lowercase letters being displayed at one time, it's more pronounced. Figure 10-6 shows a display from the original "PICAXE Primer" article that includes all five characters for comparison (reprinted with the kind permission of *Nuts and Volts* magazine); line 1 displays the default characters, and line 2 shows the "new and improved" custom characters. Personally, I think that the improved display is worth the additional programming requirements.

With the completion of our serial LCD project, we now have a powerful, stand-alone peripheral display for use with any PICAXE project. All it requires is one output line, and its timing requirements are minimal; just be sure to send any "Display Clear" commands without additional characters in the same string and include a short delay after the transmission of each serial string. If you keep those two caveats in mind, the serial LCD display will be able to keep up with all the data you want to send.

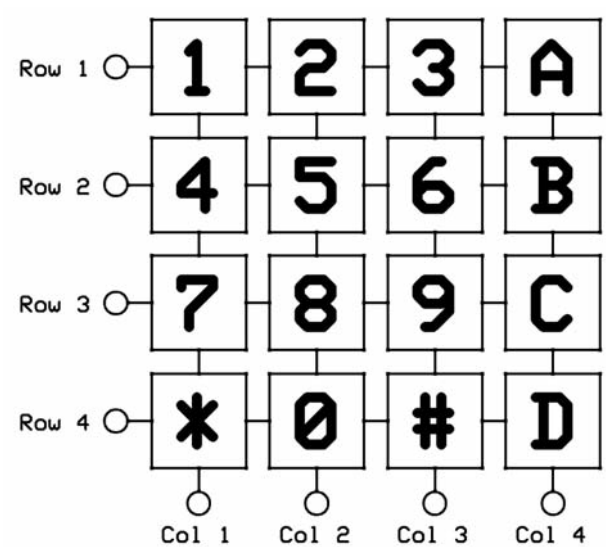
This page intentionally left blank

CHAPTER 11

Interfacing Keypads

Now that we have a versatile output device to use in any PICAXE project, we're going to shift gears and focus on developing an input device for our projects. We're going to start with a standard matrix keypad and, similarly to the LCD project we just completed, serialize it so that it can be used in any project, even one based on the little 08M2.

Matrix keyboards are generally available in two sizes: four rows by three columns and four rows by four columns. The key arrangement on the 4 × 3 version usually matches that of a landline telephone—remember them? The fourth column of a 4 × 4 keypad adds the letters A, B, C, and D, as shown in the following illustration.



We're going to be using a 4 by 4 keypad in this chapter, but the experiments and project can easily be modified for use with a 4 by 3 keypad if you prefer. The specific keypad that appears in the chapter's photos is available on my website, but any matrix keypad you happen to have on hand should work just as well—just be sure that it is, in fact, a matrix layout. I have seen keypads that look similar to a "genuine" matrix, but actually have each key connected to its own line with a common connection to all the keys. It would be possible to use this type of keypad, but the necessary modifications would be extensive.

The keypad that I am using includes a row of eight holes along its bottom edge, into which a header can be soldered so that the keypad can easily be connected to a breadboard, stripboard, or ribbon cable assembly. If you have a keypad that accepts a header along its top edge, it will work fine for the breadboard experiments, but our stripboard project would need to be redesigned to work with such a keypad. Finally, there seems to be a fair amount of variation in the specific order of the connecting points on different matrix keypads. The simplest arrangement for the user would naturally be to have the row and column connections in order. However, probably because the keypad layout is simpler and therefore less expensive, the connections are frequently not in any logical order. If you use a keypad that has its connection points in an order that differs from the

one I'm using, you will need to modify both the breadboard and stripboard layouts. The schematics, of course, will remain the same.

Decoding Matrix Keypads

There are two general methods of decoding a matrix keypad, both of which are based on the fact that whenever a key is pressed on a matrix keypad, the corresponding row trace and column trace are electrically connected. The more common approach to decoding is referred to as "scanning" the keypad. In this case, the keypad rows (or columns) are each connected to an output pin on the microprocessor, and the columns (or rows) are each connected to an input pin that's tied to Ground by a current-limiting resistor. The processor sequentially raises each output to a high level and then checks each of the inputs to see if it has been pulled high. If so, the key at the intersection of the currently active row and column has been pressed.

The second decoding method uses a resistor matrix to produce an analog voltage level that differs for each possible keypress. The analog voltage is connected to an ADC input, and the processor simply converts the resulting voltage level back to the corresponding keypress. This is the approach that we will use in our experiments and project in this chapter, for two reasons. First, it enables us to use an 08M2 as the keypad's peripheral processor because all we need is one ADC pin. More importantly, it frees the master processor from needing to repetitively take ADC readings to be sure it doesn't miss a single keypress whenever it happens. This feature enables our intelligent keypad peripheral to function effectively with any M2-class processor. However, the X2 processors implement two advanced features—hardware interrupts and hardware

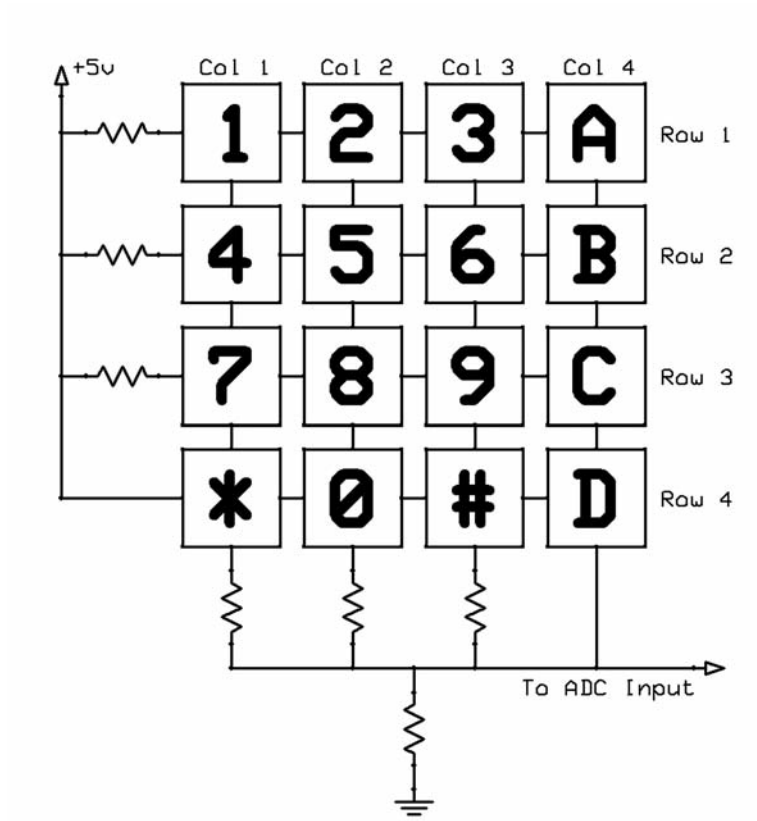


Figure 11-1 Resistor-matrix circuit for producing a range of ADC values

comparators—that enable them to interface directly with a resistor matrix keypad. Therefore, when we get to Chapter 13, we'll also connect a keypad to the 20X2 processor without the use of a peripheral processor at all.

Experiment 1: Interfacing a Matrix Keypad

Figure 11-1 presents one possible arrangement for a resistor matrix that can be used in the ADC approach to keypad decoding. As you can see, each possible keypress connects two, one, or zero resistors in series between the +5V line and Ground. By carefully choosing the values of the seven resistors in the layout, it's possible to produce 16 voltage levels that are spread out far enough to easily differentiate. I would like to be able to say that I used a sophisticated mathematical formula to determine the required resistor values, but I actually used a more primitive "trial-and-error" approach to the problem.

Before I discuss my method and the results I obtained, I need to mention the most important factor to keep in mind: Standard 1/4-watt resistors have a 5 percent tolerance rating, which means that a 10k resistor can actually measure anywhere between 9.5k and 10.5k. This is why it's important to be able to produce a wide range of analog voltages. If the ADC readings for two adjacent keys were too close to each other, variations in actual resistor values could result in misidentifying the specific key that has been pressed.

In order to make my trial-and-error approach as painless as possible, I set up a simple Excel spreadsheet to compute the ADC values that would result from a specific combination of resistors and then tried various combinations until I found one that worked. When I used the 256 levels provided by the *readadc* command, some of the ADC values for two adjacent keys were so close (differences of four or five) that I was concerned that errors could result. Switching to the 1024 levels provided by the *readadc10* command greatly simplified the task.

The resistor values that I finally chose are presented in Figure 11-2. Each of the 16 "key" positions includes two pieces of relevant data: the total resistance that is connected in series with the 10k base resistor when the corresponding key is pressed, and (in parentheses) the resulting value produced by the *readadc10* command. If you are unsure about the formula to use to compute the ADC values, refer back to our discussion in Chapter 6. Finally, I need to emphasize that these are theoretical results; your specific ADC values will almost certainly be somewhat different. We'll confront that issue in the next section when we actually construct and test our breadboard circuit.

Constructing a Breadboard Circuit to Interface a Matrix Keypad

The schematic for our breadboard circuit is presented in Figure 11-3. As you can see, the eight connections to the keyboard that I am using are not logically ordered, but it really doesn't matter much. The important thing is to make sure that the connections for each resistor are the same as the ones presented earlier in Figure 11-2. If the pin-out is different for the keypad you intend to use, simply rearrange the connections appropriately.

The parts list for our breadboard circuit is again too simple to warrant a table; just a matrix keypad with a male header, the seven resistors shown in the schematic, and an 08M2 processor. My breadboard layout is shown in Figure 11-4. You can see that the location of the keypad header is somewhat inconvenient for a breadboard circuit; it would be better if the header were at the top of the keypad so that it could connect near the bottom edge of the breadboard and therefore be closer to the user. However, when we get to our stripboard project, the reverse will be true—having the

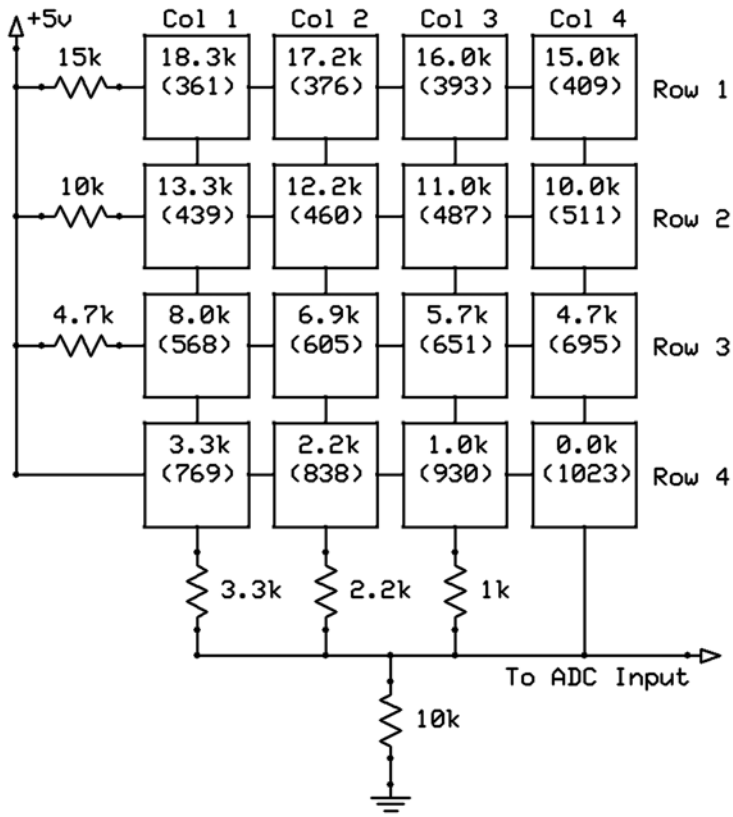


Figure 11-2 Specific resistor-matrix values and corresponding ADC values

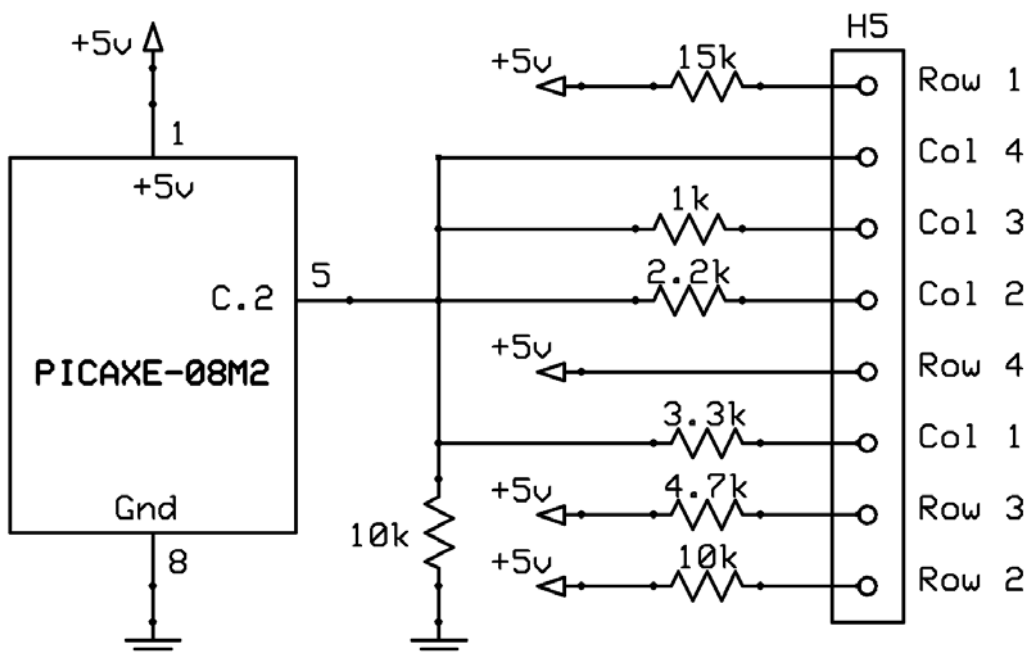


Figure 11-3 Schematic for the matrix keypad circuit

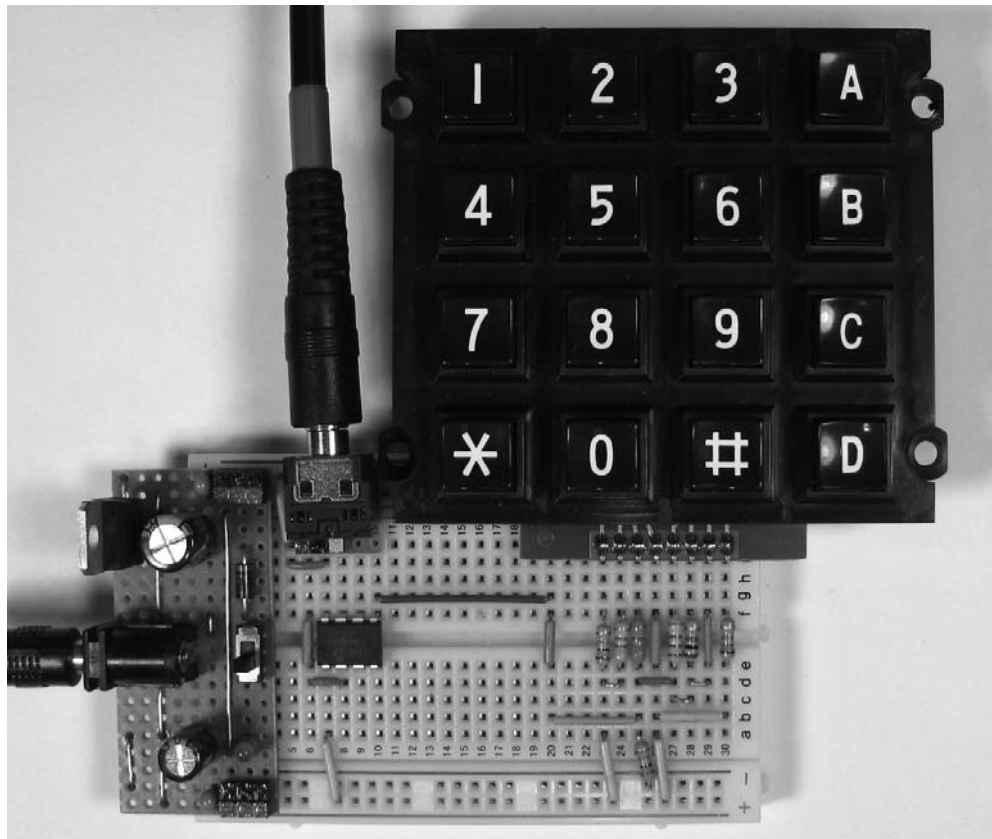


Figure 11-4 Breadboard layout for the matrix keypad circuit

keypad header on the bottom will greatly simplify the stripboard circuit, as well as the connection to the master processor.

Testing the Breadboard Circuit

When you have assembled your breadboard circuit, we're ready to discuss our first program (keypad1.bas), which is presented in Listing 11-1. Its purpose is to determine how closely our "real-world" ADC values match the theoretical values that were presented back in Figure 11-2. The main program loop repetitively carries out the following tasks:

- Wait for keypress (theoretically, *junk* = 0 if no key is pressed, but "< 5" is safer).
- Pause for 50mS to debounce the switch (the *pause 100* is halved at 8MHz).
- Get ADC value (using *readadc10*).
- Wait until switch is released.
- Send the digits of the value to the terminal window (followed by *CR* and *LF*).

Download keypad1.bas and use the Programming Editor to download it to your breadboard circuit. We're going to use the results from this experiment when we get to our next experiment, so you will need to jot them down. There should be enough blank space in Figure 11-2 for you to write your actual ADC reading in each cell. If not (or if you prefer not to write in the book), just jot them down on a piece of paper. Press each key a few times—you may get slightly variable readings for some of the keys. If so, just choose the most typical value for each key and make a note of it. (As I mentioned earlier, your results will most likely differ somewhat from the theoretical values.)

LISTING 11-1

```
' ================================ keypad1.bas ==================================
' This program uses an ADC approach to decoding a matrix keypad.
' It sends the keypress ADC key value to the terminal window.

' === Variables ===
  symbol key  = w0                    ' used in adc10; word variable is required
  symbol junk = b2                    ' throwaway variable used for debouncing

' === Directives ===
  #com 3                              ' specify serial port
  #picaxe 08M2                        ' specify processor
  #terminal 9600                      ' open terminal (8MHz produces 9600 BAUD)

' ============================== Begin Main Program ==============================
  setfreq m8
  dirsC = %00010011

do
  wait_for_keypress:
    readadc C.2, junk
    if junk < 5 then wait_for_keypress
    pause 100                         ' debounce keypress
    readadc10 C.2, key                ' get ADC value

  wait_for_release:
    readadc C.2, junk
    if junk > 5 then wait_for_release
    sertxd ("key = ",#key,cr,lf)      ' send ADC value to terminal
loop
```

Experiment 2: Decoding the Keypresses

Now that you know the ADC value that's produced by each keypress in your breadboard setup, we need to modify keypad1.bas so that it actually decodes each keypress and outputs the appropriate character. Naturally, we'll need another variable in which to store the resulting character; let's call it *char*. Also, we're going to use a *select case* statement to accomplish the decoding, but we can't use a series of equalities to convert each of the ADC values to the correct character because (as we just discovered) some of the values vary slightly. Also, additional slight variations can be introduced by changes in temperature and other factors. To make sure that we always decode the correct character, we'll use a series of "less than" phrases in our *select case* statement and work our way up the list. Using this approach, our *select case* statement will take the form of the following code fragment:

```
select case key
  case < ?
    char = 49      ' ASCII code for "1"
  case < ?
    char = 50      ' ASCII code for "2"
  ' etc., etc.
```

The question remains: What specific values should we use in place of the question marks? The safest (i.e., most error-free) choice is the midpoint between each pair of adjacent keys. I'll use our theoretical values to clarify this point, but you should substitute the actual values you obtained from running the keypad1.bas program. The theoretical value associated with the "1" key is 361, and the value associated with "2" is 376, so the midpoint is about 369. Therefore, in our *select case* statement, we'll say that any value less than 369 will be decoded as the "1" character. If we use the same approach to each of the characters, our select case statement becomes:

```
select case key
  case < 369
       char = 49  ' ASCII code for "1"
  case < 385
       char = 50  ' ASCII code for "2"
  ' etc., etc.
```

Since our *select case* statement involves 16 distinct cases, it will be fairly long to print out. To compress it a bit, we're going to use a little shortcut. Similarly to many dialects of BASIC, PICAXE BASIC supports the use of the colon symbol (:) to separate multiple statements on the same line, which means we can write:

```
select case key
  case < 369 : char = 49  ' ASCII code
                          for "1"
  case < 385 : char = 50  ' ASCII code
                          for "2"
  case < 402 : char = 51  ' ASCII code
                          for "3"
  case < 434 : char = 65  ' ASCII code
                          for "A"
  ' etc., etc.
```

The ":" shortcut is not something I'm suggesting you use frequently in your programs; it can easily make code much more difficult to read. However, our long *select case* statement is perfectly readable in this form, and also much shorter. One final point: Don't forget that as soon as one of the *case* conditions evaluates to true, the associated code is executed and the remainder of the *select case* statement is skipped. For example, if *key* equals 375, *char* is set to 50 and the compiler jumps ahead to the program line that follows the *endselect* statement.

Our second program (keypad2.bas), which is presented in Listing 11-2, incorporates all the changes we just discussed. Download it and use the Programming Editor to change the theoretical midpoint values in each of the *case* conditions to the values you calculated from the results of running keypad1.bas. Also note that the *sertxd* statement has changed: Instead of sending the individual digits of the ADC value, we're now transmitting the appropriate ASCII value for each character to the terminal window. Download the program to your breadboard circuit and test all the keypresses. You should see the correct character appear in the terminal window in response to each keypress. If not, a little troubleshooting is in order. When everything is working correctly, we're ready to convert our breadboard circuit to a stand-alone peripheral device.

LISTING 11-2

```
' ================================ keypad2.bas ==================================
' This program uses an ADC approach to decoding a matrix keypad.
' It sends the decoded character to the terminal window.

' === Variables ===
  symbol key  = w0                    ' used in adc10; word variable is required
  symbol char = b2                    ' the char that corresponds to key value
  symbol junk = b3                    ' throwaway variable used for debouncing

' === Directives ===
  #com 3                              ' specify serial port
  #picaxe 08M2                        ' specify processor
  #terminal 9600                      ' open terminal


' ============================= Begin Main Program ==============================
  setfreq m8
  dirsC = %00010011

do
  wait_for_keypress:
    readadc 2, junk
    if junk < 5 then wait_for_keypress
    pause 100                         ' debounce keypress
    readadc10 2, key                  ' get ADC value

  wait_for_release:
    readadc 2, junk
    if junk > 5 then wait_for_release

    select case key                   ' decode keypress
    ' (Replace the theoretical values with actual values.)
       case < 369 : char = 49         ' 1
       case < 385 : char = 50         ' 2
       case < 402 : char = 51         ' 3
       case < 434 : char = 65         ' A
       case < 450 : char = 52         ' 4
       case < 474 : char = 53         ' 5
       case < 500 : char = 54         ' 6
       case < 551 : char = 66         ' B
       case < 587 : char = 65         ' 7
       case < 634 : char = 56         ' 8
       case < 674 : char = 57         ' 9
       case < 733 : char = 67         ' C
       case < 804 : char = 42         ' *
       case < 885 : char = 48         ' 0
       case < 977 : char = 35         ' #
       else       : char = 68         ' D
    end select

    sertxd ("char = ", char, cr, lf)  ' send char to terminal
loop
```

Project 11
Constructing a Serialized 4 by 4 Matrix Keypad

In addition to decoding the keypresses, our stand-alone peripheral keypad needs to be able to transmit the input characters to a master processor. To accomplish this goal, we're going to use the same approach we implemented with our IR TV remote device in Chapter 8. We'll talk more about that after we have constructed and tested the stripboard circuit. The schematic for our project is shown in Figure 11-5, and the stripboard layout is presented in Figure 11-6.

The schematic in Figure 11-5 is similar to that of the breadboard circuit we just discussed, with the addition of five headers that serve the following functions:

H1 and H4: Connect the peripheral keypad to a breadboard circuit

H2: Enables a programming adapter to be connected

H3: Allows for a ribbon cable connection between keypad and master processor

H5: Connects the keypad to the stripboard

With the exception of H5, all the headers are optional. Because the driver software for the project is fairly simple, when you have the keypad functioning correctly, you may have little or no need to reprogram the 08M2, so you certainly could omit H2. Even if you did decide to reprogram the driver, you could easily remove the 08M2 from the stripboard and reprogram it in a breadboard circuit. The remaining three headers (H1, H2, and H4) provide two different methods of connecting the peripheral keypad to a project—if you construct the project for a specific purpose, you could leave off the header(s) that aren't needed. However, for maximum flexibility, you may want to include all the headers in the first version of the project that you construct. Finally, in the layout shown in Figure 11-6, you can see two additional headers (H6 and H7) that are not included in the schematic. They have no electrical function, but when male headers (with their short ends snipped) are inserted into them, they will support the keypad so that it sits parallel to the stripboard when the project is completed.

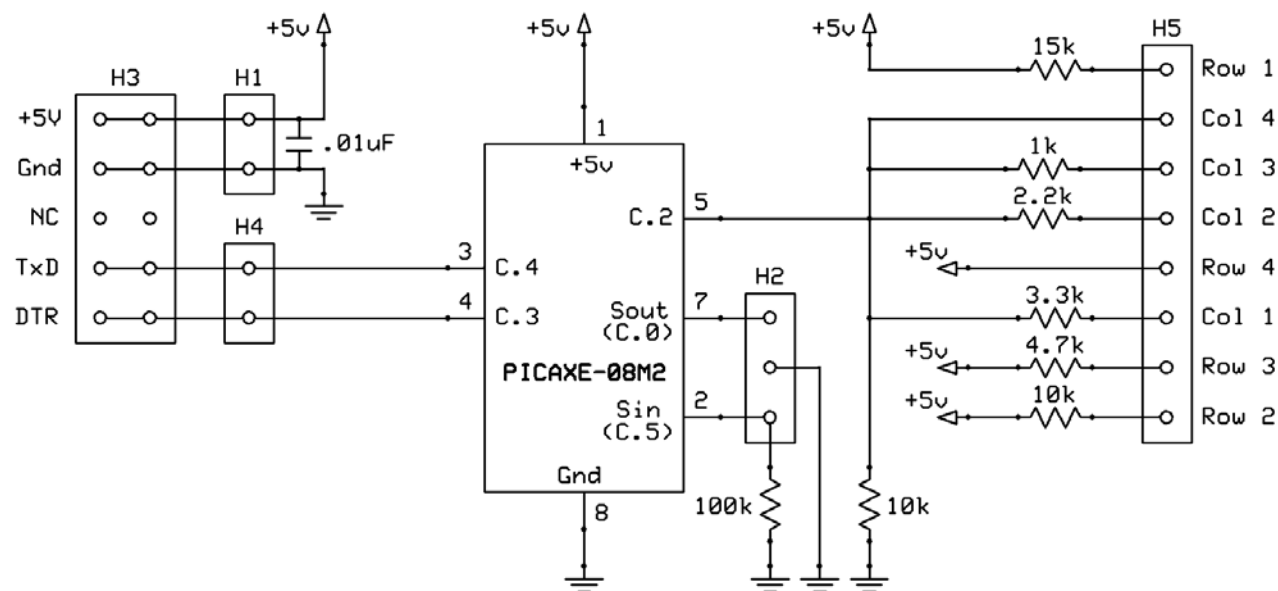


Figure 11-5 Schematic for serial keypad project

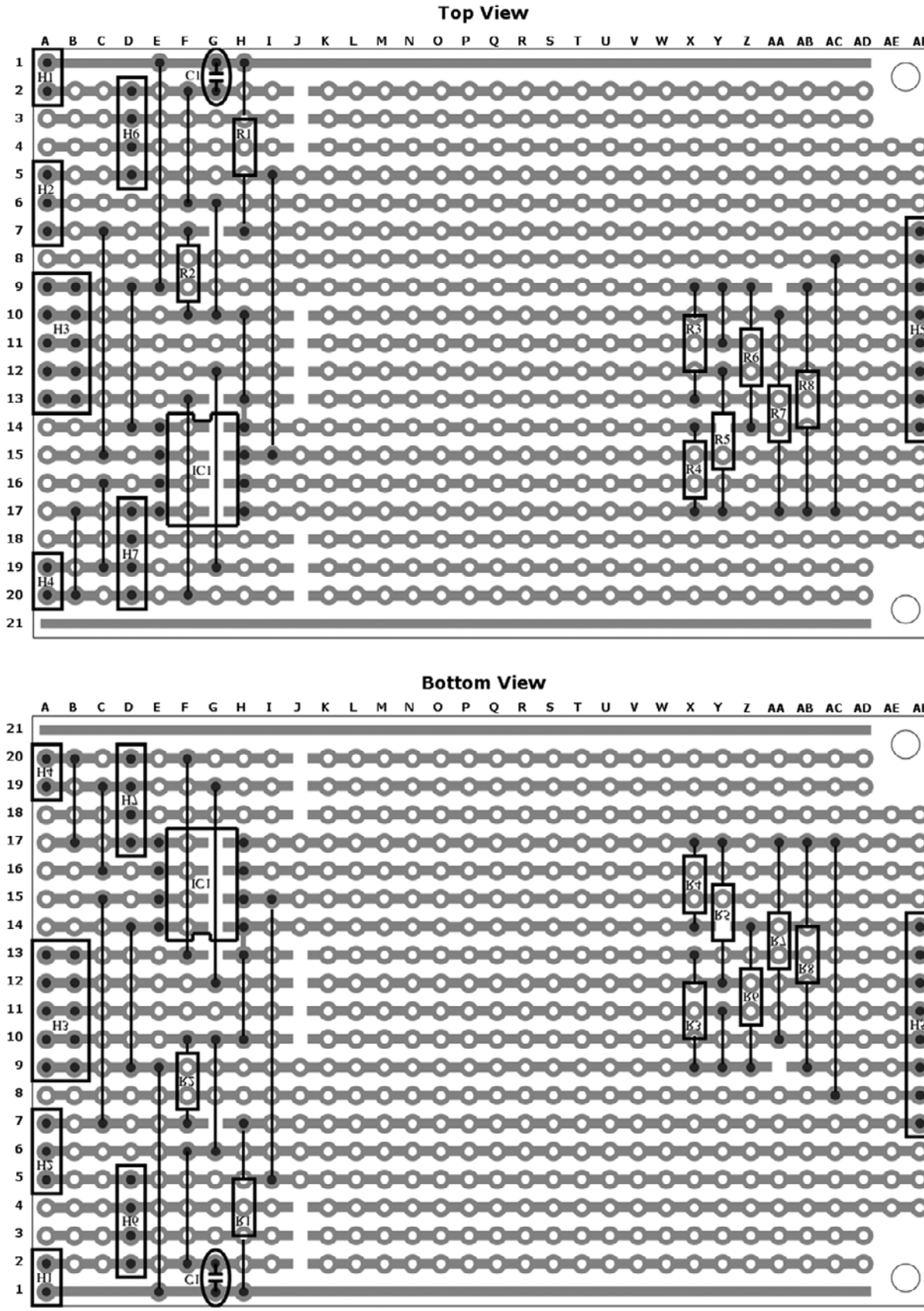


Figure 11-6 Stripboard layout for serial keypad project

The parts list for our serialized keypad project is shown here. Except for headers H2 and H5, all the necessary parts are available on my website. However, I do carry a 16-pin straight female header, from which you can easily snip a 3-pin (H2) and 8-pin (H5) section by "sacrificing" an adjacent pin in each case and sanding or filing the excess plastic.

PARTS BIN

ID	Part
—	Stripboard, 21 traces by 14 holes each
C1	Capacitor, .01μF
R1	Resistor, 15k, 1/4 Ω
R2	Resistor, 100k, 1/4 Ω
R3	Resistor, 4.7k, 1/4 Ω
R4, R6	2 resistors, 10k, 1/4 Ω
R5	Resistor, 3.3k, 1/4 Ω
R7	Resistor, 2.2k, 1/4 Ω
R8	Resistor, 1k, 1/4 Ω
IC1	IC socket, 8 machined pins
—	PICAXE-08M2 (or 08M)
H1, H4	Female header, straight, 2 pins
H2	Female header, straight, 3 pins (see text)
H3	Male header, straight, 5 × 2 pins
H5	Female header, straight, 8 pins (see text)
H6, H7	Female header, straight, 4 pins (see text)

In Figure 11-6, you can see that I am using the small stripboard for the layout, which is why the top and bottom traces contain no holes. If that's what you use, you will need to drill four holes at the appropriate positions in the top trace. If you would rather avoid that chore, you could cut the necessary stripboard from the larger one that's available on my site.

As usual, read through the complete list of assembly instructions that follows to be sure you understand the entire procedure before assembling the board.

1. Cut and sand a piece of stripboard to the required size (21 traces with 32 holes in all but the two edge traces).
2. Using a 3/64-inch (1.2-mm) or 1/16-inch (1.5-mm) bit, drill holes in the solid trace (row 1) at A1, E1, G1, and H1.
3. Sever the traces on the bottom of the board at the 18 holes indicated in Figure 11-6.
4. Clean the bottom of the board with a plastic Scotch-Brite or similar abrasive pad.
5. Insert the jumper from H10 to H13.
6. Flip the board over; bend the lead from H13 to H14, and snip it so that it just reaches the hole at H14. Solder and snip the lead at H10, but *not yet* at H13 or H14.
7. Insert the remaining 11 jumpers; solder and snip the leads.
8. Insert the eight resistors; solder and snip their leads.
9. Insert the capacitor; solder and snip its leads.
10. Insert the machined-pin IC socket. Make sure pin 1 is at E14 and that the jumper lead at H14 touches the pin at H14; solder all eight pins *and* the jumper at H13.
11. Insert the six straight female headers.
12. Flip the board over, support it on a flat surface, and solder the headers in place.
13. File or sand all the cut leads on the bottom of the board.
14. Insert the 5 × 2 male header, either normally on the top or reverse-mounted, depending on whether you want the ribbon cable to attach on the top or bottom.

15. Solder the 5 × 2 male header in place.
16. Clean the flux from the bottom of the board and allow it to dry.
17. Inspect the board carefully for accidental solder connections or other problems.

Testing the Completed Keypad Board

Before we can test the completed keypad board, we need to connect it to a breadboard circuit. You can certainly do this by simply inserting four jumper wires into the sockets of headers H1 and H4, but the keypad would probably slide around as you used it. To avoid this problem, I made a rigid connection between the keypad and the breadboard by constructing two tiny stripboard circuits. I'll describe the process shortly, but I think it will be easier to follow if you first take a look at the completed assembly shown in Figure 11-7. In the photo, you can see that I notched the bottom of my stripboard to match the shape of the keypad board; it makes absolutely no functional difference, but I couldn't help myself!

In Figure 11-7, you can see my two tiny stripboard connectors holding the keypad board tightly against the edge of the breadboard, with the

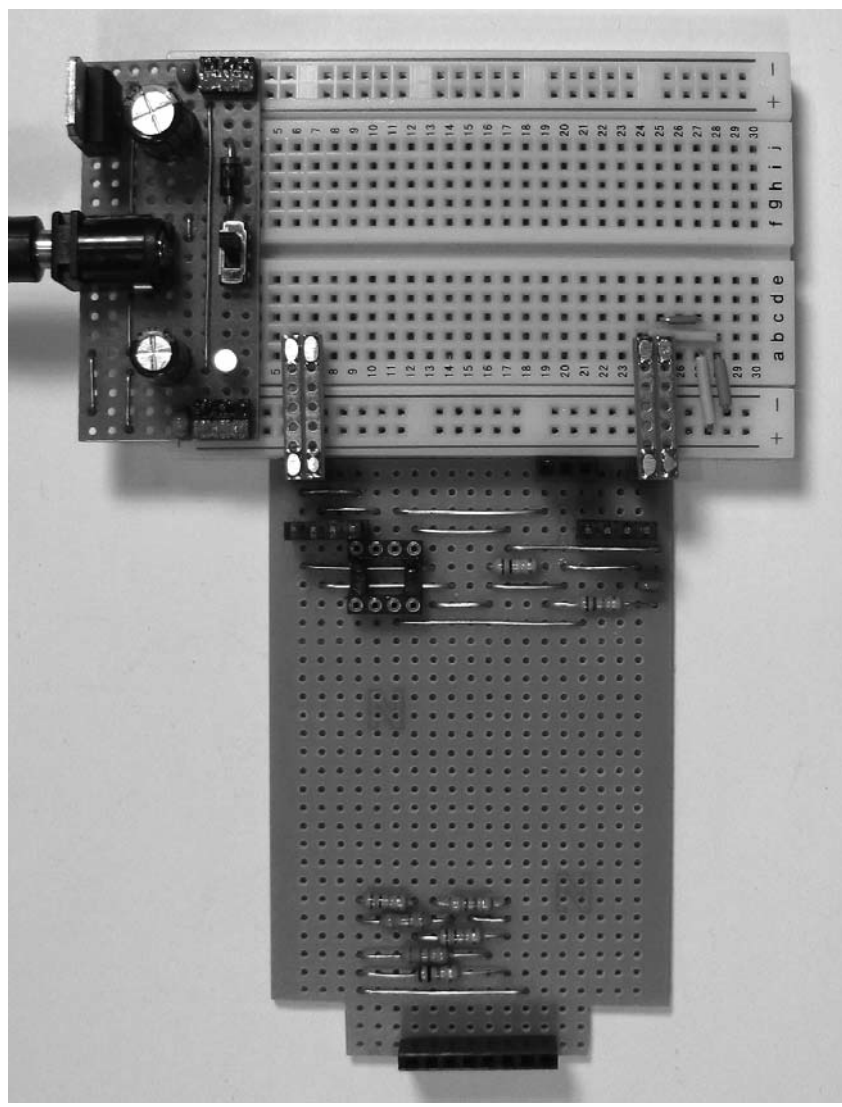


Figure 11-7 Completed board (without keypad) with rigid connection to a breadboard

keypad board's power and ground lines connected to the corresponding rails on the breadboard. If you want to construct the two little stripboard connectors, you can use the following list of assembly instructions for each of them:

1. Cut and sand a piece of stripboard to the required size of two traces with seven holes each. To sand or file the header, it helps to hold it with a pair of pliers—especially if you are using a power sander!
2. Clean the bottom of the board with a plastic Scotch-Brite or similar abrasive pad.
3. Snap two 2-pin pieces from a straight male header and insert the longer ends of each piece into a breadboard so that they have five empty breadboard rows between them.
4. With the traces facing up, mount the stripboard on the short ends of the two headers—the headers should be at the extreme ends of the traces; solder all four pins.
5. Clean the flux from the bottom of the board and allow it to dry.
6. Inspect the board carefully for accidental solder connections. (This may seem like overkill, but you don't want to short out the power and ground connections!)

Test 1

Whether you use the tiny stripboards or jumper wires, connect the keypad board to a powered breadboard. For the first test, don't install the keypad or the 08M2 (see Figure 11-7). Simply connect the keypad board's power and ground lines to those of the breadboard, power the breadboard, and test for +5V at pin 1 and Ground at pin 8 of the IC socket. Also, make sure +5V doesn't appear at any of the other IC pins.

Test 2

For the second test, turn off the power to the breadboard and insert an 08M2 into the IC socket (making sure it's oriented correctly) and connect a USBS-PA3 adapter to H2. (The USBS-PA3 serout pin should be inserted into the header pin closest to H1.) Also, connect an LED from the keypad's "TxD" pin to Ground on the breadboard, as shown in Figure 11-8. (Refer back to Figure 11-5 for the location of the "TxD" pin on header H4.) Power the breadboard and download a simple "Hello World" program that has an LED declared on output C.4. If the LED lights, everything is fine; if not, you will need to troubleshoot the wiring on the keypad board and the connections to the breadboard.

Test 3

For the final test of the board, insert the keypad into the eight-pin header on the board and download the keypad2.bas program we worked with earlier. It should function as it did in our breadboard circuit; each keypress should produce the corresponding character in the Terminal window.

Installing the Keypad Driver Software

When your keypad board has passed our three little tests, we're ready to take a look at the driver software that we need to install in order to transform the keypad into a stand-alone serial peripheral that can be used in any PICAXE project. As I mentioned earlier, we're going to use the same software interface that we did for our IR TV remote project in Chapter 8. As soon as an input character has been decoded, the peripheral processor will raise its output line to a high level to alert the master processor that there's valid data available and then "listen" on its input line for the instruction to send the data. The master processor will send a brief "high" pulse on its output pin that is connected to the 08M2's input when it's ready to receive the data. As we did in Chapter 8, we'll use the *pulsin* command to receive the master processor's "Data Terminal Ready" command. As soon as the peripheral processor has received the

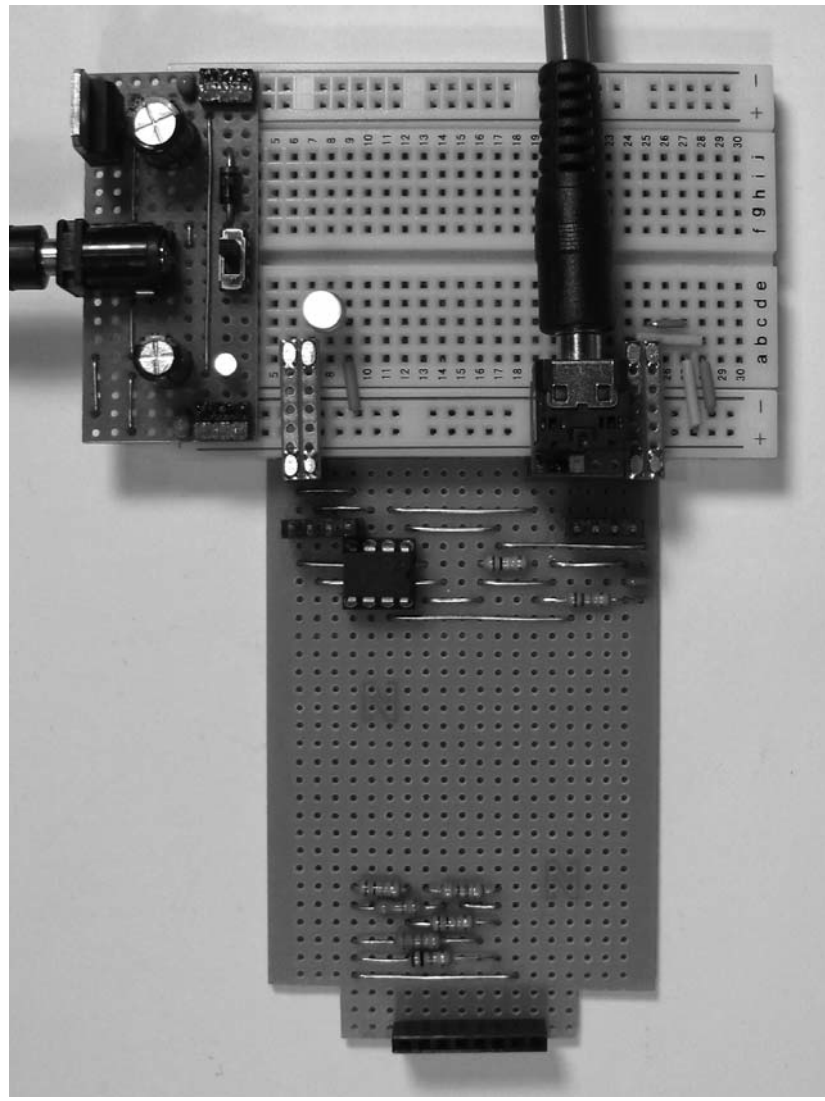


Figure 11-8 Keypad breadboard setup for Test 2

DTR command, it lowers its output pin and sends the character (serially) to the master processor, which is already waiting to receive it. If this explanation isn't clear, you may want to re-read the relevant details we covered back in Chapter 8.

The keypadDriver.bas software is essentially the same as keypad2.bas, so I'm not going to include it here. The only difference is the addition of five statements at the end of the main loop to implement the previous software interface to the master processor. When you download keypadDriver.bas from my website, take a look at those statements before reading further.

Testing the Keypad Driver Software

In order to test the keypad driver software, we first need to set up the hardware interface between the keypad and the master processor. Figure 11-9 presents the hardware setup that I used. In it, the keypad's "TxD" pin is connected to the 20X2's C.6 pin, and the keypad's "DTR" pin is connected to the 20X2's C.7 pin. Since C.6 is going to be receiving serial input, a 1N4148 diode must be connected between the C.6 pin and +5V (with the diode's cathode connected to +5V)—see the *serin* documentation in Part II of the manual for details. Having to add the diode is worth it, because this

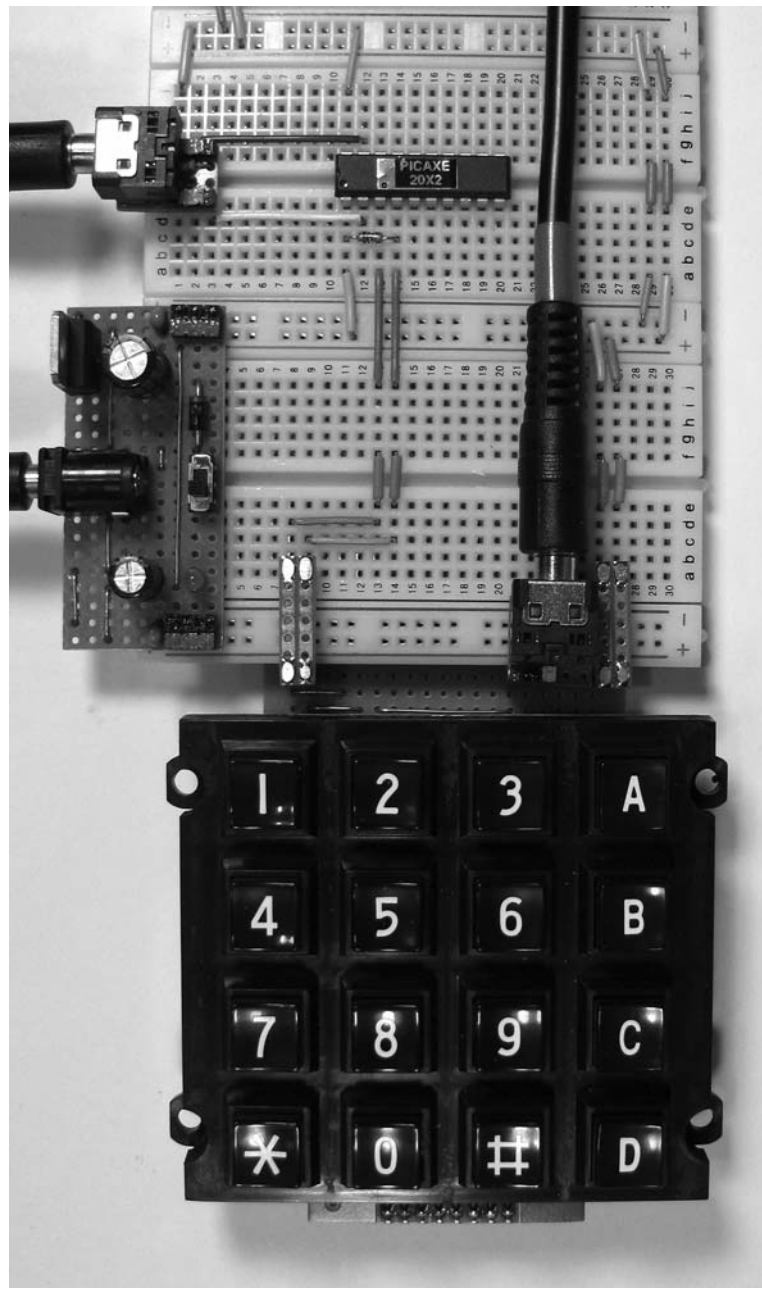


Figure 11-9 Breadboard setup for testing the keypad driver software

arrangement connects an output to a fixed input in both I/O connections, which means that we can skip the usual current-limiting resistor in each line.

When you have completed the hardware interface between the keypad and your master processor, we're ready to move on to our KeypadTest.bas program (Listing 11-3) that the master processor will use to test the system. It's similar to the software we used in Chapter 8, in that the *pause* statement in the main program loop is used to simulate a busy processor that can only check for input periodically. As soon as it gets the signal from the keypad that a character has been entered, the program jumps to a subroutine that implements the software interface we just discussed, gets the character, and sends it to the terminal window for display.

LISTING 11-3

```
' ================================ KeypadTest.bas ================================
' Program runs on a 20X2. It receives data from the keypad
' peripheral & sends it to the terminal window for display.

' === Constants ===
  symbol fromKP = C.6
  symbol toKP   = C.7

' === Variables ===
  symbol char   = b0
  symbol KPflag = pinC.6
' Note:  KPflag is a variable because its value can change

' === Directives ===
  #com 3                                    ' specify com port
  #picaxe 20X2                              ' specify processor
  #no_data                                  ' reduce download time
  #no_table                                 ' reduce download time
  #terminal 9600                            ' open terminal window

' ============================== Begin Main Program ==============================
  dirsc = %10111111
  low toKP                                  ' initialize to low

do
  pause 100                                 ' pretend to be busy
  if KPflag = 1 then gosub getChar          ' go get new char
loop

' ==================== End Main Program - Subroutines Follow ====================
getChar:
  pulsout toKP, 2                           ' 10uS "send it" pulse
  pause 1                                   ' allow time for keypad
                                            ' to lower its output
  serin fromKP, N2400_8, char               ' get char
  sertxd (char, cr, lf)                     ' send it to terminal
  return
```

Download KeypadDriver.bas to your keypad peripheral and KeypadTest.bas to your master processor. Each time you press a key on the keypad, the appropriate character should appear in the terminal window. If not, try removing the power from the setup and then turning it back on, because both programs may need to start at the same time in order for the serial interface to function correctly. Of course, if that doesn't solve the problem, you will need to troubleshoot the system.

Now that we have completed our serial keypad project, we have a stand-alone peripheral device that enables the user to input data to any PICAXE

project, even one based on the 08M2. However, as I mentioned near the beginning of this chapter, we're going to take another look at keypads in Chapter 13. At that point, we'll use two of the advanced X2-class features to implement a keypad interface with the PICAXE-20X2 that entirely eliminates the need for a peripheral processor. In the meantime, we're going to turn our attention to the task of interfacing seven-segment LED displays with PICAXE processors.

This page intentionally left blank

CHAPTER 12

SPI Communication

In this chapter, we're going to use the Maxim/Dallas MAX7219 LED display driver to develop a stand-alone, four-digit LED display that can be used in any PICAXE project. Each of our previous projects has been serially interfaced with our master processor, but this time we're going to take a different approach, for three reasons. First, the February 2010 installment of the *Nuts and Volts* "PICAXE Primer" focused on the construction of a standard serial LED display based on the MAX7219, so there's no need to duplicate that project here. Second, the 7219 supports the serial peripheral interface (SPI), which is capable of operating at much higher speeds than a standard serial connection can achieve. Also, the X1 and X2 chips support a new "hardware" SPI interface that simplifies the software involved. As a result, an SPI-based LED display is a natural fit for our 20X2 master processor, and we're going to take full advantage of that fact.

The third reason for choosing an SPI interface is that it will give us the opportunity to see how easy it is to use an M2-class processor (the 08M2) to implement a function in software (SPI output) even though it doesn't contain the built-in hardware for doing so. Some of the material in the following discussion has been adapted from information originally published in the February 2010 "PICAXE Primer" column, and is presented here with the permission of *Nuts and Volts* magazine.

The MAX7219 8-Digit LED Display Driver

The datasheet for the MAX7219 is the primary source for the information contained in the following discussion. If you would like to download it for reference, it's available on my website and elsewhere. The 7219 is capable of interfacing a microprocessor with as many as eight common-cathode, seven-segment LED displays. It can also drive bar graph displays up to 64 bars or a maximum of 64 individual LEDs (see the datasheet for details). Several of the 7219's features will be especially helpful as we develop our four-digit LED display:

- The standard SPI is supported.
- Only three output pins are required for the interface.
- An internal Scan Limit register allows the user to display from one to eight digits.
- Digits can be updated individually, rather than having to update the entire display.
- Internal "binary-coded decimal" (BCD) decoding eliminates the need for software-based lookup tables.
- Only one external resistor is required to set the current for all LED segments.
- A simple digital brightness control can be adjusted in the software.

Figure 12-1 presents the pin-out of the MAX7219. As you can see, there are eight *segment* pins and eight *digit* pins; each of these 16 outputs connects directly to the corresponding pin of the LED display. If you are using the 7219 to drive fewer than eight LED digits (as we are), make sure you connect the *digit* outputs with the lowest numbers first, because the 7219 always scans the digits from digit 0 to the highest digit specified. In other words, the 7219 can automatically multiplex digits 0–3, but not digits 4–7.

A single resistor connected between +5V and the ISET pin controls the current that flows through each of the LED segments. Table 11 in the 7219's datasheet lists suggested resistor sizes for various segment currents. I chose to use a 47k resistor, which results in more-than-adequate brightness for the LED display that I am using. You can certainly use the data presented in the datasheet to adjust that value up or down, but don't go below the recommended minimum of 10k; doing so can cause excessive current flow that could damage or destroy the 7219.

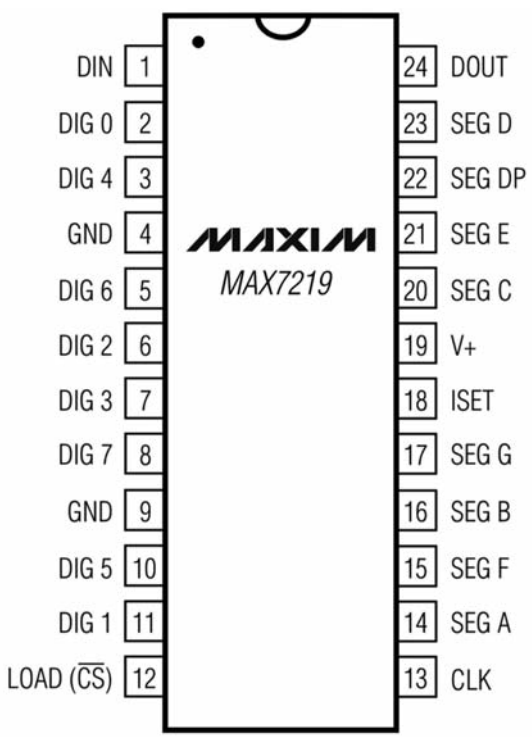


Figure 12-1 MAX7219 pin-out

The MAX7219's Three-Pin Interface

The 7219 uses a three-pin interface that is SPI-compatible. A 16-bit data word (more on that shortly) must be serially shifted into the chip via the *DIN* (data in) pin, under the control of the *CLK* (clock) pin. Each data bit that is placed on the DIN line is shifted into the 7219 on the rising edge of the CLK line. The most significant bit (MSB) of the data word should be sent first; the high byte of the word should contain the address of the 7219 internal register into which we want to place the data, and the low byte should contain the value of the data. The 7219 has a total of 14 internal digit and control registers. We will primarily be concerned with the four that are listed in Table 12-1 and the address registers for the LED digits themselves, which we'll soon discuss. When both bytes have been serially shifted into the 7219, its *LOAD* pin must be briefly pulsed in order for the data to be actually displayed on the LEDs.

Table 12-2 lists the 16 BCD characters that the 7219 can automatically decode. Conveniently, the BCD value for each of the ten digits is the same as the digit itself. If you want to light the decimal point to the right of any character, just add 128 to the corresponding BCD value (i.e., set bit7 to "1"). For example, to display a value of 5.2, you would send the following two values to the LED display: 133 (5 + 128) and 2.

The programming requirements for a seven-segment LED display usually include what's referred to as a "lookup" table, which serves to convert the value of a specific character to the corresponding pattern of segments that need to be lit to display the character. Since we're going to use BCD decoding in our project, we'll be able to avoid this programming chore. However, in case you're interested in being able to display characters other than those listed in Table 12-2, Figure 12-2 (Table 6 in the 7219 datasheet) presents the standard segment-labeling convention for a seven-segment LED, along with the 7219's

TABLE 12-1 Selected MAX7219 Internal Memory Registers

Register	Function
Decode Mode (Address 9) Values = 0–255	Enables BCD decoding for any combination of digits Example: 15 = %00001111 = decode first four digits from the right
Intensity (Address 10) Values = 0–15	Determines brightness of LED display (in addition to ISET resistor) Example: 7 or 8 = Average brightness
Scan Limit (Address 11) Values = 0–7	Determines the highest digit (from the right) that will be scanned Example: 3 = Scan digits 0 through 3 from the right
Shutdown (Address 12) Values = 0 or 1	0 = Display off 1 = Display on

TABLE 12-2 MAX7219 BCD Values

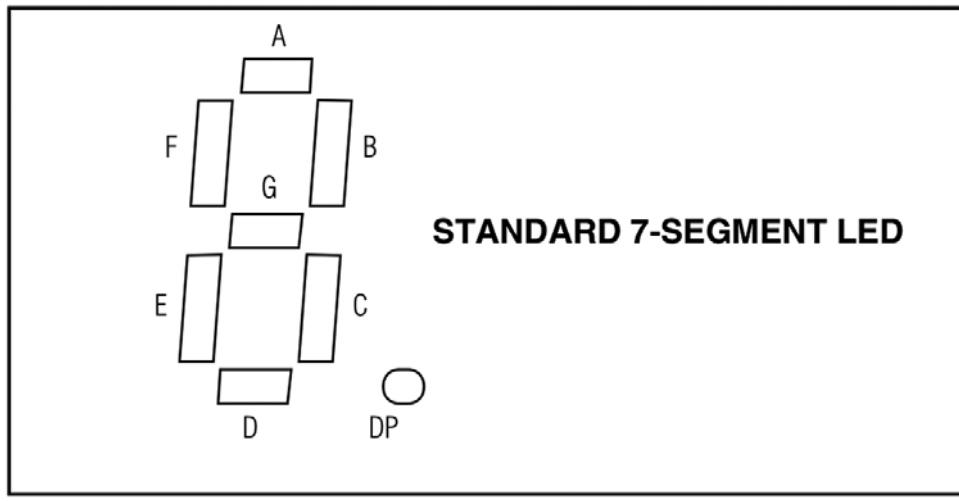
BCD Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Character	0	1	2	3	4	5	6	7	8	9	—	E	H	L	P	blank

mapping of the LED segments to the binary digits of a data byte. In order to clarify the data presented in Figure 12-2, let's assume that you have turned off the BCD decoding for one of the seven-segment LEDs and you want to display a capital F on it. To do so, you need to light segments A, E, F, and G, so the data value you need is %01000111, or decimal 71. If you intend to display several additional non-BCD characters, you will probably want to implement a software lookup table to retrieve the necessary values.

The 7219's decoding scheme allows us to individually enable or disable BCD decoding for each of the display's digits by setting or resetting the corresponding bits in the value stored in the Decode Mode register. In our current project, we're going to decode all four of our display's digits (i.e., digits 0 through 3), so we need to store %00001111 (decimal 15) in the Decode Mode register. If you decide to experiment with non-BCD characters, you will need to appropriately change the value you store in the Decode Mode register.

Before we actually begin construction of our project, let's take a brief look at the protocol for the transmission of the data that we want to display. As I mentioned earlier, the 7219 requires a 16-bit data word for each character that we want to display on one of the seven-segment LEDs in a display. The data word must consist of two bytes; the first (high) byte must be the address of the LED on which we want to display the character. Unfortunately, in the 7219 pin-out presented earlier in Figure 12-1, the eight possible LED digits are labeled from *DIG 0* to *DIG 7,* but the addresses that are used for the digits range from 1 to 8—just a potential little source of confusion to keep us on our toes!

The second byte that we need to send to the 7219 is the value of the character to be displayed. For our project, that's simply one of the BCD values presented earlier in Table 12-2. Since we will be displaying mostly numerical data, this byte is simply the value of the digit itself. For example, if we want to display "3" on digit 0 (the first LED



	REGISTER DATA							
	D7	D6	D5	D4	D3	D2	D1	D0
Corresponding Segment Line	DP	A	B	C	D	E	F	G

Figure 12-2 Standard 7-segment LED labeling and 7219 mapping of segments to binary values

from the right), we would need use the 20X2's *hspi* command (we'll discuss that later) to send the value "1" followed by the value "3." Once that's been done, we need to briefly pulse the 7219's LOAD pin to actually display the "3."

Project 12
Constructing an SPI 4-Digit LED Display

The schematic for our LED display project is presented in Figure 12-3. The circuit is extremely straightforward—all we are doing is making the necessary connections between the MAX7219 and our four-digit LED display and including the ISET resistor circuit and the 10uF electrolytic capacitor recommended in the 7219 datasheet. The 12 pins of the LED display are arranged in two rows of six pins each, which follow the standard IC pin-numbering convention. If you are looking down at the display, with the decimal points at the bottom right of each digit, pin 1 is the first pin on the left in the bottom row and the numbers run counterclockwise so that pin 12 is opposite pin 1.

Even though the circuit is simple, the breadboard wiring can be messy, so we're going to skip that part and construct our stripboard circuit right away. Actually, I'm going to present two different layout options and you can choose which one you prefer to construct. The first version includes a 5 by 2 straight male header that enables the stripboard to be connected to a breadboard by a ribbon cable in case you want to locate it some distance from your master processor circuit. The layout for this version of the project is presented in Figure 12-4.

As you can see, the 5 by 2 male header (H5) provides the same five connections that are available on H3 and H4. (See the schematic presented in Figure 12-3 for the specific ordering

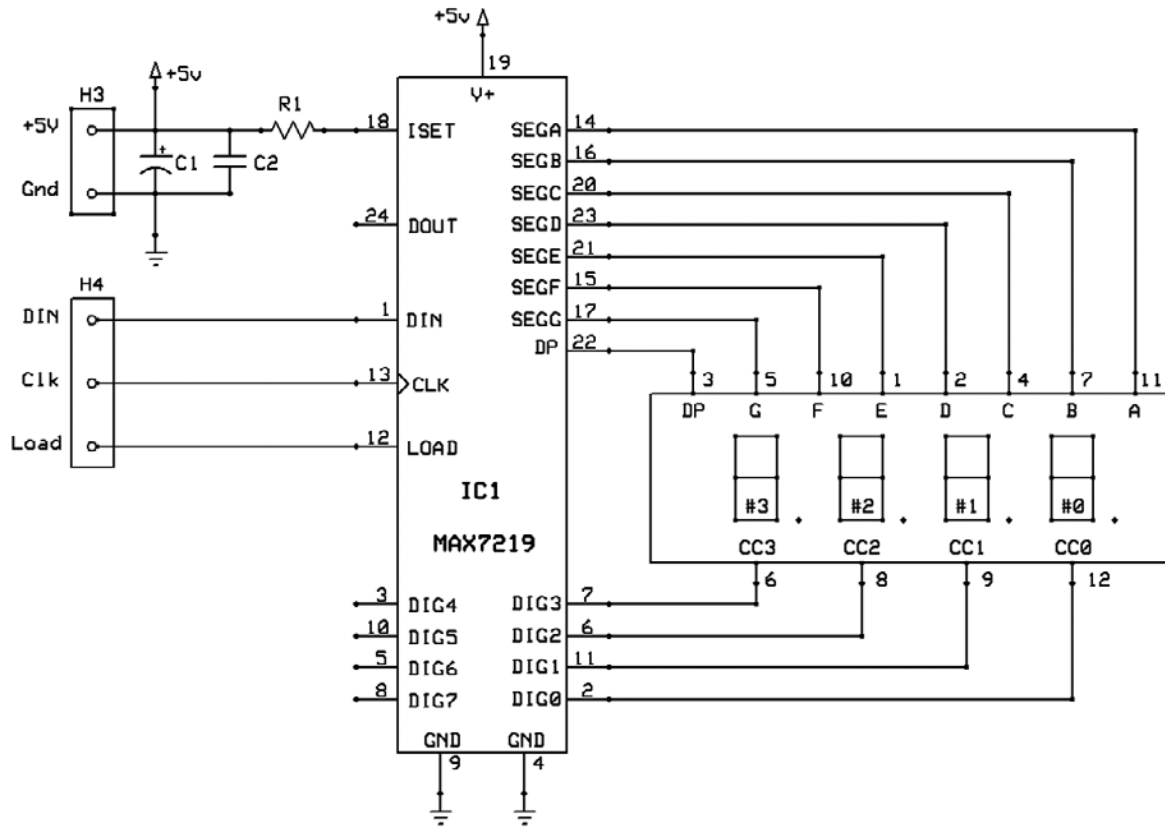


Figure 12-3 Schematic for the four-digit LED display

of the five connections.) Also, note that the trace in column A is solid; that's because I again used a small stripboard for the circuit, so I had to include one of the solid traces to total 20 traces. H1 and H2 are straight female headers into which you can insert the LED display. If you prefer, you could omit them and solder the LED display directly to the stripboard. (Of course, you would need to check your wiring carefully before doing so, because fixing any problems might require the removal of the LED from the board.) Finally, the three large black circles indicate where you can safely drill holes in the board for mounting. If you solder the LED display to the board, the top two holes would not be available unless you attached two small bolts before soldering the display in place.

The second version of the layout is presented in Figure 12-5. The only functional difference is that the 5 by 2 male header is not included because I knew that I only wanted to use the display in my breadboard circuits. However, I also tried to make the board as small as possible, which resulted in a slightly more complicated construction process. I'll explain the complications involved, and you can decide whether the smaller size of this version is worth the effort.

In Figure 12-5, you can see that row 8 and row 16 each contain two jumpers. This isn't that much of a complication; you just need to be sure to install them properly. In each row, first solder the short bare jumper in place and then cut and strip the ends of a longer insulated jumper and solder it in place. That way, there's no danger of a possible "short" on the board. The two insulated jumpers should be straight; I angled them in the layout for clarity. Also, at location K13 there are two different jumpers that need to be inserted into the same hole. In order to do so, you will need to use the thin leads from a .01µF capacitor or a 1/6-watt

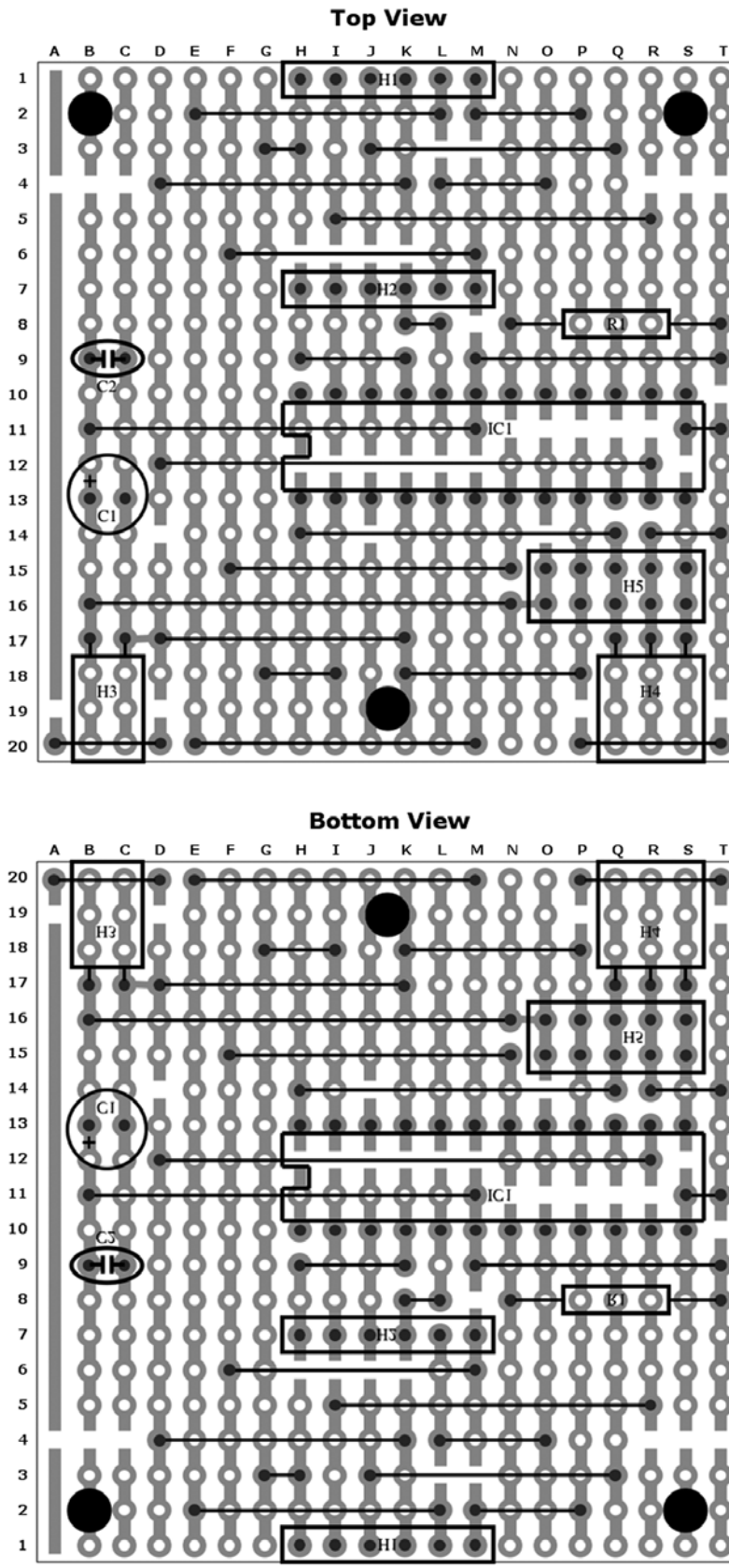


Figure 12-4 Stripboard layout for four-digit LED display (version 1)

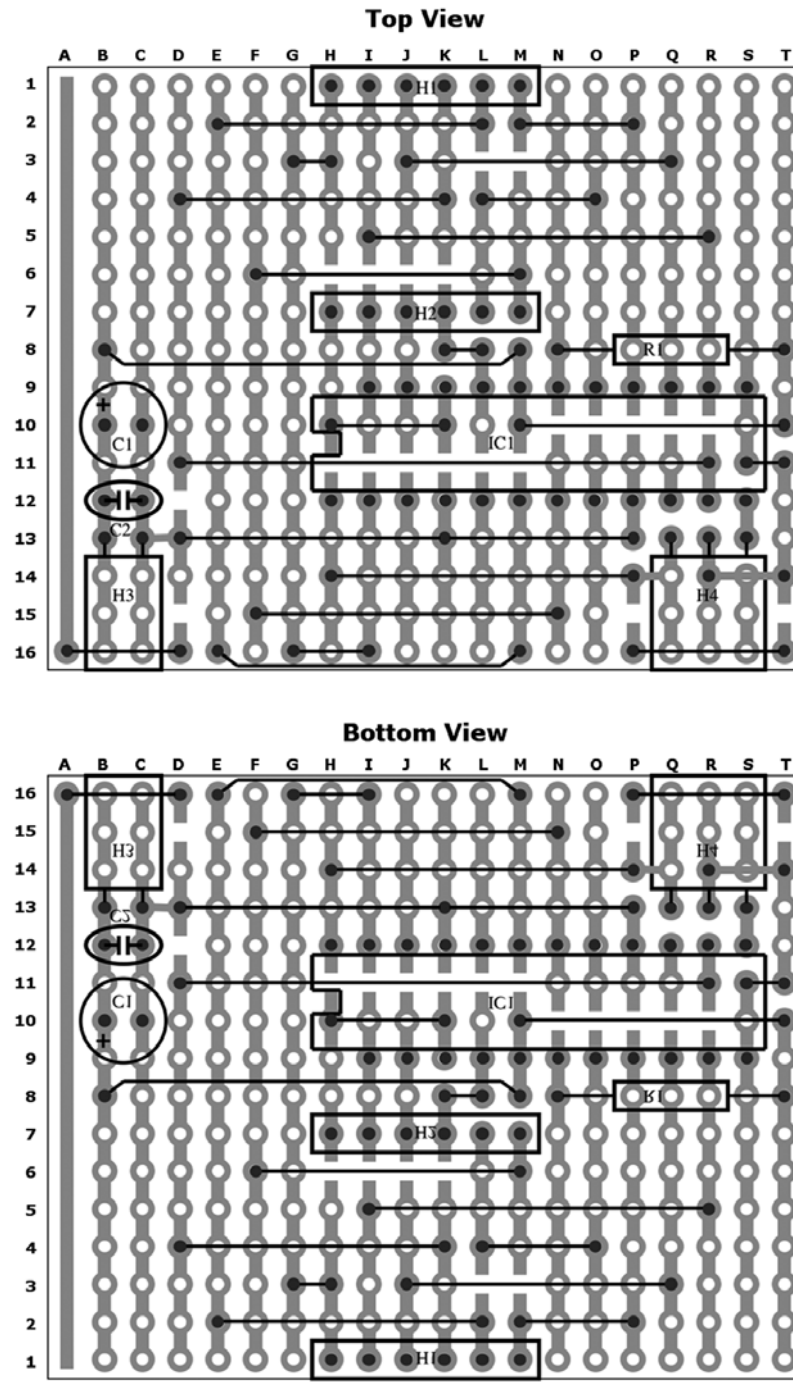


Figure 12-5 Stripboard layout for four-digit LED display (version 2)

resistor. Even with thin leads, you may need to widen the hole slightly—a 3/64-inch (1.2-mm) bit should be sufficient.

Finally, if you look carefully at the IC socket, you will see that there's no solder connection at pin 24 (hole H9). That's because I completely removed that pin from the socket before I soldered it in place. You could also just snip off the thin portion of the pin that gets inserted into the hole. Either way, what's important is that pin 24 of the 7219 is not electrically connected to any part of the circuit. The trace that contains hole H9 is used to connect pin 21 of the 7219 to pin 1 of the LED display; also, connecting pin 24 at H9 could damage or destroy the MAX7219.

CAUTION **If you choose to construct version 2 of the layout, make sure you remove or snip pin 24 of the IC socket before soldering it in place.**

Except for the 5 by 2 male header, the parts list is the same for both versions of the project. If you construct the version that includes the 5 by 2 male header, it should be reverse-mounted so that a ribbon cable can be inserted from the back of the board. Also, don't forget that you can make a three-pin, right-angle female header from the four-pin header, as we did in the previous chapter.

As I mentioned earlier I used a small stripboard for the layout, which is why the trace on the left is solid (i.e., no holes). If that's what you use, you will need to drill one hole in the solid trace at A16. If you would rather avoid that chore, you could cut the necessary stripboard from the larger one that's available on my site. The following set of assembly instructions is for version 2 of the project. If you decide to construct version 1 (which is simpler), just modify these instructions appropriately. As usual, read through the complete list of assembly instructions that follows to be sure you understand the entire procedure before beginning.

PARTS BIN

ID	Part
—	Stripboard, 20 traces by 16 (or 20) holes
C1	Capacitor, electrolytic, 10µF
C2	Capacitor, .01µF
R1	Resistor, 47k, 1/6 W
IC1	IC socket, 24 machined pins
—	MAX7219 LED display driver
H1, H2	Female headers, straight, 6 pins (see text)
H3	Female header, straight, 2 pins
H4	Female header, straight, 3 pins (see text)
H5	Male 5 by 2 straight header (only for v1)

1. Cut and sand a piece of stripboard to the required size (20 traces with 16 or 20 holes in all but one trace).
2. Using a 3/64-inch (1.2-mm) bit, drill a hole at A16 (solid trace) and enlarge the hole at K13.
3. Sever the traces on the bottom of the board as indicated in Figure 12-5.
4. Clean the bottom of the board with a plastic Scotch-Brite or similar abrasive pad.
5. Using the thin cutoff leads from a 0.01uF capacitor or 1/6W resistor, insert the jumper from D13 to K13 and the jumper from K13 to P13.
6. Flip the board over; bend the lead from D13 to C13 and snip it so that it just reaches the hole at C13.
7. Solder and snip the leads at K13 and P13, but not yet at C13 or D13.

8. Install the jumper between H14 and P14. On the bottom of the board, bend the lead from P14 to Q14 and snip it so that it just reaches Q14. Solder the lead at P14 and Q14; solder and snip the lead at H14.
9. Insert all the remaining jumpers *except* for the following: the one that runs from B8 to M8, the one that runs from E16 to M16, and the two that run over the top of H3 and H4. Solder and snip the leads.
10. Install an *insulated* jumper between B8 and M8. Solder and snip its leads.
11. Install an *insulated* jumper between E16 and M16. Solder and snip its leads.
12. Insert resistor R1. Solder and snip its leads.
13. Insert headers H3 and H4.
14. Flip the board over and support it on a flat surface. Make sure the lead from D13 touches the header pin at C13. Solder the five header pins and the lead at D13.
15. Insert the two jumpers that run over the tops of H3 and H4. Use a small spring clamp to hold each header and jumper tightly against the board, and solder and snip the leads.
16. Insert capacitor C. Solder and snip its leads.
17. Snip or remove pin 24 from the IC socket.
18. Insert the socket and solder all its remaining pins.
19. Insert capacitor C1. (The positive lead should be at B10.) Solder and snip its leads.
20. Insert headers H1 and H2 (or the LED display if you prefer to solder it directly to the board). Flip the board over, support it on a flat surface, and solder all the pins.
21. On the bottom of the board, place a short jumper that spans from R14 to T14. Using a small spring clamp to hold it in place, solder it at R14 and T14.
22. File or sand all the cut leads on the bottom of the board.
23. Clean the flux from the bottom of the board and allow it to dry.
24. Inspect the board carefully for accidental solder connections or other problems.

Testing the Completed 4-Digit LED Display

Figure 12-6 is a photo of the completed LED display (version 2) installed on my 20X2 master processor board and running the MAXhelp20X2.bas program that we are about to discuss. (There is a piece of red plastic on top of the display because it photographed better that way.) To connect the display to the 20X2, refer to the pin-outs for H3 and H4 that are shown in the schematic of Figure 12-3. (In the photo, you can't see the power connections to H3 because they are behind the board.) If you refer to the 20X2 pin-out presented back in Chapter 7 (Figure 7-1), you will see that pin 9 of the 20X2 is its *hspi sdo* pin (*sdo* stands for *serial data out*) and pin 11 is the *hspi sck* pin (*sck* stands for *serial clock*), so they need to be connected to the 7219's *DIN* pin and *CLK* pin, respectively. We can use any output we want to connect to the 7219's *LOAD* pin—I used C.2 (pin 8).

To test the completed display board, we're going to run the MAXhelp20X2.bas program in Listing 12-1. It takes advantage of the 7219's BCD decoding to display the word "HELP" on the LEDs. The program is thoroughly commented, but the *hspisetup* command does require a brief explanation. The complete syntax for the command is *hspisetup mode*, *spispeed*. If you read the *hspisetup* documentation in Part II of the manual, you'll see that several values can be used for the mode parameter. However, they all refer to different options for the reception of SPI data. Since the 7219 doesn't send any data back to the master processor, it doesn't matter which value we use, so I just chose the first one (*spimode00*). The *spispeed* parameter accepts one of three different

LISTING 12-1

```
' ================================ MAXhelp20X2.bas ================================
' This program runs on a PICAXE-20X2 & controls a MAX7219 LED display
' driver. The MAX7219 is connected to a 4-digit, 7-segment LED display.
' The program displays the word "HELP"

' === Constants ===
' Hardware interface to the MAX7219
  symbol load = C.2               ' briefly pulse C.2 to transfer data to LEDs

' Register addresses for the MAX7219
  symbol decode =  9              ' decode register; specify digits to decode
  symbol brite  = 10              ' intensity (brightness) register; 15 = 100%
  symbol scan   = 11              ' scan-limit register; specify how many digits
  symbol on_off = 12              ' 1 = display on; 0 = display off

' === Directives ===
  #com 3                          ' specify the com port
  #picaxe 20X2                    ' specify the PICAXE processor
  #no_data                        ' speed up the download
  #no_table                       ' speed up the download
  #terminal off                   ' disable terminal window

' ============================== Begin Main Program ==============================
  setfreq m64
  dirsb = %11111111               ' set portB as all outputs
  dirsc = %10111111               ' set portC as outputs (except C.6)
  hspisetup spimode00,spifast     ' set up hspi

' Initialize MAX7219
  hspiout (scan,3)                ' set scan limit for digits 0-3
  pulsout load,1
  hspiout (brite,5)               ' set brightness to 5 (15 = 100%)
  pulsout load,1
  hspiout (decode,15)             ' set BCD decoding for digits 0-3
  pulsout load,1
  hspiout (on_off,1)              ' turn display on
  pulsout load,1

' Send data to the four digits
  hspiout (1,12)                  ' 1st LED from left = "H"
  pulsout load,1
  hspiout (2,11)                  ' 2nd LED from left = "E"
  pulsout load,1
  hspiout (3,13)                  ' 3rd LED from left = "L"
  pulsout load,1
  hspiout (4,14)                  ' 4th LED from left = "P"
  pulsout load,1
```

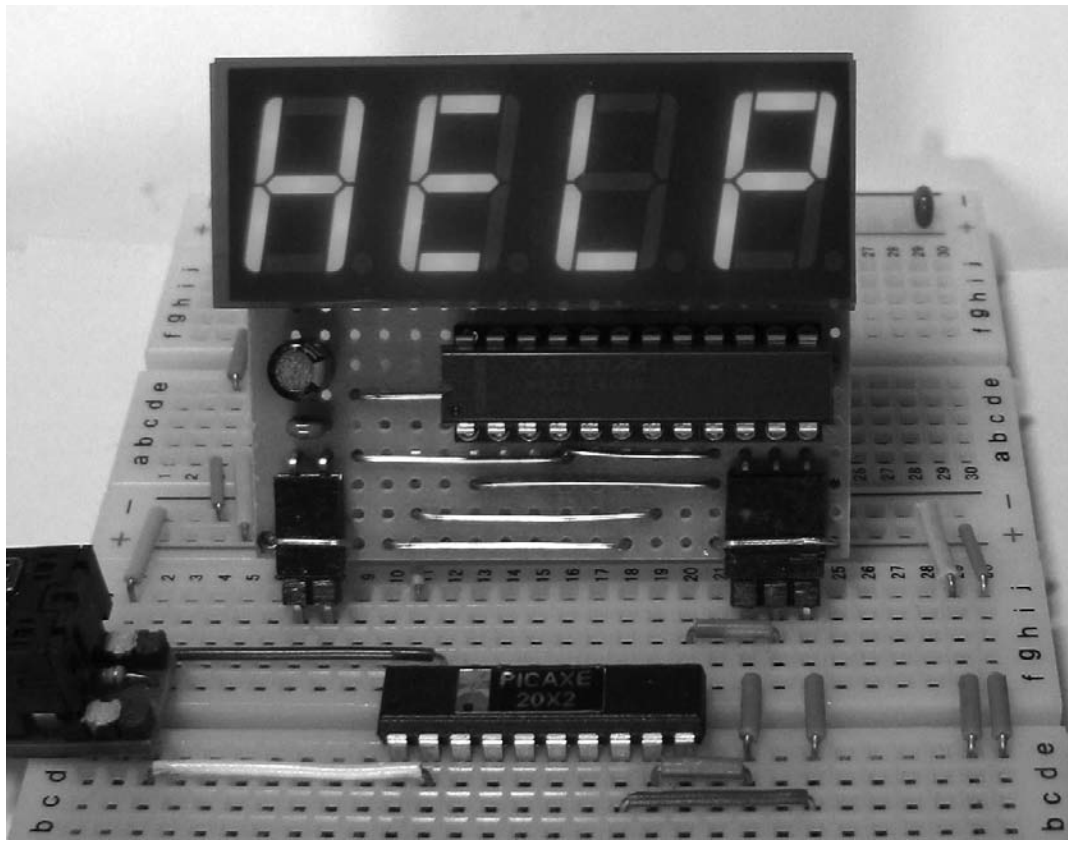


Figure 12-6 Completed LED display (version 2)

values. Since we're running the 20X2 at 64MHz, I initially thought that the *spifast* value would be too fast, but it works fine. Download MAXhelp20X2.bas to your master processor—you should see the word "HELP" immediately appear on the display. If it doesn't, you're in for a little troubleshooting session.

Interfacing the LED Display with an M2 Processor

When your MAXhelp20X2.bas program is running correctly, we're ready to take a look at the program modifications that are required to interface our LED display with any M2-class processor. As you already know, the main consideration is that the M2 processors do not support the hardware SPI interface. Therefore, we need to take a software approach to shifting the 16-bit data out to the display. In order to make the necessary software as simple as possible, we're going to break a rule, so this is going to be fun. I have mentioned more than once that whenever you declare a word variable, you should avoid using either of the associated two byte variables in the same program. For example, if your program uses the w0 variable, you should not also use b0 or b1 because it can result in erratic program behavior. However, in this case, breaking that rule actually simplifies our program. Take a look at the variable declarations in the MAXhelp.bas program shown in Listing 12-2, and then I'll explain what I mean.

As you can see, *outByte* (the eight-bit data byte to be sent to the display) is declared as variable b0 and *maxReg* (the 7219 register that is going to receive outByte) is declared as variable b1. However, the very next statement declares *forMax* as the w0 variable. Before you send for the programming police, let me explain. That combination of variable declarations makes *forMax*

LISTING 12-2

```
' ================================= MAXhelp.bas ================================
' This program uses a PICAXE-08M2 to manually shift-out SPI data to
' a MAX7219 LED display driver. The 7219 is connected to a 4-digit,
' 7-segment LED display. The program displays the word "HELP"

' === Constants ===
' Hardware interface to the MAX7219
  symbol  sData  =  0             ' data out line to Max7219
  symbol  clock  =  1             ' clock line
  symbol  sLoad  =  2             ' pulse briefly to load data onto LEDs

' Register addresses for the MAX7219
  symbol  decode =  9             ' decode register; specify digits to decode
  symbol  brite  = 10             ' intensity (brightness) register; 15 = 100%
  symbol  scan   = 11             ' scan-limit register; specify # of LEDs
  symbol  on_off = 12             ' 1 = display on; 0 = display off

' === Variables =====
  symbol outByte = b0             ' data byte to be transmitted to the LED
  symbol maxReg  = b1             ' MAX register that receives the data byte
  symbol forMax  = w0             ' We're breaking a rule here! (See text)
  symbol index   = b2             ' used in for-next loop

' === Directives ====
  #com 3                          ' specify com port
  #picaxe 08M2                    ' specify PICAXE processor
  #terminal off                   ' disable terminal window

' ============================== Begin Main Program ============================
  setfreq m8
  dirsC = %00010111

' Initialize MAX7219
  maxReg  = scan                  ' set scan limit for digits 0-3
  outByte =  3
  gosub shout16

  maxReg  = brite                 ' set brightness to 5 (15 = 100%)
  outByte =  5
  gosub shout16

  maxReg  = decode                ' set BCD decoding for digits 0-3
  outByte = 15
  gosub shout16

  maxReg  = on_off                ' turn display on
  outByte =  1
  gosub shout16

' Send data to the four digits
  maxReg  =  1                    ' 1st LED from left = "H"
```

LISTING 12-2 (continued)

```
    outByte = 12
    gosub shout16

    maxReg  =  2                    ' 2nd LED from left = "E"
    outByte = 11
    gosub shout16

    maxReg  =  3                    ' 3rd LED from left = "L"
    outByte = 13
    gosub shout16

    maxReg  =  4                    ' 4th LED from left = "P"
    outByte = 14
    gosub shout16

' ==================== End Main Program - Subroutines follow ====================
    shout16:
    for index = 15 to 0 step -1     ' MAX7219 requires a 16-bit word
      if bit15 = 1 then             ' set sdata to correspond to bit15
         high sData
      else
         low  sData
      endif
      pulsout clock, 2              ' briefly pulse the clock line
      forMax = forMax * 2           ' shift char left for next MSB
    next index

    pulsout sLoad, 2                ' briefly pulse the load line
    return
```

a 16-bit word variable with its high byte equal to *maxReg* and its low byte equal to *outByte,* which is exactly the data format that the 7219 expects to receive. If you look back at any one of the *hspiout* statements in the MAXhelp20X2.bas program, you will see what I mean; for example, consider the very first one: *hspiout (scan, 3)*. That statement causes the 20X2 SPI hardware to serially shift out (MSB first) the eight-bit value associated with the scan register (which is %00001011, or decimal 11), followed by the eight-bit value of 3 (which is %00000011).

In the 08M2 version of the software, the *shout16* subroutine accomplishes in software what the 20X2 can do in hardware; it serially shifts out the 16-bit data word (MSB first) in a single *for/next* loop. All we have to do is be sure that *maxReg* (i.e., b0) and *outByte* (i.e., b1) contain the correct values before we call the *shout16* subroutine. We could have done exactly the same thing without declaring *forMax*, but that would have required two separate for/next loops in the *shout16* subroutine. Using *forMax* enables us to simplify the software and we get to break a rule as well—what more could you ask for?

Figure 12-7 is a photo of my LED display installed on a breadboard and interfaced with an 08M2 processor. To simplify the interface as much as possible, I placed the display's three I/O connections directly in line with the 08M2's C.0, C.1, and C.2 output pins as follows: DIN = C.0, CLK = C.1, and LOAD = C.2. In the photo, you

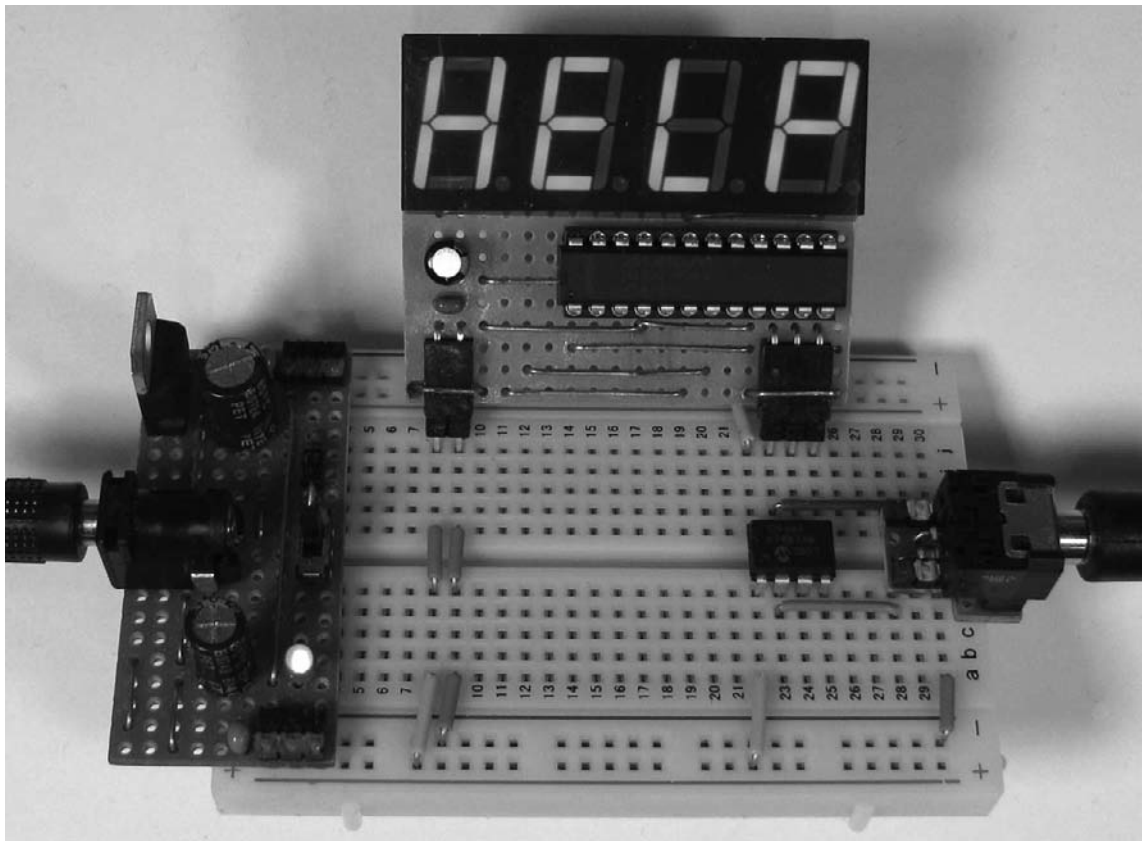


Figure 12-7 My LED display installed on a breadboard and interfaced with an 08M2 procesor

can also see the necessary power and ground connection to the display and the USBS-PA3X2 programming adapter connections. Pin 1 of the 08M2 is at its lower-left corner, so the USBS-PA3X2 is connected from behind the 08M2. When you have completed your breadboard setup, download MAXhelp08M2.bas to the 08M2; "HELP" should again be displayed. If not, you will need to check your breadboard wiring.

Learning to Count

When your MAXhelp08M2.bas program is running correctly, we're ready to try a second program on the display. This program (MAXcount20X2.bas) simply counts from 0 to 9999 in an infinite loop (see Listing 12-3). My main purpose for including it is to demonstrate how fast the *hspi* interface is on the PICAXE-20X2. The program is fairly simple, but two aspects of it do require a brief explanation. First, at the beginning of the main counting loop, we're taking advantage of the 20X2's *dig* command to simplify the task of isolating the individual digits of our counter variable (*cntr*). The *dig* command, which is short for "digit," returns the value of the specified digit of a 16-bit number. The specification of which digit to return follows the same pattern that we have been using with our LED display. In other words, "0" refers to the first digit from the right (the "ones" digit), "1" refers to the second digit from the right (the "tens" digit), etc. If you look for *dig* in Part II of the manual, it's in the section titled "Variables – Mathematics."

Second, the "zero-blanking" aspect of the program may also require an explanation. It would certainly be simpler to always display a four-digit number, no matter how small it is. In other words,

LISTING 12-3

```
' ============================== MAXcount20X2.bas ==============================
' Program runs on a PICAXE-20X2 & controls a MAX7219 display driver.
' The MAX7219 is connected to a 4-digit, 7-segment LED display.
' The program counts from 0 to 9999 in an infinite loop.

' === Constants ===
  symbol  blank = 15                ' used by MAX7219 to blank a digit

' Hardware interface to the MAX7219
  symbol   load = C.2               ' briefly pulse C.2 to transfer data to LEDs

' Register addresses for the MAX7219
  symbol decode =  9                ' decode register; specify digits to decode
  symbol brite  = 10                ' intensity (brightness) register; 15 = 100%
  symbol scan   = 11                ' scan-limit register; specify num of digits
  symbol on_off = 12                ' 1 = display on; 0 = display off

' === Variables ===
  symbol   cntr = w0                ' used to count from 0 to 9999
  symbol   ones = b2                ' used to access the ones digit of cntr
  symbol   tens = b3                ' used to access the tens digit of cntr
  symbol   hnds = b4                ' use to access the hundreds digit of cntr
  symbol   thos = b5                ' used to access the thousands digit of cntr
  symbol   val  = b6                ' data to be transmitted to the MAX7219

' === Directives ===
  #com 3                            ' specify the com port
  #picaxe 20X2                      ' specify the PICAXE processor
  #no_data                          ' speed up the download
  #no_table                         ' speed up the download
  #terminal off                     ' disable terminal window

' ============================ Begin Main Program ==============================
  setfreq m64
  dirsb = %11111111
  dirsc = %10111111
  hspisetup spimode00,spifast       ' set up hspi

' Initialize MAX7219
  hspiout (scan,3)                  ' set scan limit for digits 0-3
  pulsout load,1
  hspiout (brite,5)                 ' set brightness to 5 (15 = 100%)
  pulsout load,1
  hspiout (decode,15)               ' set BCD decoding for digits 0-3
  pulsout load,1
  hspiout (on_off,1)                ' turn display on
  pulsout load,1
```

(continued)

LISTING 12-3 (continued)

```
do
  for cntr = 0 to 9999              ' main counting loop
    ones = cntr dig 0               ' 1st digit from right
    tens = cntr dig 1               ' 2nd digit from right
    hnds = cntr dig 2               ' 3rd digit from right
    thos = cntr dig 3               ' 4th digit from right
    if thos = 0 then                ' thousands digit (1st from left)
      val = blank
    else
      val = thos
    endif
    hspiout (1,val)
    pulsout load,1
    if hnds=0 AND cntr<100 then     ' hundreds digit (2nd from left)
      val = blank
    else
      val = hnds
    endif
    hspiout (2,val)
    pulsout load,1
    if tens=0 AND cntr<10 then      ' tens digit (3rd from left)
      val = blank
    else
      val = tens
    endif
    hspiout (3,val)
    pulsout load,1
    hspiout (4,ones)                ' ones digit (4th from left)
    pulsout load,1
    pause 500                       ' slow down the count a bit
  next cntr
loop
```

we could easily display "0003" instead of "3" but the latter version is the way the number "3" should be displayed. The *if/then/else* statements in the program determine whether a "0" or a blank space should be displayed, depending on the number of digits in the number that is to be displayed. Download MAXcount20X2.bas to your master processor and have fun watching the display count from 0 to 9999. If you want to see just how fast the *hspi* command can run on the 20X2, comment out (or remove) the *pause 500* statement at the end of the counting loop—it's very impressive!

A Programming Challenge

Of course, the 08M2 can count just like the 20X2, but I'm going to leave that one to you as a programming challenge. However, I will give you a hint. The M2 processors don't support the *dig* command, so you'll need to find another way to individually access the four digits of the *cntr* variable. Here's the hint: Use the *bintoascii* command in combination with the fact that for the ten digits, each ASCII value is equal to the value of the digit plus 48. So, once you get the four

ASCII values (by using *bintoascii*), just subtract 48 from each one to get the value of the individual digit. Have fun!

Once you have risen to the challenge, you may also want to experiment with turning off the "BCD decode" mode for one or more of the LEDs and displaying additional characters. For example, you can display the word "yES" by using a lowercase "y," an uppercase "E," and an uppercase "S." To figure out the value that you need for each of these letters, refer back to the information presented earlier in Figure 12-2. Many different uppercase and lowercase letters can be displayed this way. If you want to use several different letters, a software lookup table would be the simplest way to access the required values to send to the MAX7219.

In the next chapter, we're going to explore the 20X2's advanced timing capabilities. As we'll see, our LED display will come in handy for this purpose.

This page intentionally left blank

CHAPTER 13

Background Timing on the 20X2 Processor

Each of the projects we have constructed thus far in Part Two has consisted of a single peripheral device. However, for the remainder of Part Two we're going to take a different approach and implement projects that will integrate the use of two different peripherals. In this chapter, we're going to use a modified keypad and our four-digit LED display to construct a simple countdown timer. In Chapter 14 we'll extend the concept further and construct the multifunction peripheral device (MPD), a fully programmable stand-alone peripheral device that will be able to carry out a variety of useful functions. In Chapter 15 we'll develop a major application for the MPD.

However, before we get to our timer project, we need to discuss some of the details of the 20X2's background timing capabilities. (You have to eat your broccoli before you get the ice cream!) All X2-class processors include two hardware timer modules that can operate in the background. Timer1 is a 16-bit timer/counter that can be automatically preloaded with a specific value; it then repetitively counts up (and overflows) from that value. Timer3 is also a 16-bit timer, but it is "free running," which means that it can't be preset—it just repetitively counts up (and overflows) in the background. Being able to automatically preload Timer1 gives us a considerable amount of control over the actual timing interval that occurs in the background. Therefore, Timer1 is the better choice for our countdown timer project later in this chapter, so let's take a detailed look at how it functions.

Using Timer1 on the 20X2 Processor

On the 20X2 processor, Timer1 simply operates as an internal timer. (An external counting mode is also available on the 28X2 and 40X2 processors—see the manual for details.) The timer operates with what are called "minor ticks" and "major ticks." A minor tick occurs every (256 / clock frequency) seconds; therefore, at 16MHz, a minor tick occurs every 16µS (256 / 16,000,000). An internal word variable automatically increments on each minor tick. The minor tick variable is not accessible from a running program, but each time it overflows from 65535 to 0, a major tick occurs and a second internal word variable (*timer*) automatically increments. The *timer* variable can be accessed in a program to determine the number of major ticks that have occurred.

Timer1 is configured with the *settimer* command. For our purposes, there are two variations of this command that we need to understand: *settimer preload* and *settimer off*. The first variation starts the timer running, and the *preload* parameter specifies the value at which the minor ticks begin counting up; the second variation simply turns off the background timing. By carefully choosing the *preload* value, we can

control the rate at which the *timer* variable increments. For example, suppose we want the *timer* variable to increment at the rate of once per second, which is exactly what we will do in the countdown timer project. We have already figured out that a minor tick occurs every 16µS when the clock frequency is 16MHz. Since one second equals 1,000,000µS, we need (1,000,000µS / 16µS), or 62,500, minor ticks to produce an exact one-second delay. Therefore, because the timer is counting up and it rolls over from 65,535 to 0, we need to preload the timer with 3036, which is 65,536 – 62,500. In sum, if we issue a *settimer* 3036 command in a 20X2 program running at 16MHz, the built-in *timer* variable will increment exactly once per second.

Actually, you can use Timer1 without going through all these calculations because the PICAXE compiler includes three predefined timing constants that produce a one-second major tick at three different clock speeds, as shown in Table 13-1. (This approach is almost as easy as using the M2-class *time* variable, which is preconfigured to count exact seconds.) However, it's a good idea to have some understanding of how these values are determined. One reason is that the clock speed of processors that have internal resonators (such as the 20X2 and all M2-class processors) is rarely as precise as that of processors that have external resonators (such as the 28X2 and 40X2). Therefore, if the timing requirements of a project are somewhat critical, you may need to adjust the preload value slightly to achieve the precise timing that is required. When we get to our countdown timer project, we'll talk more about this issue.

TABLE 13-1 Timer1 Preload Constants that Produce One-Second Major Ticks

Constant	Value	Clock Speed
t1s_4	49910	4MHz
t1s_8	34286	8MHz
t1s_16	3036	16MHz

As I have already mentioned, once you have properly configured the *settimer* command, your program can simply access the *timer* variable to determine the elapsed time. However, there is a second, more powerful approach that involves interrupts and the *setintflags* command. All PICAXE X1 and X2 processors include a system *flags* byte that contains eight flags that can be used to trigger an interrupt in various situations. The one that relates to background timing is the "timer overflow flag" (*toflag*), which occupies bit 7 in the *flags* byte.

The basic syntax for the *setintflags* command is *setintflags flags, mask.* (There's also a *setintflags off* option to discontinue the interrupts when needed.) The *flags* parameter specifies which condition(s) will trigger an interrupt. Since the *toflag* is bit 7 of the *flags* byte, setting the *flags* parameter to %10000000 will cause an interrupt to be triggered whenever the *timer* variable overflows. As usual, a "0" in any position of the *mask* parameter is used to ignore the corresponding bit, so we would also want to set the *mask* parameter to %10000000. Thus, the complete command to enable an interrupt whenever the *timer* variable overflows is *setintflags %10000000, %10000000.*

Once the proper *setintflags* command has been issued, all we need is an *interrupt* subroutine to implement the necessary tasks whenever the timer overflows. If we use the specific commands we have been discussing, our interrupt routine will be called exactly once per second, so all it needs to do is update the relevant timing variables (*seconds, minutes, etc.*). We will discuss the details when we get to our countdown timer project. But before we do that, we need to do a little "surgery" on our matrix keypad.

"Deconstructing" a Matrix Keypad

I have always enjoyed taking things apart to see how they work, so as soon as I finished writing Chapter 11, I decided to disassemble the keypad we used to see how it's constructed. It turned out to be a surprisingly easy process, as we'll soon see. Figure 13-1 is a photo of the backs of three different keypads. The keypad in the middle is the one we used in Chapter 11; the other two are included for comparison. I have several different keypads, and they are all constructed in a similar manner. There are several round plastic posts on which a PC board is mounted. Each post has been melted a little to expand it in order to hold the PC board in place—in effect, the posts have become little plastic rivets.

To disassemble a keypad, simply use a pair of flush-cutting pliers to snip off the head of each plastic rivet. You will also need to use a sharp hobby knife to shave off any remaining excess plastic on the tops of the rivets. Next, use a small flat-bladed screwdriver to pry off the PC board. (Be sure to mark or remember the correct orientation of the board so that you can easily reassemble it later.) Inside, you will find a rubber sheet that contains 16 conductive "buttons" that make the necessary contact when a key is pressed. Figure 13-2 is a photo of the keypad we used in Chapter 11 with the rubber sheet removed. (Again, make sure to mark or remember the correct orientation of the rubber sheet so that you can easily reassemble the keypad.) In Figure 13-2, you can also see that four of the keys on my keypad were formed slightly differently. As far as I can tell, it has nothing to do with their functionality; those keys probably just came from two different manufacturing batches.

Once I disassembled my keypad, I made two major changes that involve rearranging the keys. First, since the project in this chapter will be simpler to wire if the connector is on the top edge of the keypad and the project in the next chapter actually requires that arrangement, I decided to rotate each key 180 degrees. (It might seem simpler to just rotate the PC board, but I discovered that the plastic pegs didn't line up exactly with the holes when I tried that, so I rotated each individual key instead.) Originally, I thought I might have to glue the PC board back in place when I finished the "surgery," but it snapped back onto the plastic pegs and, at least so far, it seems to be solidly affixed to the keypad. If that

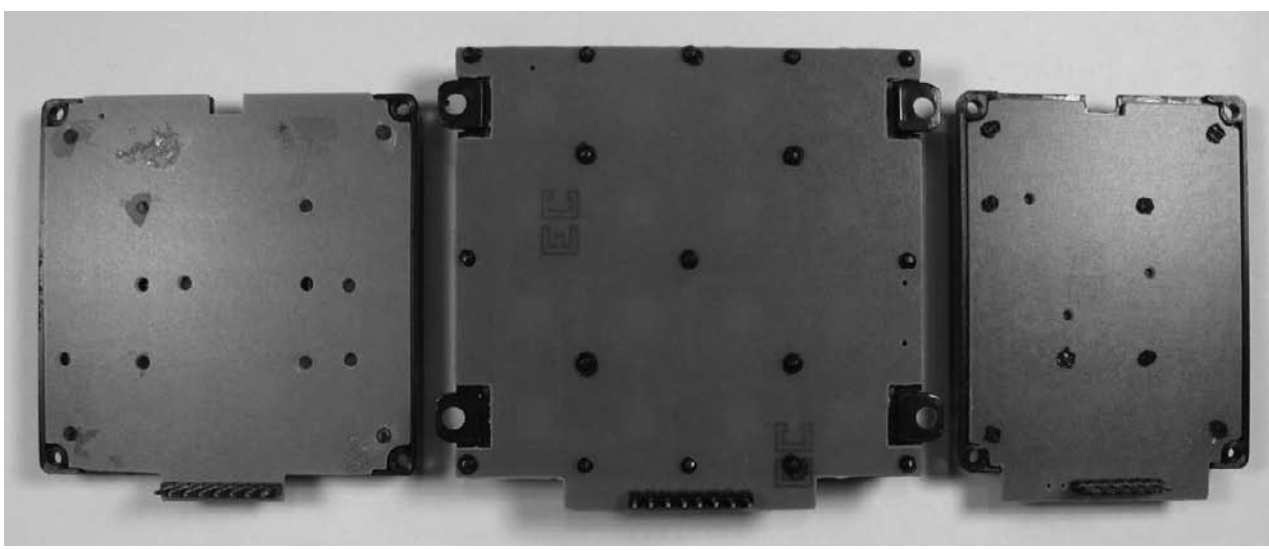


Figure 13-1 Bottom view of three matrix keypads

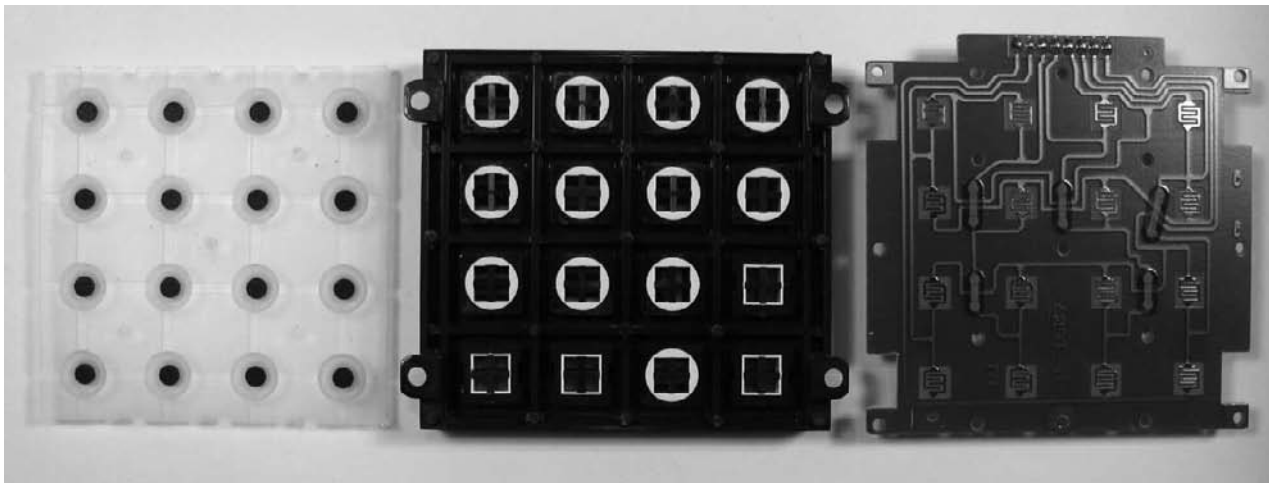


Figure 13-2 Disassembled matrix keypad

doesn't happen for you, a small dot of silicone adhesive on three or four plastic pegs should do the trick.

In order to make the new keyboard as useful as possible, I also decided to include a typical four-key cursor arrangement on the keypad. The final configuration is shown in the photo in Figure 13-3. The keypad on the left has had all its keys rearranged and rotated 180 degrees so the connector is now on the top, and I have added four "arrow" symbols that I printed from a Brother PT-300 label maker to identify my cursor keys. I used black tape with white letters so that I could completely cover the original characters on the keys. Also, I deliberately retained the "B" key as a "Back" key in a menu structure and the "A" key as an "Accept" or "Enter" key. The keypad on the right shows the same rearrangement of the keys, but I didn't rotate anything so the connector is still on the bottom of the keypad. That way, I can still use that keypad with the stripboard project we developed in Chapter 11. After I took the photo in

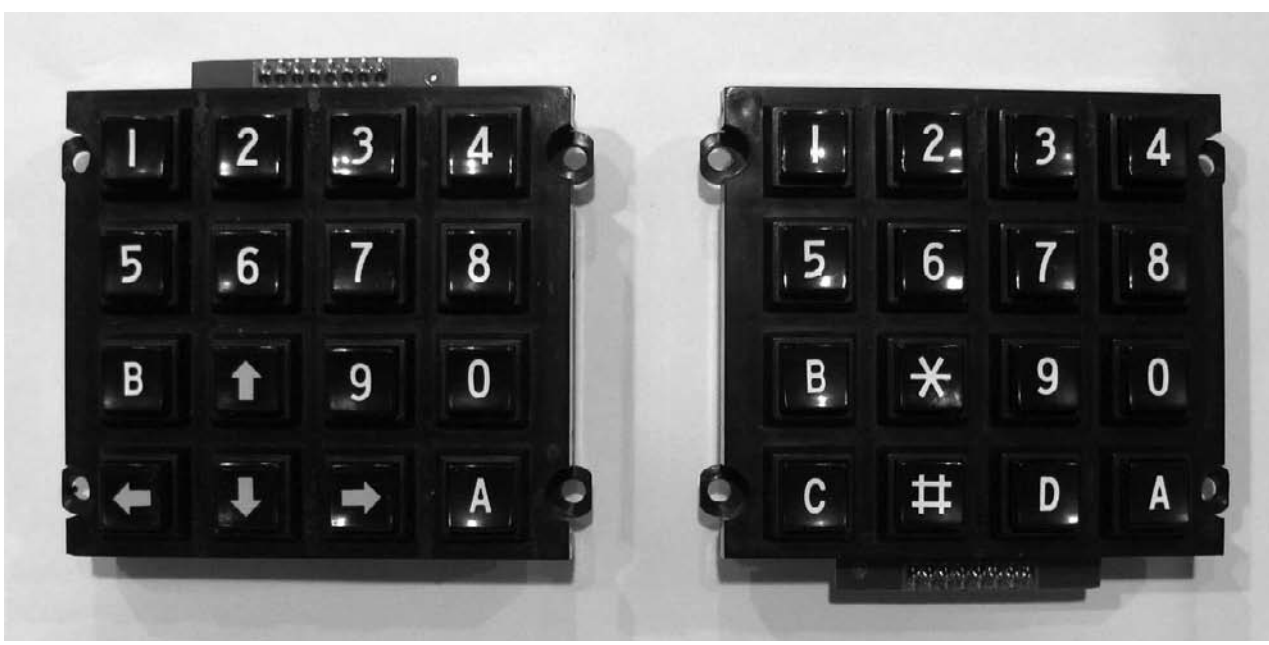


Figure 13-3 Reassembled matrix keypad

Figure 13-3, I added the same four arrow symbols to the keypad on the right as well.

As you can see in Figure 13-3, the new key layout results in a slightly unusual arrangement for the ten digit keys. I think this compromise is justified by the convenient layout for the four arrow keys and the fact that the "Enter" key is in the lower-right corner where it should be. However, if you prefer a different arrangement, now is the time to implement it. When we discuss the necessary changes in the schematic and the software, it would be a simple matter to make additional adjustments for any key arrangement you choose.

Testing the "New and Improved" Keypad

Figure 13-4 presents the schematic for a simple test circuit for our rearranged keypad. If you look back at the schematic in Figure 11-3 that we used with our original keypad, you will see that the new schematic is essentially just a 180-degree rotation of the original one, which makes perfect sense. The only other difference is that this time we're using the 20X2 processor rather than the 08M2.

Figure 13-5 is a photo of my breadboard setup for the test circuit. If you compare Figure 13-5 to the breadboard layout we used with our original keypad (Figure 11-4), you will also see the 180-degree rotation in the layout.

When you have assembled your breadboard circuit, we're ready to determine whether our keypad surgery was successful. The program I used for this purpose (KeypadNew.bas) is presented in Listing 13-1. It is functionally identical to the Keypad2.bas program we discussed in Chapter 11, so I won't go into the details, except to point out the rearrangement of the characters. If you look at the various *case* statements in the *select case* command, you will see that the ADC values are identical to those that we used in Chapter 11. However, the *char* assignments are completely rearranged. It all makes more sense if you look at it "from the bottom up." In other

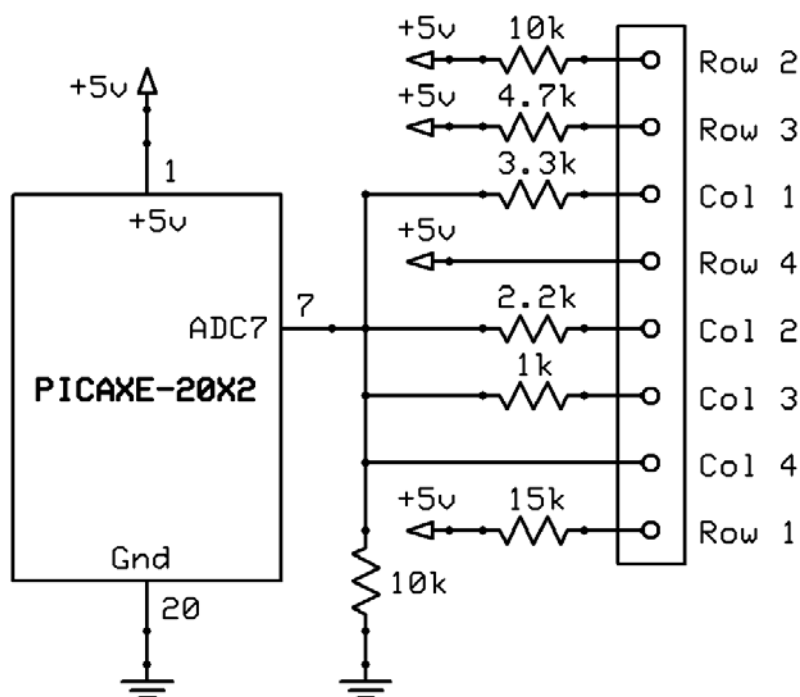


Figure 13-4 Schematic for the rotated keypad circuit

LISTING 13-1

```
' ================================ KeypadNew.bas ================================
'   This program uses an ADC approach to decoding a matrix
'   keypad that has had all its keys rotated and rearranged.
'   It sends the decoded character to the terminal window.

' === Variables ===
  symbol key  = w0                  ' readadc10 requires a word variable
  symbol char = b2                  ' char that corresponds to each key value
  symbol junk = b3                  ' throwaway variable used for debouncing

' === Directives ===
  #com 3                            ' specify serial port
  #picaxe 20X2                      ' specify processor
  #no_data                          ' speed up download
  #no_table                         ' speed up download
  #terminal 19200                   ' open terminal at 19200 baud (for 16MHz)

' ============================== Begin Main Program ==============================
setfreq m16                         ' set clock speed

do
wait_for_keypress:
  readadc 7, junk
  if junk < 5 then wait_for_keypress
  pause 50                          ' debounce keypress
  readadc10 7, key                  ' get ADC value

wait_for_release:
  readadc 7, junk
  if junk > 5 then wait_for_release

  select case key                   ' decode keypress
  ' (Characters are simply rearranged from Ch.11 Program.)
    case < 369 : char = 65          ' A
    case < 385 : char = 82          ' R
    case < 402 : char = 68          ' D
    case < 434 : char = 76          ' L
    case < 450 : char = 48          ' 0
    case < 474 : char = 57          ' 9
    case < 500 : char = 85          ' U
    case < 551 : char = 66          ' B
    case < 587 : char = 56          ' 8
    case < 634 : char = 55          ' 7
    case < 674 : char = 54          ' 6
    case < 733 : char = 53          ' 5
    case < 804 : char = 52          ' 4
    case < 885 : char = 51          ' 3
    case < 977 : char = 50          ' 2
    else       : char = 49          ' 1
  end select

  sertxd ("char = ", char, cr, lf)  ' send char to terminal
loop
```

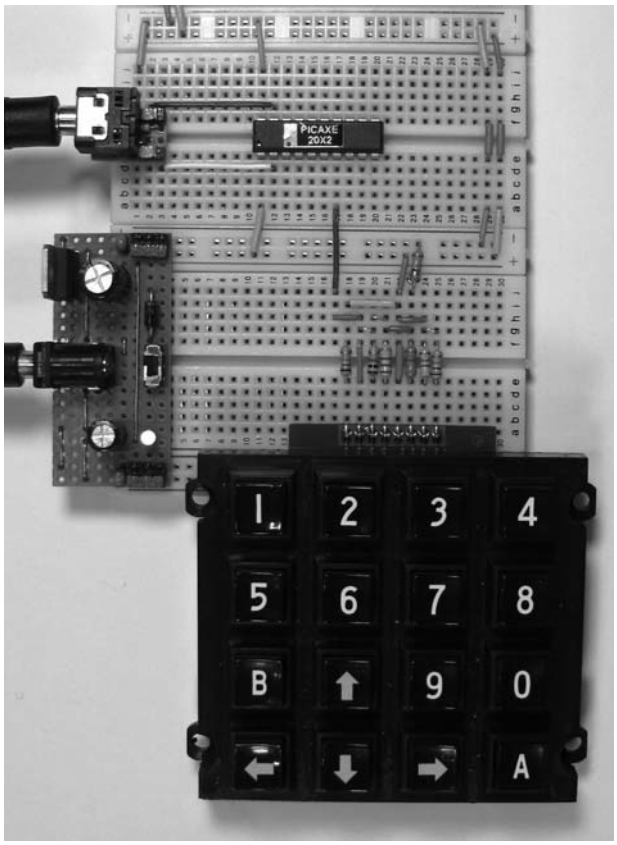


Figure 13-5 Breadboard layout for the rotated keypad circuit

words, if you start at the bottom (which used to be the top before our little operation), the first three characters ("1," "2," and "3") are the same. Next, "A" has been replaced with "4," "4" has been replaced with "5," etc. Finally, rather than using "*," "#," "C," and "D," I substituted "U" for "Up," "D" for "Down," "L" for "Left," and "R" for "Right" so that I can easily remember the character that stands for each cursor key.

Download KeypadNew.bas to your breadboard circuit and test all the keypresses. (Don't forget that you may need to change the theoretical values in the *case* statements to the actual values you obtained earlier.) You should see the correct character appear in the terminal window in response to each keypress. If not, double-check each of the *case* statements in the *select case* command. When everything is working correctly, we're ready to begin working on our countdown timer project.

Project 13
Constructing a Countdown Timer

Now that we have our new keypad functioning correctly, the hardware portion of our countdown timer project is really simple. In addition to the keypad, all we need is the four-digit LED display we developed in Chapter 12 and a piezo buzzer—the one we used in Chapter 5 would be fine. Figure 13-6 presents the schematic for the complete circuit, and Figure 13-7 is a photo of my breadboard setup.

In order to understand the details of our project's software, we need to discuss how the timer is actually used. When the timer is first powered up, "0000" is displayed on the LEDs. Each time the user presses a digit key, that digit enters the display from the right; as additional digits are entered, the display scrolls to the left. If an entry error occurs, each press of the "Back" key will scroll the display one position to the right and remove the most recently entered digit. When the correct time has been entered as "MM.SS," pressing the "Accept" key will begin the timer countdown. The countdown can be aborted at any time by pressing the "Back" key. When the countdown is completed, the usual annoying beeps will occur until the user presses the "Back" key to reset the timer to "0000."

The program for our timer project (TimerDown.bas) is presented in Listing 13-2. It may appear a little daunting at first, due to its length. However, most of it is composed of code modules that we have already discussed, so I won't repeat those details. The new portions can be roughly divided into the following three areas: the use of the *runflag* variable, the second *select case* command, and the *interrupt* subroutine. After you have browsed through the program listing, we'll address each of these aspects.

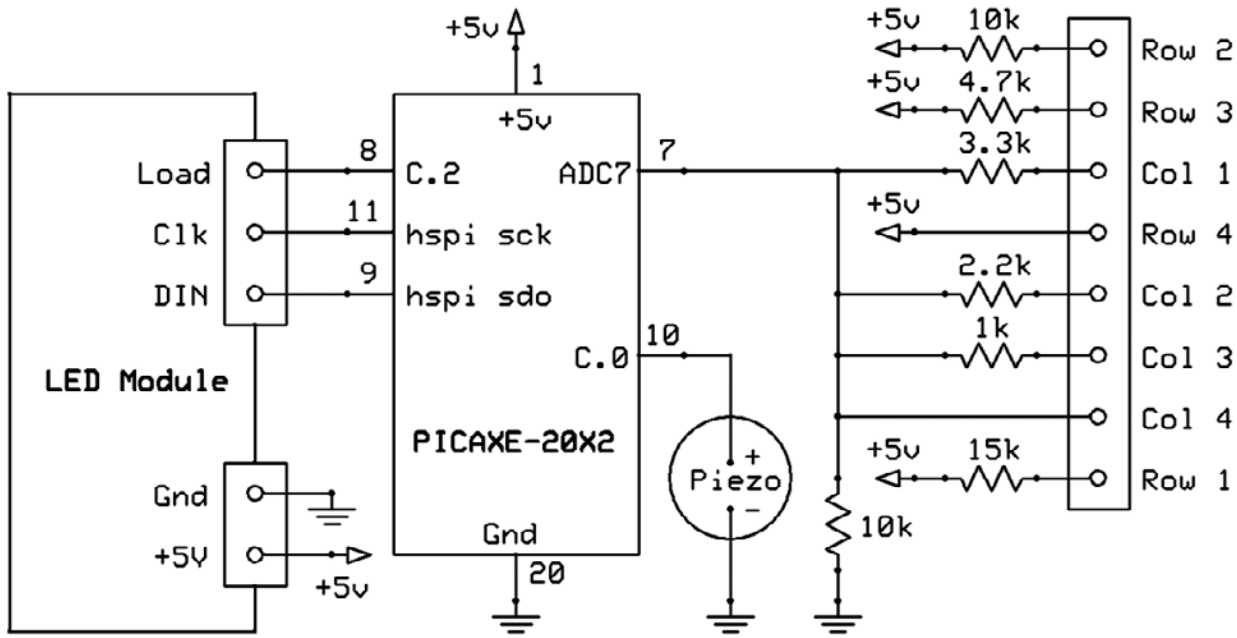


Figure 13-6 Schematic for countdown timer project

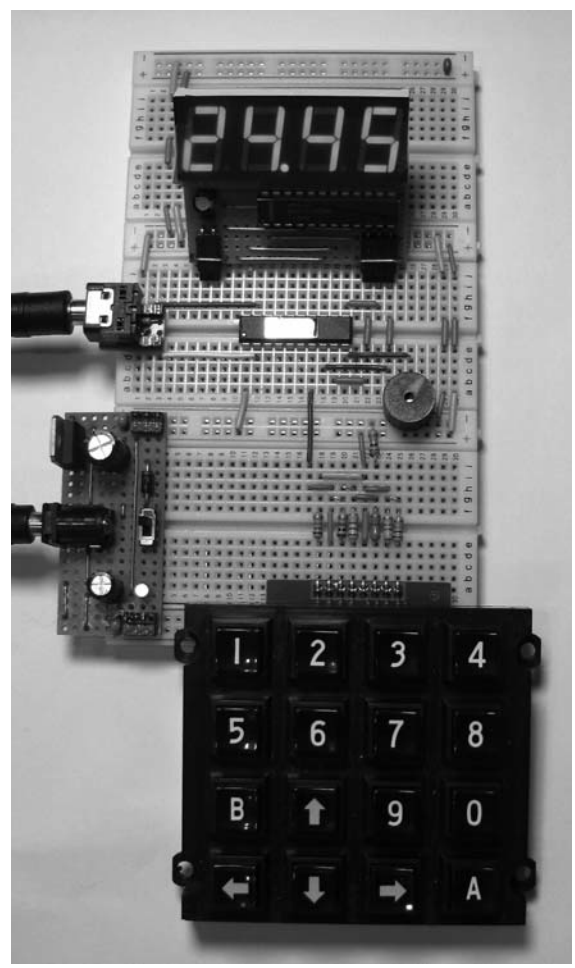


Figure 13-7 Breadboard setup for countdown timer project

LISTING 13-2

```
' ================================ TimerDown.bas ==================================
' This program runs on a PICAXE-20X2 & implements a countdown timer.
' The time (minutes & seconds) is displayed on a 4-digit, 7-segment
' LED display that is controlled by a MAX7219 LED display driver.
' A 16-key matrix keypad provides user input to set the timer.

' === Constants ===
' Hardware Interface
  symbol keypad = 7                     ' keypad input on ADC7 (C.3)
  symbol load   = C.2                   ' pulse C.2 to xfer data to 7219
  symbol piezo  = C.0                   ' piezo beeper on C.0
' Register addresses for the 7219
  symbol decode =  9                    ' decode register
  symbol brite  = 10                    ' LED intensity register; 15 = max
  symbol scan   = 11                    ' scan-limit register
  symbol on_off = 12                    ' 1 = display on; 0 = display off

' === Variables ===
  symbol  key = w0                      ' ADC key value
  symbol junk = b2                      ' temporary variable
  symbol mins = b3                      ' minutes
  symbol secs = b4                      ' seconds
  symbol  rep = b5                      ' repetitions in for/next loop
  symbol   d0 = b6                      ' display on 1st LED from right
  symbol   d1 = b7                      ' display on 2nd LED from right
  symbol   d2 = b8                      ' display on 3rd LED from right
  symbol   d3 = b9                      ' display on 4th LED from right
  symbol decimal = b10                  ' used to insert decimal point
  symbol runflag = b11                  ' 0 = setting up / 1 = running

' === Directives ===
  #com 3                                ' specify the serial port
  #picaxe 20X2                          ' specify the PICAXE processor
  #no_data                              ' speed up the download
  #no_table                             ' speed up the download
  #terminal off                         ' no terminal window

' ============================== Begin Main Program ==============================
  setfreq m16                           ' set clock speed
  hspisetup spimode00, spifast          ' set up hspi
  dirsb = %11111111                     ' PortB all outputs
  dirsc = %10110111                     ' ADC keypad input on C.3
  pullup  %00000010                     ' enable C.6 pullup (see Manual)
  adcsetup = %0000000010000000          ' configure ADC7 (see Manual)

' Initialize MAX7219
  hspiout (scan,3)                      ' set scan limit for digits 0-3
```

(continued)

LISTING 13-2 (continued)

```
  pulsout load,1
  hspiout (brite,5)                    ' set brightness to 5 (15 = 100%)
  pulsout load,1
  hspiout (decode,15)                  ' set BCD decoding for digits 0-3
  pulsout load,1
  hspiout (on_off,1)                   ' turn display on
  pulsout load,1

do
  gosub UpdateDisplay

wait_for_keypress:
  readadc keypad, junk
  if junk < 5 then wait_for_keypress
  pause 100                            ' debounce key
  readadc10 keypad, key                ' get ADC value

wait_for_release:
  readadc keypad, junk
  if junk > 5 then wait_for_release

  select case key                      'decode keypress
    case < 369 : key = 10              ' A
    case < 385 : key = 14              '
    case < 402 : key = 12              ' D
    case < 434 : key = 13              ' L
    case < 450 : key =  0              ' 0
    case < 474 : key =  9              ' 9
    case < 500 : key = 15              ' U
    case < 551 : key = 11              ' B
    case < 587 : key =  8              ' 8
    case < 634 : key =  7              ' 7
    case < 674 : key =  6              ' 6
    case < 733 : key =  5              ' 5
    case < 804 : key =  4              ' 4
    case < 885 : key =  3              ' 3
    case < 977 : key =  2              ' 2
    else       : key =  1              ' 1
  end select

  select case key
    case 0 to 9                        ' digit key has been pressed
      If runflag = 0 then
        d3=d2                          ' rotate left and add keypress
        d2=d1
        d1=d0
        d0=key
```

LISTING 13-2 (continued)

```
      endif
    case 10                              ' "Accept" key has been pressed
      If runflag = 0 then
        runflag = 1
        mins = 10 * d3 + d2              ' initialize the timer display
        secs = 10 * d1 + d0
        settimer t1s_16                  ' set timer1 to 1S interrupts
        timer = $ffff                    ' interrupt on next count ($0)
        setintflags %10000000,%10000000  ' enable timer interrupt
      endif
    case 11                              ' "Back" key has been pressed
      If runflag = 0 then
        d0=d1                            ' rotate right and add "0"
        d1=d2
        d2=d3
        d3=0
      else
        reset                            ' stop the annoying beeping!
      endif
    case 12 to 15                        ' ignore arrow keys
  end select
loop

' ================== End Main Program - Subroutines Follow ====================
Interrupt:
  toflag = 0                             ' clear timer overflow flag
  settimer t1s_16                        ' set timer1 to 1 Sec interrupts
  timer = $ffff                          ' interrupt on next count ($0)
  setintflags %10000000, %10000000       ' re-enable timer interrupt
  dec secs                               ' update seconds
  if secs = 255 then                     ' if seconds wrap from 0 to 255
    secs =  59                           ' the reset seconds to 59
    dec mins                             ' and decrement minutes
  endif
  d3 = mins dig 1                        ' 4th digit from right
  d2 = mins dig 0                        ' 3rd digit from right
  d1 = secs dig 1                        ' 2nd digit from right
  d0 = secs dig 0                        ' 1st digit from right
UpdateDisplay:
  hspiout (1,d3)                         ' 4th digit from right
  pulsout load,1
  decimal = d2 + 128
  hspiout (2,decimal)                    ' 3rd digit from right
  pulsout load,1
  hspiout (3,d1)                         ' 2nd digit from right
  pulsout load,1
  hspiout (4,d0)                         ' 1st digit from right
```

(continued)

LISTING 13-2 (continued)

```
  pulsout load,1
  if runflag = 1 AND mins = 0 AND secs = 0 then gosub wait_for_reset
  return

Paws:
  for junk = 1 to 120
    pause 1
    if pinC.3 = 1 then                  ' press "back" key to reset timer
      reset
    endif
  next junk
  return

WaitForReset:
  setintflags off                       ' disable interrupts
  adcsetup = 0                          ' disable ADC to use C.3 digitally

 do
    for rep = 1 to 4
      sound piezo, (115,10)
      gosub Paws
    next rep
    for rep = 1 to 3
      gosub Paws
    next rep
  loop
  return
```

In order to understand the TimerDown.bas program, it's best to conceptualize it as having two modes of operation: setup and run. This distinction is important because the same keypress is processed differently in each operational mode. For example, during the *setup* phase, the digit keys are used to enter the desired time, but during the *run* phase, they are ignored. The function of the *runflag* variable is to keep track of which mode is currently active. When the program starts running, it's in the *setup* mode and all the variables (including *runflag*) have been automatically initialized to 0; when the user presses the "Accept" key, *runflag* is set to 1 and the timer is started.

From the beginning of the program through the completion of the first *select case* statement, the code should all be familiar. The second *select case* statement is a major part of the code that is new in this program. Each time a key is pressed, the first *select case* statement assigns the appropriate value (from 0 to 15) to the *key* variable. The second *select case* statement then decides how to respond to the keypress. Since this is the heart of the program, let's examine each of the four cases individually.

- **Case 0 to 9 (digit keys):** If runflag = 0 (setup mode), then the display is shifted left one by one digit and the key value is displayed on digit 0 (first from right). Since there is no else clause, if runflag = 1 (run mode), then nothing happens (i.e., the digit key is ignored).

- **Case 10 ("Accept" key):** If runflag = 0 (setup mode), then it's changed to 1 (run mode), the minutes and seconds variables are set equal to the displayed values, and the one-second timer interrupt is enabled. If runflag = 1 (run mode), then nothing happens (i.e., the "Accept" key is ignored).
- **Case 11 ("Back" key):** If runflag = 0 (setup mode), then the display is rotated right and the most recently entered key value is discarded. If runflag = 1 (run mode), then the else clause executes and the timer is reset to the setup mode.
- **Case 12 to 15 (arrow keys):** Because there is no code in this case, the arrow keys are always ignored by the program.

The last major part of the program that requires explanation is the *interrupt* subroutine. The important thing to keep in mind about interrupts is that as soon as the *interrupt* subroutine is executed, interrupts are automatically disabled. If they weren't, it would be theoretically possible for another interrupt to occur while a previous one was still being processed, which could cause a program to malfunction. Because our interrupt is only occurring once a second, this isn't an issue for us. However, since the compiler automatically disables interrupts, our subroutine needs to re-enable the interrupt each time it executes. The remainder of the *interrupt* subroutine simply updates the *secs* and *mins* variables, as well as the four-digit LED display. One other aspect of the *interrupt* subroutine (and the initial enabling of interrupts) needs to be explained. In both cases, the *timer* variable is set to (hexadecimal) $ffff. Therefore, *timer* will overflow (from $ffff to $0000) and trigger an interrupt the next time that it is incremented, which will be one second later because the appropriate *preload* value has been used.

The *WaitForReset* subroutine only executes when the timer has counted down to zero. It disables the ADC input so that the C.3 pin can be used digitally in the *Paws* subroutine. It also produces the annoying beeps and calls the *Paws* subroutine, which repetitively checks for a keypress indicating that we can't stand the beeps anymore. The *Paws* subroutine may seem a little strange, but it is the simplest way I could get the results I wanted. Essentially, *Paws* subdivides a much longer *pause* command so that a keypress can be tested much more frequently. Also, I didn't want to have to include an additional "Reset" key, so I used the C.3 input that's used for the ADC keypad input, but this time I used it digitally. Pressing the "Back" key produces a high digital level that resets the program immediately. In fact, the 1 through 8 keys and the "up arrow" key do exactly the same thing, but we won't tell that to the user!

I think that covers all of the important aspects of the program, so download it to your 20X2 master processor and try it out. You may want to check the accuracy of the timer against another timer or clock that you know is accurate because, as I mentioned, the internal resonator of the 20X2 is not as accurate as the external resonators that are available on the other X2 processors. When I first tested my timer, it gained approximately two seconds per hour. If you require more precision, the simplest approach would be to experiment with adjusting the value of the *preload* constant. Instead of using the default value *(t1s_16 = 3036)*, you could increase (or decrease) it a little to slightly increase (or decrease) the rate of the interrupts. I was able to reduce my timing error to approximately one second per hour by using 3041 for my *preload* value. For even more precision, you could use a separate time-keeping chip such as the Dallas/Maxim DS1307 real-time clock, which uses a crystal-controlled time-base that's highly accurate.

This page intentionally left blank

CHAPTER 14

Constructing a Programmable Multifunction Peripheral Device

In this chapter and the next, we're going to focus on the development of a programmable, multifunction peripheral device (MPD) that we'll be able to program to implement a variety of useful functions. You may be relieved to learn that there is almost no theory to discuss in this chapter; we'll be focusing exclusively on the construction details of the MPD. In case you're disappointed by the lack of theory, there's a fair amount to look forward to in Chapter 15 when we develop a major application for the MPD.

Project 14
The Evil Genius Multifunction Peripheral Device

In order to have a clear sense of where we're going, let's start at the end! Figure 14-1 presents a photo of the completed MPD. Essentially, it integrates two peripheral devices that we have already discussed in detail. However, in this case, the whole equals much more than the sum of its parts. As you can see in Figure 14-1, the MPD includes a 16-key keypad for user input and a 16-character by 2-line LCD display for output. What isn't visible in the photo is the 20X2 processor inside the plastic enclosure that enables the MPD to implement an almost limitless variety of useful tasks.

Figure 14-2 presents the MPD schematic. The interface to our rotated matrix keypad is identical to that of our previous project in Chapter 13, except that pin B.4 is used for the analog input, rather than C.3—we'll find out why in Chapter 15. The LCD interface is also almost identical to the one we discussed in Chapter 10. The only difference is that the MPD uses pin C.3 as a

Figure 14-1 The completed Evil Genius multifunction peripheral device (MPD)

replacement for B.4, and A.0 as the replacement for B.6—again, we'll discuss the reasons for these substitutions in Chapter 15.

The breadboard interface, which is implemented with a nine- or ten-pin ribbon cable, provides power for the project and also includes three I/O lines for interfacing with various breadboard circuits. Most importantly, the 20X2's B.6 pin is available on the breadboard connector. As you may remember, B.6 is the 20X2's *hserin* line, so its availability to our breadboard projects means that the MPD is capable of receiving serial data in the background. This, of course, also means that the MPD is ideally suited to function as an I/O terminal for any project that needs one.

The remainder of the schematic is simple: just the standard PICAXE programming connection and a piezo beeper for audible output. Actually, I ended up replacing the piezo with a small 16-ohm speaker—we'll discuss that possibility when we get to the construction of the MPD. Finally, I didn't include the size of the current-limiting resistor for the LCD backlight (pin 15 on the LCD connector) because it depends on the specific LCD that you use. I used the SparkFun #LCD-00791 16 × 2 LCD with red characters on a black

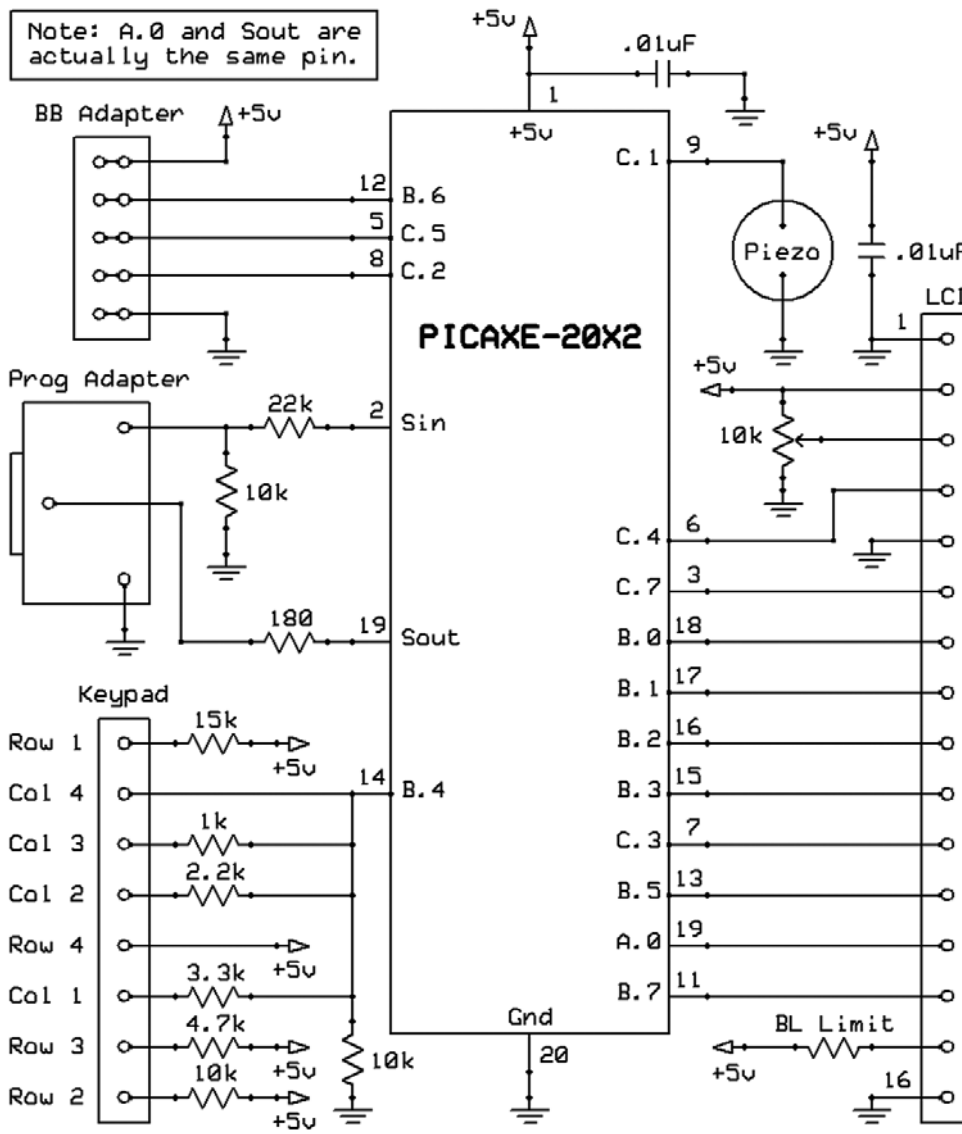


Figure 14-2 MPD schematic

background. This display does not require a current-limiting resistor for the backlight, so I just connected pin 15 directly to +5V.

The MPD stripboard layout is our largest one yet. In order to make the details easier to see, the layout is presented in two separate figures: Figure 14-3 shows the top view, and Figure 14-4 shows the bottom view. The size of the stripboard was determined by the plastic enclosure that I used, and that's a subject that we need to discuss in detail before going any further. Plastic enclosures certainly make a completed project look professional, but cutting the necessary openings in the plastic can also be the most difficult part of the project. You can certainly construct the MPD and simply mount it on a small piece of wood, but if you're up for a challenge, I'll discuss the relevant details of all the steps that I took to modify the enclosure and get everything to fit correctly.

The plastic enclosure that I used is a PacTec #60616-510-000-HP that I purchased from www.mouser.com, but you can also order it directly from www.pactecenclosures.com. The exterior dimensions are 5.7 inches (14.5 cm) long by 3.6 inches (9.1 cm) wide by 1.1 inches (2.8 cm) deep. If you already have another enclosure that you want to use, just make sure it's at least that big in all three dimensions. In fact, the 1.1-inch depth required a little extra work to get things to fit—a depth of at least 1.2 inches would have simplified the process. If you do use a different enclosure, you may need to modify the size of the stripboard accordingly.

The complete parts list for the MPD is presented next. With the exception of the LCD, the plastic enclosure, and the hardware used to attach things to the enclosure, all the parts are available on my website. Also, I don't have 8-pin straight

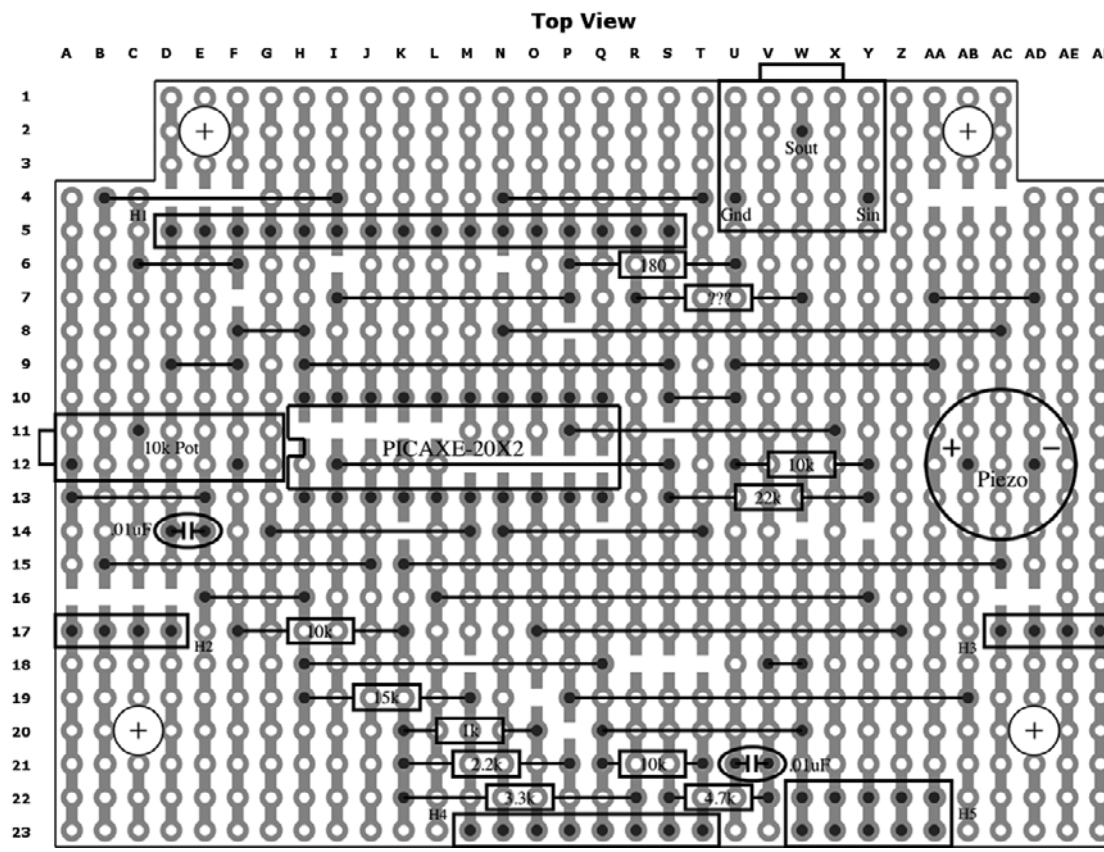


Figure 14-3 Top view of MPD stripboard layout

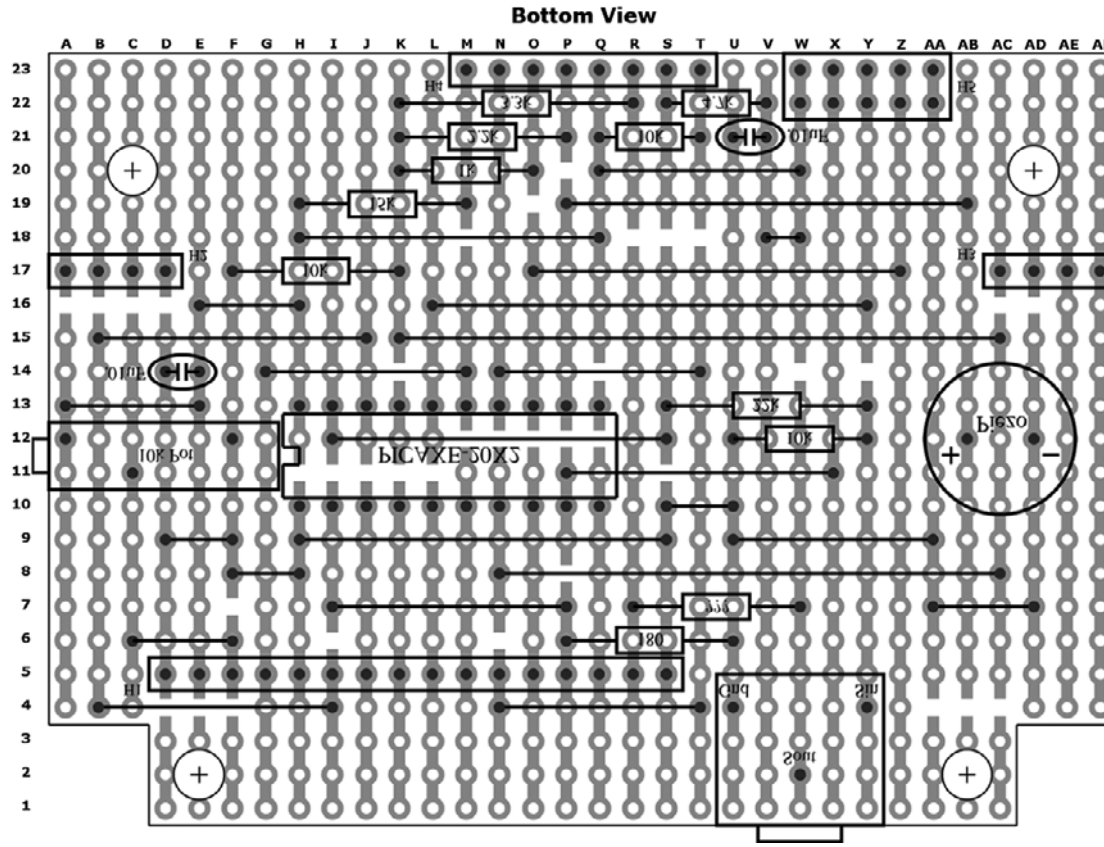


Figure 14-4 Bottom view of MPD stripboard layout

female headers, but you can use two 4-pin headers, or cut an 8-pin section from a 16-pin header by sacrificing an adjacent pin and sanding or filing the excess plastic.

Preparing the Stripboard

As usual, the first thing you need to do is cut and sand a piece of stripboard to the required size (32 traces with 23 holes each if you are using the same enclosure I did). However, because the stripboard needs to fit correctly into the enclosure, a fair amount of additional preparation is necessary before actually assembling the circuit. Start by cutting the two notches as indicated earlier in Figure 14-3, but don't drill the four mounting holes yet. Test-fit the board in the enclosure, making sure that the top edge of the board can fit tightly against the edge of the bottom half of the enclosure. Next, securely tape the stereo connector to the board in the proper position—the round portion of the connector should overhang the stripboard slightly. Then slide the board back into the enclosure, centering it from left to right. Add a piece of masking tape to the outside of the enclosure, and mark the exact location of the round portion of the stereo connector (making sure the board is centered) so that you can drill a 1/4-inch (6.4-mm) hole to accommodate the round portion of the connector. Remove the board and drill the hole in the edge of the enclosure. Test-fit the board again; if the hole doesn't line up exactly, enlarging it slightly should do the job. You can use a sharp hobby knife to shave off any plastic that interferes with the connector fitting into the hole.

When you are satisfied with the fit of the stereo connector in the hole in the enclosure, remove the connector from the stripboard and follow the same procedure to locate and drill a hole for access to

PARTS BIN

ID	Part
—	Stripboard, 32 traces by 23 holes each
—	Two capacitors, .01μF each
—	Resistor, ???Ω, 1/4 W (see text)
—	Resistor, 180Ω, 1/4 W
—	Resistor, 1k, 1/4 W
—	Resistor, 2.2k, 1/4 W
—	Resistor, 3.3k, 1/4 W
—	Resistor, 4.7k, 1/4 W
—	Three resistors, 10k, 1/4 W each
—	Resistor, 15k, 1/4 W
—	Resistor, 22k, 1/4 W
—	Potentiometer, 10K
—	IC socket, 20 machined pins
—	PICAXE-20X2
H1	Female header, straight, 16 pins
H2, H3	Female header, straight, 4 pins (see text)
H4	Female header, straight, 8 pins
H5	Male header, straight, 5 × 2 pins
—	Stereo connector, low-profile
—	Piezo or small speaker (see text)
—	Ribbon cable, 9 or 10 wire, 6 inches long
—	Two IDC connectors, 5 × 2 male
—	LCD, 16 × 2 (see text)
—	Keypad, 4 × 4 matrix (rotated)

the adjustment screw on the potentiometer. Once that's done, you're ready to drill the four mounting holes as shown earlier in Figure 14-3. The diameter of the holes depends upon the hardware you use to mount the board in the enclosure—7/64 inch (2.8 mm) works well for #4 bolts, and 9/64 inch (3.6 mm) is big enough for #6 bolts. In order to support the completed MPD at a convenient angle, I used a 3/16-inch (4.8-mm) bit to drill the two holes near the bottom corners of the stripboard, and then attached the stripboard to the enclosure with two 4.25-inch (11-cm) pieces of #10 threaded rod. The photo in Figure 14-5 is a side view of my completed MPD, which shows the arrangement of the supports. In the photo, I have covered the threaded rods with black heat-shrink tubing and added small rubber feet on the ends of the rods to protect my desk and to keep the MPD from sliding around when the keys are pressed.

When you have chosen the hardware you want to use, drill the four mounting holes in the stripboard, but not yet in the enclosure. Next, retape the stereo connector and potentiometer to the board at the appropriate locations and double-check that everything fits correctly—make sure that the USB programming cable plugs into the stereo connector correctly, and that you can use a small screwdriver to adjust the potentiometer through the hole in the enclosure. When everything fits correctly, clamp the stripboard to the enclosure

Figure 14-5 Side view of completed MPD

(making sure it's centered from left to right); you may need to shim under the board to avoid bending it excessively. Using the holes in the stripboard as guides, drill the same-sized mounting holes in the bottom of the enclosure.

Assembling the Stripboard Circuit

To help clarify the following assembly procedures, Figure 14-6 presents a close-up photo of my completed stripboard circuit installed in the enclosure. However, my circuit does differ slightly from that of the schematic (Figure 14-2) and layout (Figure 14-3) presented earlier, so don't use Figure 14-6 as an assembly guide; just use it as a guide for installing the stripboard in the enclosure.

There are a couple of reasons for the differences in the circuit that I assembled. First, when I originally completed and tested the circuit, I found that the volume of the sound produced by the piezo wasn't as loud as I would like, so I replaced it with a small 16Ω speaker that is much louder. This turned out to involve more work than I thought because the speaker, which is only about 1.2 inch (3 cm) in diameter, was too large to fit in the space between the stripboard and the LCD. I ended up drilling a circular pattern of holes in the back of the enclosure underneath the keypad, mounting the speaker with silicone adhesive, and running its leads back to the stripboard.

If you decide to make a similar modification, be sure to read the documentation for the *tune* command in Part II of the manual, because it's necessary to add a 10µF electrolytic capacitor and a current-limiting resistor to the speaker circuit. You can see the capacitor I added just above the four-pin header near the lower-right corner of the stripboard. You can't see the 47Ω resistor that I

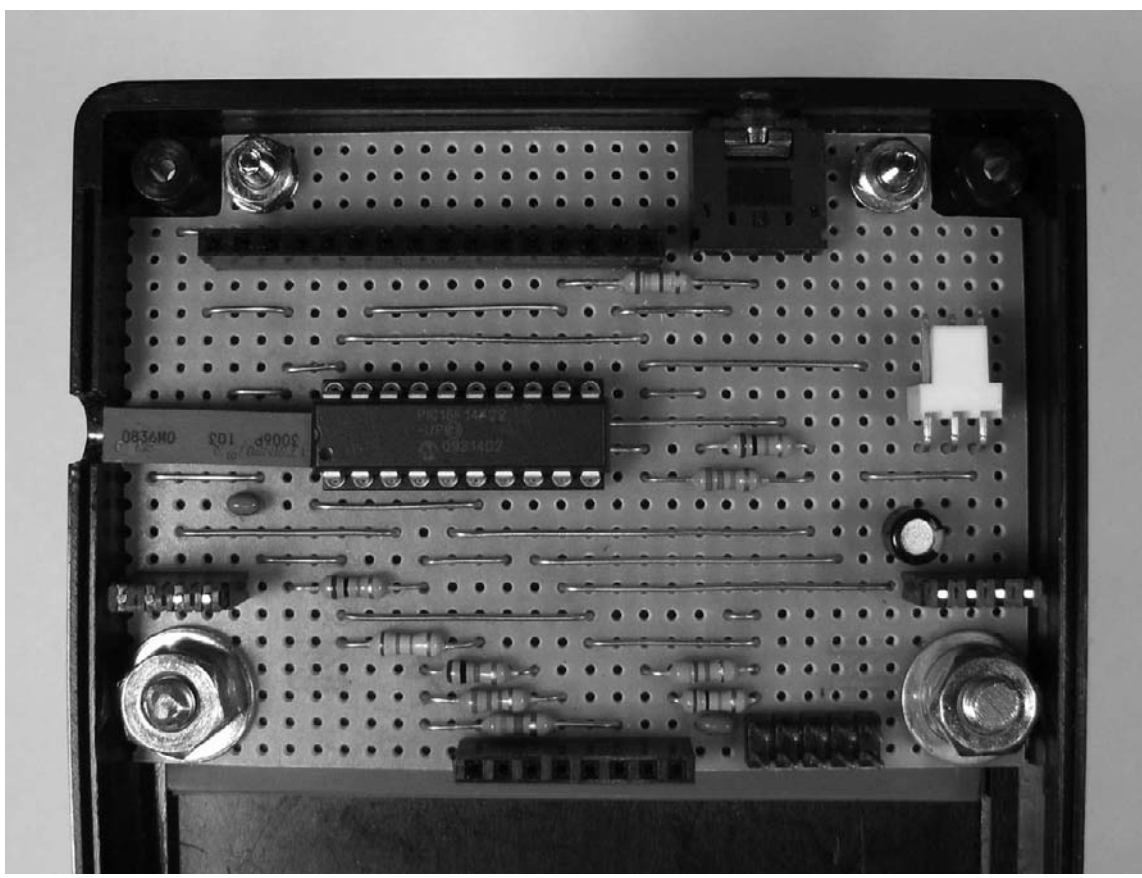


Figure 14-6 Completed stripboard circuit installed in the bottom of the enclosure

used because I soldered it directly to one of the speaker contacts—it was simpler to do it that way. I also added the three-pin, right-angle header that you can see in Figure 14-6 (I didn't have a two-pin header available) so that I can disconnect the board from the speaker if needed. You could also solder the speaker leads directly to the stripboard if you prefer.

The second reason for the differences in my circuit is a little more involved. When I first designed the circuit, I intended to use the same software that we discussed in Chapter 11 to interface the keypad. However, after experimenting with the MPD for a while, I realized that it would be much more efficient to employ an interrupt-based approach to obtaining user input. To do so required some modifications to the circuit. The schematic presented earlier in Figure 14-2 and the layout of Figure 14-3 both reflect the necessary changes. However, rather than constructing an entirely new stripboard, I took the easy way out and "ugly-wired" the changes on the board that I had already built.

So use Figures 14-2 and 14-3 to construct your version of the MPD and Figure 14-6 for reference when you install the stripboard circuit in your enclosure. When we get to the point where we test the completed MPD, we'll just use the same approach that we did in Chapter 11. In Chapter 15 we'll get into the details of the interrupt-driven approach. In spite of the additional complexity, I think you'll agree that it's a much more powerful method of processing user input.

Before we actually begin construction of the MPD stripboard, one other aspect of the photo in Figure 14-6 requires clarification. When the stripboard is installed in the enclosure, it rests on two thick plastic ridges on each side of the enclosure. As a result, tightening the four mounting bolts causes the stripboard to flex downward. To avoid possibly cracking the board, you will need to include some sort of spacer underneath each mounting point. I used a rubber pad that I happened to have on hand and just sandwiched it underneath the entire board. The only important consideration is that whatever you use should be nonconductive.

The following list of assembly instructions is based on the original piezo version of the circuit. If you decide to use a speaker, just make the necessary modifications. Before beginning the assembly, make sure that you have chosen the correct current-limiting resistor for the backlight of the LCD that you will be using and read through the complete list of assembly instructions that follows to be certain you understand the entire procedure.

1. Remove the stereo connector, the potentiometer, and any tape residue from the board.
2. Sever the traces on the bottom of the board at the 36 holes shown in Figure 14-3.
3. Clean the bottom of the board with a plastic Scotch-Brite or similar abrasive pad.
4. Insert all the jumpers; solder and snip their leads.
5. Insert all the resistors; solder and snip their leads. (The size of the resistor marked "???" depends upon the specific backlit LCD you are using.)
6. Insert the two .01µF capacitors; solder and snip their leads.
7. Insert and solder the 20-pin socket in place (make sure pin 1 is at I13).
8. Insert and solder the low-profile stereo adapter.
9. If you add the 10µF capacitor, insert it at this point; solder and snip its leads.
10. Insert the 10k potentiometer; solder and snip its leads.
11. Insert the 5 × 2 straight male header from the top of the board, with the longer ends of the pins pointing up; solder it in place.

12. Insert all the straight female headers. Flip the board over, support the headers on a flat surface, and solder the headers in place.
13. If you decide to use a piezo, insert it at this point; solder and snip its leads.
14. File or sand all the cut leads on the bottom of the board.
15. Clean the flux from the bottom of the board and allow it to dry.
16. Inspect the board carefully for accidental solder connections or other problems.

Installing the Components in the Enclosure

When you have completed the assembly procedure, use the photo in Figure 14-6 as a guide for installing the stripboard in the upper portion of the bottom of the enclosure. Once that's done, we're ready to complete the installation of the remaining components. Figure 14-7 is a photo of my MPD after installing the speaker, the ribbon cable, and the rotated keypad. It may be hard to see in the photo, but I had to cut away a small portion of the keypad's circuit board to the right of the header pins in order to insert the ribbon cable connector into the 5 × 2 header on the stripboard. To make the cable, I used a 6-inch (15-cm) piece of 10-wire ribbon, but 9-wire cable will also work fine. The orientation of the cable and the two 5 × 2 IDC connectors is important. With the (red) stripe on the left (as shown in Figure 14-7), install the 5 × 2 IDC connector in the photo so that it's facing *down*. Also, you will need to remove a thin slice of plastic from the enclosure where the ribbon cable exits; a sharp hobby knife will do the job quickly and cleanly.

The orientation of the connector on the other end of the cable depends on how you want to connect it to your breadboard circuits. Figure 14-8 is a close-up photo of my breadboard connection.
I decided to install the connector facing up so that any unused pins wouldn't be connected to the breadboard. If you opt for this approach, you can use jumper wires to make any connections you need. Of course, you can also connect power and ground (pins 1 and 5, respectively) with jumpers, but I used two tiny pieces of stripboard and soldered male headers to them so that the connection is rigid and holds the ribbon cable connector firmly in place. It doesn't show in the photo, but I painted one red and the other black to remind myself of the correct connections. One other detail is worth mentioning. If you look closely at a 5 × 2 ribbon cable connector, you will see that one side (the top side in the photo) has a raised area; this is designed to fit into polarized male connectors to avoid the possibility of inserting the connector backwards. For our application, be sure to orient the connector so that the raised area

Figure 14-7 MPD with ribbon cable and keypad attached

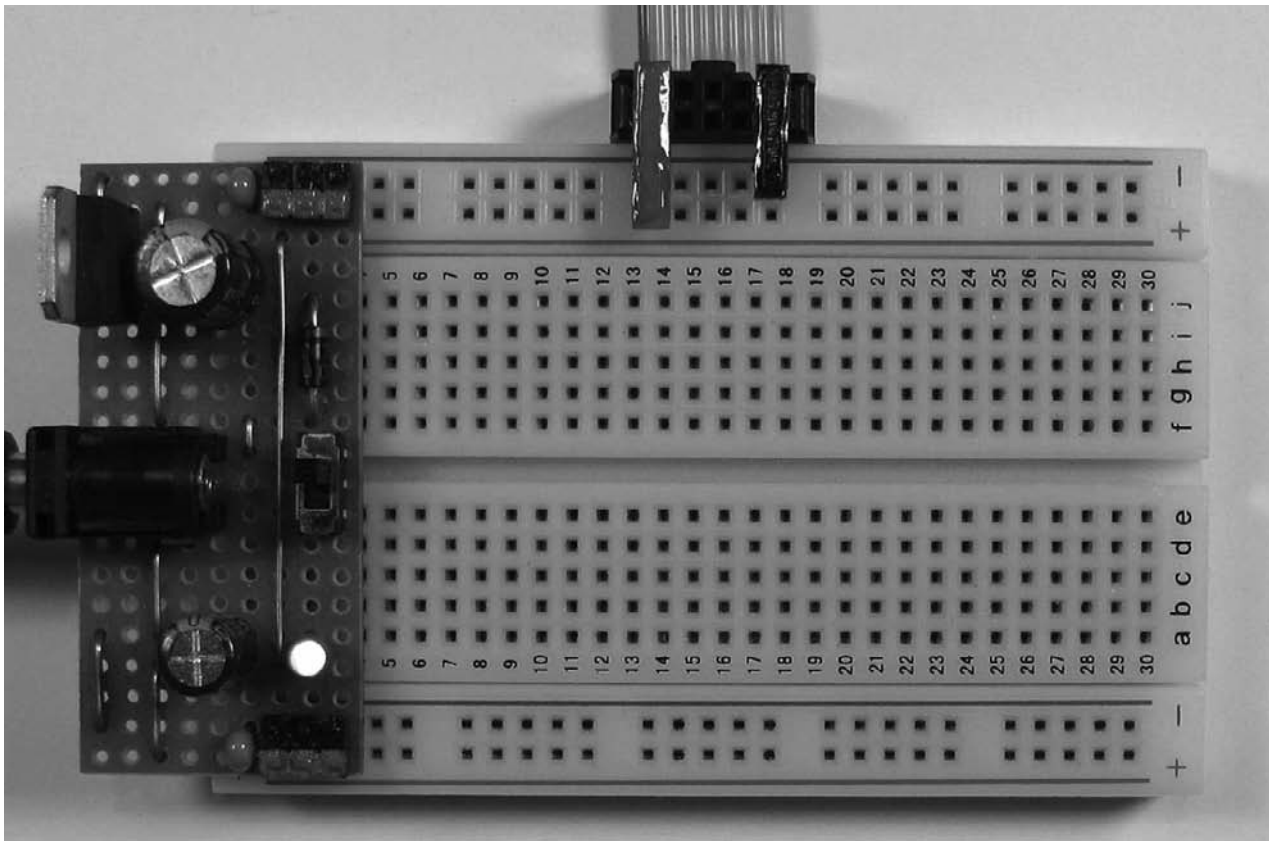


Figure 14-8 MPD ribbon cable attached to a breadboard

is on the side that doesn't abut the breadboard; this orientation results in the exact 0.1-inch (2.5-mm) spacing needed for preformed jumpers or the tiny stripboard connectors that I used.

My original intention was to bolt the keypad to the bottom of the enclosure, but a funny thing happened on the way to implementing that intention. When I cut the openings in the top of the enclosure for the LCD and keypad (we'll get to that shortly), I discovered that the keypad was about 0.1 inch (2.5 mm) too high to allow the enclosure pieces to fit together properly. After a brief moment of panic, I realized that I could (carefully) remove the black plastic from the keypad's header pins and snip 0.1 inch off each pin. This enabled the keypad to sit low enough so that the two halves of the enclosure fit together properly. In fact, the fit is so exact that the keypad is, in effect, pressed between the top and bottom of the enclosure. As a result, I decided not to attach the keypad to the enclosure at all. Rather, I added the two bolts that you can see in Figure 14-7 (near the left arrow key and the "A" key) and adjusted them with washers directly under their heads until the keypad sat perfectly level. With this arrangement, the enclosure firmly holds the keypad in place without any additional mounting bolts.

Figure 14-9 is a photo of the MPD with the LCD inserted into the stripboard header. Before you do that, don't forget to snip the short ends of the pins from two 4-pin male headers and insert them into the two 4-pin female headers on the stripboard so that the LCD will sit parallel to the stripboard (as we did in Chapters 9 and 10). Coincidentally, the LCD ends up at exactly the same height as the frame of the keypad. I would like to be able to say that I planned it that way, but occasionally good things just happen to Evil Geniuses!

Figure 14-9 MPD with ribbon cable, keypad, and LCD attached

Figure 14-10 Top of the enclosure with the completed openings for the keypad and LCD

And Now for the Hard Part!

I saved the worst for last: cutting the openings in the enclosure for the LCD and keypad. I think the best approach is to measure everything very carefully and make a paper template to tape on the top of the enclosure. To avoid potential disaster, draw the two openings approximately 0.1 inch (2.5 mm) too small in each dimension. The bulk of the plastic can be easily removed with a drill and a spade-bit, but the finishing work is tedious at best. You can use a small hobby saw, a powered rotary tool, or even a hand file. The main thing is to work slowly and carefully, and check your work frequently by test-fitting the enclosure. When the openings are close to their final sizes, you may find it helpful to use masking tape to outline the desired edges and then carefully sand or file to the edges. Figure 14-10 is a photo of the top of my enclosure after I finished filing the two openings. In it, you can see the minor mistakes that I made; fortunately, everything is black so these little mistakes aren't noticeable at all when the enclosure is completely assembled.

Testing the Completed Multifunction Peripheral Device

When everything fits together correctly, we're ready to test the MPD. (You may not want to actually secure the two halves of the enclosure at this point, in case you need to do a little troubleshooting to get everything to function properly.) To test the keypad, connect the MPD to a powered breadboard as shown earlier in Figure 14-8 and download the TestKeypad.bas program shown in Listing 14-1. Similarly to our keypad-testing program in Chapter 11, you should see an

LISTING 14-1

```
' ================================ TestKeypad.bas ================================
' Program decodes a "rotated" ADC 4X4 matrix keypad


' === Variables ===
  symbol key  = w0                              ' ADC key value
  symbol junk = b2                              ' throwaway char


' === Directives ===
  #com 4                                        ' specify com port
  #picaxe 20X2                                  ' specify processor
  #no_data                                      ' save time downloading
  #no_table                                     ' save time downloading
  #terminal 9600                                ' open terminal


' ============================ Begin Main Prograqm ==============================

  dirsB = %11101111                             ' set pinB.4 as input
  adcsetup = %0000000001000000                  ' set ADC6 (see Manual)

do
  wait_for_keypress:
    readadc 6, junk
    if junk < 5 then wait_for_keypress
    pause 25                                    'debounce keypress
    readadc10 6, key                            'get Key value

  wait_for_release:
    readadc 6, junk
    if junk > 5 then wait_for_release
    sertxd ("key = ",#key,cr,lf)                'send Key to terminal
loop
```

ADC value in the terminal window each time you press a key. Since you used a different set of resistors to construct the circuit, these values may differ slightly from those of Chapter 11. Make a list of all the values; you will need them later when we decode the ADC values.

When you're sure the keypad is functioning correctly, we're ready to test the LCD portion of the MPD. To do so, we'll use the following TestLCD.bas program (Listing 14-2); it's essentially the same as the software we discussed in Chapter 10, except that we're now replacing pin B.4 with pin C.3 and B.6 with A.0. Download TestLCD.bas to your MPD; you should see the two messages alternating on the LCD, with each one accompanied by a short "beep" from the piezo or speaker. If not, you're in for a little troubleshooting session!

When all your MPD components are functioning correctly, we can move on to the final

LISTING 14-2

```
' ================================ TestLCD.bas ==================================
' Program runs on a PICAXE-20X2; sends 8-bit data to 16X2 LCD display


' === Constants ===
  symbol cmnd   = 0                    ' used to set up for cmd/txt byte
  symbol text   = 1                    ' used to set up for cmd/txt byte
  symbol enable = C.7                  ' LCD enable pin connected to C.7
  symbol RegSel = C.4                  ' LCD RegSel pin connected to C.4


' === Variables ===
  symbol  char = b0                    ' character to be sent to LCD
  symbol index = b1                    ' used as counter in For-Next loops

' Note:  The following are variables because their values can change
  symbol newB.4 = outpinC.3            ' replacement pin for B.4
  symbol newB.6 = outpinA.0            ' replacement pin for B.6 (hserin)


' === Directives ===
  #com 4                               ' specify download port
  #picaxe 20X2                         ' specify processor
  #no_data                             ' save time downloading
  #terminal off                        ' make sure terminal is off


' === Table ======================
  Table  0, ("The Evil Genius ")
  Table 16, ("Capstone Project")
  Table 32, ("A Multi-Function")
  Table 48, ("PeripheralDevice")


' ============================== Begin Main Program ==============================
  setfreq m16
  dirsB = %11101111                    ' all outputs, except B.4
  dirsC = %10111111                    ' all outputs, except C.6


' === Initialize the LCD ===
  pause 400                            ' pause 200mS for LCD initialization
  char = 56                            ' setup: 8 bits, 2 lines, 5X7 dots
  gosub OutCmd                         ' send instruction to LCD
  char = 12                            ' display on, cursor off
  gosub OutCmd                         ' send instruction to LCD
```

LISTING 14-2 (continued)

```
' === Main Program Loop - Send data to the LCD ===
do
  char = 1                              ' clear display & go home
  gosub OutCmd                          ' send instruction to LCD
  sound C.1,(40,100)

  for index = 0 to 15
    readtable index, char               ' send line one to LCD
    gosub OutTxt
  next index

  char = 192                            ' move cursor to start of line two
  gosub OutCmd                          ' send instruction to LCD

  for index = 16 to 31                  ' send line two to LCD
    readtable index, char
    gosub OutTxt
  next index

  wait 4
  char = 1                              ' clear display & go home
  gosub OutCmd                          ' send instruction to LCD
  sound C.1,(60,100)

  for index = 32 to 47
    readtable index, char               ' send line one to LCD
    gosub OutTxt
  next index

  char = 192                            ' move cursor to start of line two
  gosub OutCmd                          ' send instruction to LCD

  for index = 48 to 63                  ' send line two to LCD
    readtable index, char
    gosub OutTxt
  next index

  wait 4
loop


' ====================== End Main Program - Subroutines Follow ====================


OutCmd:
  low RegSel                            ' set up for command byte
  goto Doit                             ' do it
```

(continued)

LISTING 14-2 (continued)

```
OutTxt:
  high RegSel                          ' set up for text byte
Doit:
  outpinsB = char                      ' load byte onto outpinsB
  newB.4 = bit4                        ' substitute outpinC.3 for B.4
  newB.6 = bit6                        ' substitute outpinA.0 for B.6
  pulsout enable,2                     ' send data
  return
```

step of calibrating the keypad's ADC values. The TestMPD.bas software that follows (Listing 14-3) should look familiar; it's a combination of keypad and LCD routines that we have covered in detail in the earlier chapters. However, before you download it to your MPD, you will need to follow the same procedures we used in the "Experiment 2" section of Chapter 11 and replace the theoretical ADC midpoint value in each of the *case* statements in the *select case* command with the actual values your MPD hardware produces. To do so, use the list you jotted down earlier and

LISTING 14-3

```
' ================================== TestMPD.bas ==================================
' This program tests the Evil Genius Multifunction Peripheral Device.

' === Constants ===
  symbol cmnd   = 0                    ' used to set up for cmd/txt byte
  symbol text   = 1                    ' used to set up for cmd/txt byte
  symbol enable = C.7                  ' LCD enable pin
  symbol RegSel = C.4                  ' LCD RegSel pin

' === Variables ===
  symbol char   = b0                   ' character to be sent to LCD
  symbol junk   = b1                   ' throwaway character
  symbol key    = w1                   ' ADC key value
  symbol index  = b4                   ' used in for/next loop

' Note:  The following are variables because their values can change
  symbol newB.4 = outpinC.3            ' replacement pin for B.4
  symbol newB.6 = outpinA.0            ' replacement pin for B.6 (hserin)

' === Directives ===
  #com 4                               ' specify COM port
  #picaxe 20X2                         ' specify compiler mode
  #no_data                             ' save time downloading
  #no_table                            ' save time downloading
  #terminal off                        ' make sure terminal is off
```

LISTING 14-3 (continued)

```
' ============================== Begin Main Program ==============================
  setfreq m16
  dirsA = %11111111                      ' configure pinA.0 as output
  dirsB = %11101111                      ' configure pinB.4 as input
  dirsC = %10111111                      ' configure pinC.6 as input
  adcsetup = %0000000001000000           ' enable ADC6 (see adcsetup in Manual)

' === Initialize the LCD ===
  pause 400                              ' pause 200 mS for LCD initialization
  char = 56                              ' setup for 8-bits,2 lines & 5X8 dots
  low RegSel                             ' set up for command byte
  gosub OutByte2                         ' send instruction to LCD
  char = 12                              ' display on, cursor off
  gosub OutByte                          ' send instruction to LCD
  char =  1                              ' clear display and go home
  gosub OutByte                          ' send instruction to LCD
  wait 1

' === Main Program Loop - Get keypress and display it on LCD ===
do
wait_for_keypress:
  readadc 6, junk
  if junk < 5 then wait_for_keypress
  pause 50                               ' debounce key
  readadc10 6, key                       ' get ADC value

wait_for_release:
  readadc 6, junk
  if junk > 5 then wait_for_release

  select case key                        ' decode key
    case < 364 : char = 65               ' A
    case < 379 : char = 82               ' R
    case < 395 : char = 68               ' D
    case < 422 : char = 76               ' L
    case < 452 : char = 48               ' 0
    case < 476 : char = 57               ' 9
    case < 502 : char = 85               ' U
    case < 542 : char = 66               ' B
    case < 588 : char = 56               ' 8
    case < 629 : char = 55               ' 7
    case < 674 : char = 54               ' 6
    case < 733 : char = 53               ' 5
    case < 804 : char = 52               ' 4
    case < 884 : char = 51               ' 3
    case < 976 : char = 50               ' 2
    else       : char = 49               ' 1
  end select
```

(continued)

LISTING 14-3 (continued)

```
  if char = "B" then                    ' if "B" then
    char = 1                            ' clear LCD display
    sound C.1,(20,150)
    gosub OutByte
  else                                  ' else
    sound C.1,(50,50)
    gosub OutByte                       ' display char
  endif
loop


' ==================== End Main Program - Subroutines Follow ====================

OutByte:
  select case char
    case 32 to 127                      ' set up for text
      high RegSel
    else                                ' set up for command
      low RegSel
  endselect

Outbyte2:
  outpinsB = char                       ' load character onto outpinsB
  newB.4 = bit4                         ' substitute outpinC.3 for pinB.4
  newB.6 = bit6                         ' substitute outpinA.0 for pinB.6
  pulsout enable,2                      ' send data
  return
```

calculate the midpoint value between each pair of ACD readings. (If this isn't clear, you may need to re-read the "Experiment 2" section in Chapter 11.)

When you have updated the midpoint ADC values in TestMPD.bas, save it and download it to your MPD. Except for the "Back" key, which clears the display, each keypress should produce the corresponding character that the program has assigned to the key; if not, double-check the ADC midpoint values you entered into the program.

This completes the construction of our Evil Genius Multifunction Peripheral Device. In the next chapter, we'll explore the details of implementing an ADC interrupt routine, and then put the MPD to work by developing a major software application.

CHAPTER 15

Developing Software for the Evil Genius MPD

Now that we have completed construction and testing of our MPD, we're ready to develop some software for it. As I mentioned in Chapter 14, we're also going to modify our keypad input routine so that it's interrupt-based, which will actually simplify the programming involved and greatly improve the speed of the MPD's response to user input. However, in order to do so, we first need to discuss a little theory related to the hardware comparators built into all X2-class microprocessors. As usual, we'll focus on the 20X2 hardware, but most of what we cover will also be directly relevant to the 28X2 and the 40X2 processors as well.

Understanding the 20X2's Built-in Comparator Hardware

All X2-class processors include two hardware comparators (labeled C1 and C2), which can compare two analog voltages. Both voltages can be external (i.e., connected via an ADC pin), or one can be external and the other can be derived from the chip's internal voltage reference (IVR). Figure 15-1 presents the pin-out for the PICAXE-20X2, which includes the relevant pin assignments for the 20X2's comparators. As you can see, two pins (15 and 16) are assigned to comparator 2, but only one pin (14) is assigned to comparator 1. This means that for comparator 1, the C1+ voltage must be internally derived from the 20X2's IVR.

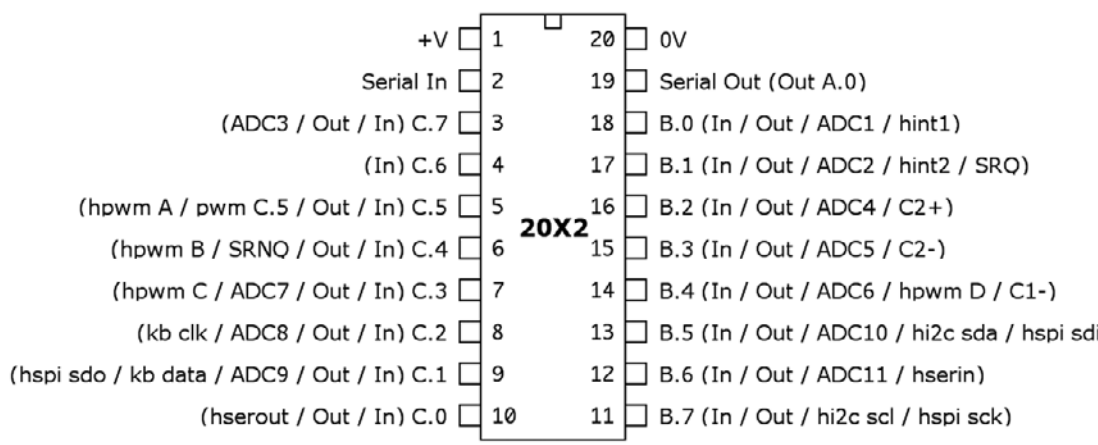
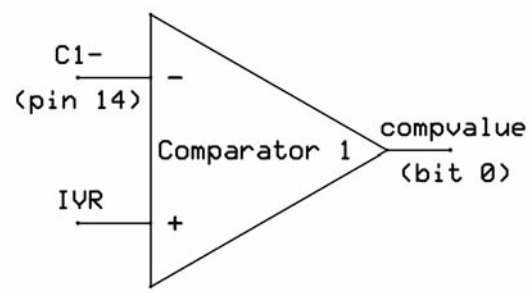


Figure 15-1 PICAXE-20X2 pin-out with comparator pin assignments

Consequently, comparator 1 is the ideal choice for the MPD because we're only interested in the level of the single ADC input from the keypad. Because ADC6 and C1 are both implemented on pin 14 of the 20X2, ADC6 is the obvious choice for the keypad input; that's the reason for the circuit changes that I mentioned in the previous chapter.

Let's take a closer look at comparator 1 on the 20X2; its schematic is presented in the following illustration. The IVR is connected to the positive input of the comparator, and our (external) keypad voltage is connected to the negative input, which means that comparator 1, by default, is an inverting comparator. In other words, whenever the external input is higher than the IVR input, the comparator's output goes low; whenever the external input is lower than the IVR input, the comparator's output is high. As we'll soon see, we can modify the default configuration.



PICAXE comparators can be set up in a variety of configurations that are controlled by the *compsetup* command. The complete syntax is *compsetup config, ivr*. *Config* is a constant or variable that specifies the configuration options, and *ivr* is a constant or variable that specifies the configuration of an internal voltage ladder that can be used to adjust the value of IVR. We're not going to use the internal voltage ladder, so we can just set *ivr* to 0, which disables it; I'll explain the reason shortly.

On the 20X2, *config* is a ten-bit configuration word that determines the exact mode of operation for each of the comparators, as shown in Table 15-1.

TABLE 15-1 20X2 Comparator Configuration Settings

Bit	Value	Comparator Settings
0	0	Comparator 1 is disabled.
	1	Comparator 1 is enabled.
1	0	Comparator 2 is disabled.
	1	Comparator 2 is enabled.
2	0	Comparator 1 output is not inverted.
	1	Comparator 1 output is inverted.
3	0	Comparator 2 output is not inverted.
	1	Comparator 2 output is inverted.
4	0	Change in comparator 1 does not cause change in *compflag*.
	1	Change in comparator 1 sets *compflag*.
5	0	Change in comparator 2 does not cause change in *compflag*.
	1	Change in comparator 2 sets *compflag.*
6	1	Bit 6 is not implemented; always set it to 1.
7	0	Comparator 2 Vin+ is ADC2.
	1	Comparator 2 Vin+ is from voltage divider/fixed ref.
8	0	Comparator 1 Vin+ is set from voltage divider.
	1	Comparator 1 Vin+ is from fixed 1.024V reference.
9	0	Comparator 2 Vin+ is set from voltage divider.
	1	Comparator 2 Vin+ is from fixed 1.024V reference.

Let's consider each bit separately to see how we want to configure the *compsetup* command.

- **Bit 0 = 1:** Enable comparator 1.
- **Bit 1 = 0:** Disable comparator 2.

- **Bit 2 = 1:** We'll invert the inverted output, so it's no longer inverting!
- **Bit 3 = 0:** Doesn't really matter because we have disabled comparator 2.
- **Bit 4 = 0:** Just for now—later we'll discuss and change this setting.
- **Bit 5 = 0:** Doesn't really matter because we have disabled comparator 2.
- **Bit 6 = 1:** No choice because bit 6 isn't implemented.
- **Bit 7 = 0:** Doesn't really matter because we have disabled comparator 2.
- **Bit 8 = 1:** We'll discuss this one next!
- **Bit 9 = 0:** Doesn't really matter because we have disabled comparator 2.

The setting for bit 8 requires a brief explanation. The IVR on the 20X2 can be derived from two different sources: the fixed 1.024V reference voltage or any step of an internal 32-step resistor ladder that I mentioned earlier. Configuring the IVR from the resistor ladder is a little complicated; fortunately for us, it's not necessary because the fixed 1.024V IVR will work just fine. To see why this is the case, we need to refer back to Chapter 6, where we first discussed ADC inputs. For convenience, Figure 15-2 reproduces Figure 6-1 and includes the basic voltage divider formula.

As you can see in Figure 15-2, the greater the resistance above the V_{out} line, the lower the output voltage (because R_{total} is larger). If you look back at our original keypad circuit in Chapter 11, you'll see that the greatest resistance above the ADC output line occurs when the 15k and 3.3k resistors are connected in series to +5V. When this occurs, the formula in Figure 15-2 simplifies as follows:

$$V_{out} = \frac{R_b}{R_{total}} * V_{in} = \frac{10k}{10k + 15k + 3.3k} * 5V$$

$$= \frac{10k}{28.3k} * 5V = \frac{50V}{28.3} = 1.767V$$

Therefore, the lowest voltage produced by any keypress is 1.767V. In other words, every keypress produces a voltage that is greater than the 20X2 IVR of 1.024V. On the other hand, when no key is pressed, the 10k resistor in the circuit ties the ADC line to ground. As a result, the comparator 1 output will change whenever any key is pressed. This output is always available to a running program by accessing the built-in *compvalue* variable; the value of the comparator 1 output is bit 0 of this variable, and the output of comparator 2 is bit 1. Since we're disabling comparator 2, bit 1 of *compvalue* will always be 0. Therefore, for our purposes, whenever the comparator 1 output is high, *compvalue* will equal 1 (because we "inverted the inversion"), and whenever the comparator 1 output is low, *compvalue* will equal 0.

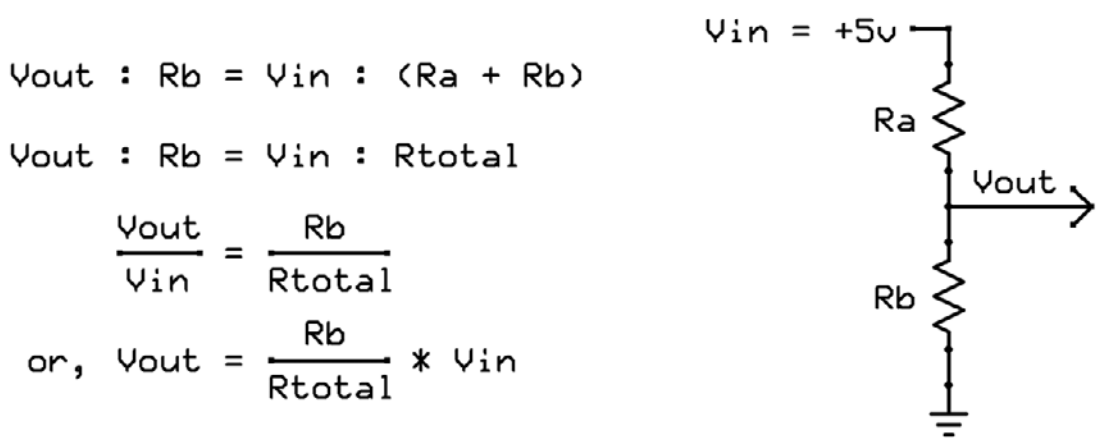


Figure 15-2 Basic voltage divider circuit and formula

Testing Our Comparator 1 Configuration

In order to test the comparator configuration, simply attach your MPD to a powered breadboard and add an LED (and current-limiting resistor) from the C.5 line (pin 3 of the MPD's breadboard connector) to Ground, as shown in Figure 15-3. In the photo, you can also see that I have added labels below the MPD's keypad to remind myself of the connector's pin-out. (I got tired of having to find the schematic every time I wanted to connect something!)

Figure 15-3 Breadboard circuit for testing the comparator 1 configuration

The software for our little test (Comparator1Demo.bas) is presented in Listing 15-1. The only part of it that I haven't explained is the *adcsetup* statement. Actually, we have discussed this before, but it's worth repeating. On the M2-class and X1-class processors, *adcsetup* isn't required; issuing a *readadc* or *readadc10* command automatically configures the specified pin as an ADC input. However, on X2-class processors, *adcsetup* is required whenever a pin is to be used as an ADC input or a comparator input. Therefore, we need to properly configure the ADC6 pin as shown in the Comparator1Demo.bas program. (See the *adcsetup* documentation in Part II of the manual for details.)

One final point before you download and test the program; *adcsetup* is technically a word-length variable so the equal sign is required in the assignment statement. On the other hand, *compsetup* is technically a command, so no assignment statement is needed; in fact, including an equal sign in a *compsetup* statement will produce a compiler error. Download Comparator1Demo.bas to your MPD. When no key is being pressed on the keypad, the LED should remain off; pressing any key should light the LED as long as the key is pressed.

"We Interrupt This Program to Bring You a Keypress!"

By now you're probably wondering why we're even discussing the 20X2 comparators. The answer is simple: the *setintflags* command (which we discussed in Chapter 13) includes a *compflag* bit (bit4) that enables us to configure an interrupt whenever a comparator output changes state. That means we can convert our keypad-decoding software to an *interrupt* subroutine that will make each keypress available to our program immediately.

LISTING 15-1

```
' ============================ Comparator1Demo.bas ============================
' Program demonstrates the use of comparator 1 on the EG MPD.


' === Constants ===
  symbol LED = C.5                    ' LED on C.5


' === Directives ===
  #com 4                              ' specify com port
  #picaxe 20X2                        ' specify compiler mode
  #no_data                            ' save time downloading
  #no_table                           ' save time downloading
  #terminal off                       ' make sure terminal is off


' ============================ Begin Main Program ============================
setfreq m16                           ' set frequency to 16MHz
dirsB = %11101111                     ' configure pinB.4 as input
dirsC = %10111111                     ' configure pinC.6 as input
pullup  %00000010                     ' enable internal pullup on C.6
adcsetup = %0000000001000000          ' see text
compsetup  %0101000101, 0             ' see text

do
  if compvalue = 1 then               ' any keypress is > 1.024v
    high LED
  else                                ' no keypress equals 0v
    low LED
  endif
loop
```

The following program (TestInterrupt.bas, shown in Listing 15-2) implements the necessary changes in our software. In the main portion of the program, the *setintflags %00010000,%00010000* command enables an interrupt each time a key is pressed. The *do…loop* simply toggles the LED on C.5 to keep busy while waiting for an interrupt to occur. In the *interrupt* subroutine, we need to temporarily disable the interrupts so that another one doesn't occur while we're processing the first one. We also need to disable the comparator so that the ADC6 function on the B.4 pin can be used to determine which keypress triggered the interrupt. The main part of the interrupt should be familiar. We simply decode the keypress, "beep," and send the key value to the terminal window for display. Just before exiting the *interrupt* subroutine, we need to clear the interrupt flag and re-enable both the comparator and the interrupt.

LISTING 15-2

```
' ============================== TestInterrupt.bas ==============================
' This program tests the ADC keypad interrupt on the Evil Genius MPD.


' === Constants ===
  symbol  horn = C.1                    ' speaker or piezo on C.1
  symbol   LED = C.5                    ' LED on C.5


' === Variables ===
  symbol  char = b0                     ' character to be sent to LCD
  symbol  junk = b1                     ' throwaway character
  symbol   key = w1                     ' ADC keypress value


' === Directives ===
  #com 4                                ' specify com port
  #picaxe 20X2                          ' specify compiler mode
  #no_data                              ' save time downloading
  #no_table                             ' save time downloading
  #terminal 19200                       ' open terminal window


' ============================ Begin Main Program ============================
  setfreq m16                           ' set frequency to 16MHz
  dirsB = %11101111                     ' configure pinB.4 as input
  dirsC = %10111111                     ' configure pinC.6 as input
  pullup  %00000010                     ' enable internal pullup on C.6
  adcsetup = %0100000001000000          ' see "adcsetup" in Manual
  compsetup  %0101010101, 0             ' see "compsetup" in Manual
  setintflags %00010000,%00010000       ' see "setintflags" in Manual

do
  toggle LED                            ' toggle LED slowly
  wait 8                                ' pretend to be busy
loop


' ================== End Main Program - Subroutines Follow ====================
interrupt:
  setintflags off                       ' disable comparator interrupt
  compsetup 0,0                         ' disable comp so ADC is available
  pause 10                              ' debounce the keypress
  readadc10 6, key                      ' get ADC value
wait_for_release:                       ' wait for key to be released
  readadc 6, junk
  if junk > 5 then wait_for_release
```

LISTING 15-2 (continued)

```
select case key                          ' decode key
  case < 364 : char = 65                 ' A
  case < 379 : char = 82                 ' R
  case < 395 : char = 68                 ' D
  case < 422 : char = 76                 ' L
  case < 452 : char = 48                 ' 0
  case < 476 : char = 57                 ' 9
  case < 502 : char = 85                 ' U
  case < 542 : char = 66                 ' B
  case < 588 : char = 56                 ' 8
  case < 629 : char = 55                 ' 7
  case < 674 : char = 54                 ' 6
  case < 733 : char = 53                 ' 5
  case < 804 : char = 52                 ' 4
  case < 884 : char = 51                 ' 3
  case < 976 : char = 50                 ' 2
  else       : char = 49                 ' 1
end select

sertxd ("key = ",char,cr,lf)             ' display keypress on terminal
sound horn,(50,20)                       ' beep
compsetup  %0101010001, 0                ' re-enable comparator 1
compflag = 0                             ' clear the interrupt flag
setintflags %00010000,%00010000          ' re-enable comparator interrupt
return
```

The slow rate at which the LED toggles enables the program to demonstrate two important characteristics of interrupts. If you press a key shortly after the LED changes state, you'll notice that it again changes state the instant the key is released. From this program behavior, you can deduce both of the characteristics I have in mind. First, the interrupt condition (C1 output change) is checked for continuously during the *wait* statement (this is also true for *pause* statements). Second, when the program returns from the interrupt, it does not complete the *wait* (or *pause*) statement; it moves on to the next instruction in the program.

Project 15
A Simple MPD Operating System

Now that we have an understanding of how a comparator *interrupt* subroutine functions on the 20X2, we're ready to tackle our software project: a simple MPD operating system (MPDOS) that includes the following features:

- Completely "open-source" code that can be easily modified by the user.
- Approximately 80 percent of 20X2 memory remains available for user-developed programs.
- All user-developed programs have access to the interrupt-driven keypad input.

- Built-in "prompting" characters for full arrow key functionality.
- Menu-driven user interface for selecting, running, and inputting data to programs.
- Custom lowercase LCD characters with true descenders.

The MPDOS "prompting" feature requires an explanation. As you know, our keypad includes four arrow keys for use with the LCD display; in MPDOS, these keys are intended to implement the following functions:

- **Up arrow:** Scroll line two of the display up one line.
- **Down arrow:** Scroll line two of the display down one line.
- **Left arrow:** Move left among the choices listed on a line (not yet implemented).
- **Right arrow:** Move right among the choices listed on a line (not yet implemented).

In addition, the last two character positions on a line can be reserved to display "prompts" to the user; position 15 is for the "left and/or right arrow" prompts, and position 16 is for the "up and/or down arrow" prompts. Three built-in characters can be displayed in the "left and/or right" position: a left-arrow, a right-arrow, and a bidirectional "left and right" arrow. Similarly, three additional built-in characters can be displayed in the "up and/or down" position: an up arrow, a down arrow, and a bidirectional "up and down" arrow. The purpose of the "prompt" characters is to remind the user which arrow keys are currently functional—this will become clear later when you actually run the MPDOS software.

Two of the six arrow characters are already built into the LCD display as characters 126 and 127. The remaining four characters are defined as custom characters and loaded into the LCD using data statements in the MPDOS software. As you may remember, there are eight locations in the LCD for custom characters and we have been using five of them for lowercase letters with true descenders. Because we need four locations for the four arrow characters, one of the custom lowercase letters has to be sacrificed; I chose to eliminate "y" because it required its own *case* statement in the software and the other four characters share the same *case* statement. Of course, you can change that arrangement however you want. Also, I wanted to assign values to the four "arrow" characters that are in the normal text range (32 to 127) because that simplified the LCD routine. As a result, I had to choose four ASCII characters that I was also willing to sacrifice. I chose the following four, but you can certainly choose differently if you prefer:

- **ASCII 92:** This is the yen symbol on my LCD; I re-assigned it to the up arrow.
- **ASCII 94:** This is the ASCII "^ symbol; I re-assigned it to the down arrow.
- **ASCII 123:** This is the ASCII "{ symbol; I re-assigned it to the "up and down" arrow.
- **ASCII 125:** This is the ASCII "} symbol; I re-assigned it to the "left and right" arrow.

The MPDOS.bas software is too long to include here; download it from my website and print out a copy for reference. When you first see the program, it may seem a little daunting to say the least. At almost 400 program lines, it's by far the longest piece of software we have discussed. However, it's really not as difficult as it seems; much of it is composed of routines that we have already discussed. The four "programs" in MPDOS (BeamMeUp, Cloaking, DeathRay, and Doomsday) are all trivial; they are merely included to demonstrate how the "prompt" characters can be used in a program. The program is also thoroughly commented, so it should be reasonably easy to understand.

Before you test MPDOS.bas, I should remind you of the functions of the "Accept" and "Back" keys. "Accept" is like "Enter" on your PC; in MPDOS, pressing "A" will run the program currently displayed on the LCD. Pressing "Back" takes you back one level in the menu system; for example, pressing "B" when a program is running returns you to the program selection menu. Watch the prompts at the right end of line two as you test the program; that should clarify how they function.

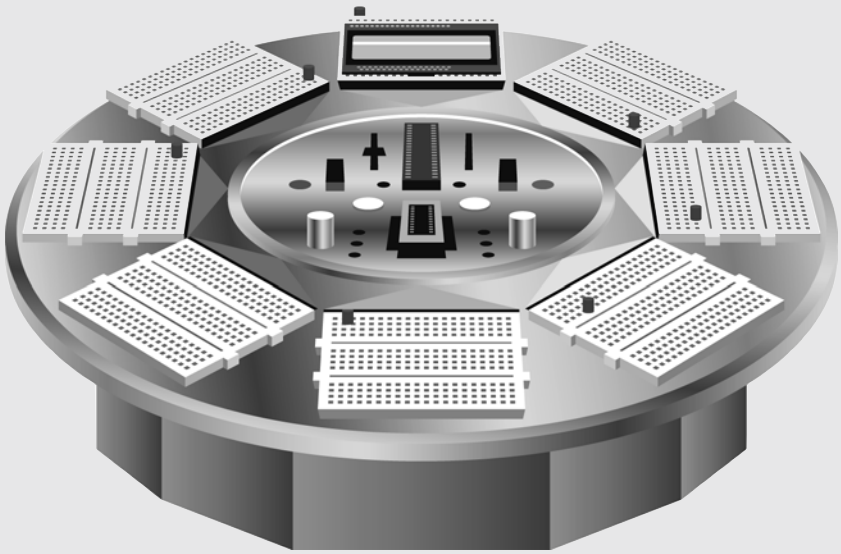
Developing your own programs for use with the MPD is a simple process. All you need to do is replace one of the trivial program names in the table with the name of your program and remove the code from within the corresponding subroutine. (Of course, you can also add lines to the table and subroutines if you develop more than four programs.) I already have two programs in mind: an I/O terminal for use with my projects, and a code timer to help me determine which version of a software snippet runs faster.

If you develop an interesting application for the MPD and MPDOS.bas, I would love to hear about it; you can reach me by e-mail at Ron@JRHackett.net. If you're willing to share your program (and if I get sufficient responses), I'll set up a page on my site dedicated to the MPD so that we all can benefit from each other's work.

This page intentionally left blank

PART THREE

Octavius: An Advanced Robotics Experimentation Platform



This page intentionally left blank

CHAPTER 16

Birthing Octavius

Over the years, I have designed and built many small robots. At first, I tackled a couple of kits that simply needed to be assembled (e.g., the robot on the left in Figure 16-1). Robot kits can provide a great learning experience, but they tend to be somewhat limited because they are usually difficult or impossible to modify once they're completed. To overcome this limitation, I soon decided to design my own robots that included what I thought was ample breadboard space so that the circuitry could be easily modified to incorporate new features as they occurred to me (e.g., the robot on the right in Figure 16-1). This type of robot was a great improvement over a kit, but I was surprised to discover how quickly I ran out of space on the robot's breadboard development area.

About a year ago, I began work on "Octavius," a PICAXE-powered robot whose goal in life is to grow up to become a sophisticated robotics experimentation platform. I specifically designed

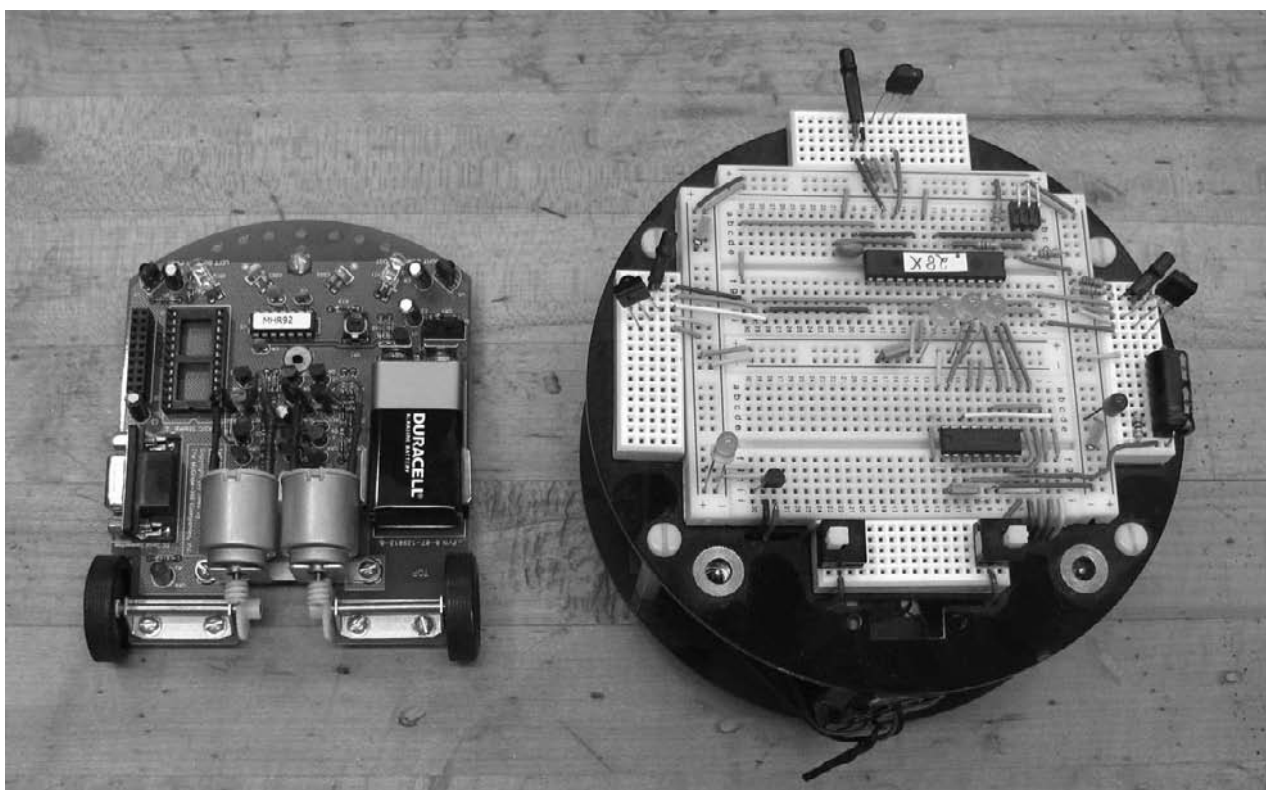


Figure 16-1 First- and second-generation robots

Octavius' main logic board so that he has ample breadboard space in which to implement and test as many features as I want. The logic board, which is shown in Figure 16-2, has a unique octagonal shape that is designed to bring the simplicity and flexibility of breadboard-based project development to the world of robotics.

Eight 30-pin female headers are positioned around Octavius' perimeter. With a 400-point breadboard installed next to each of the headers, Octavius is able to communicate with 75 square inches of development space spread out among his eight breadboard units (see Figure 16-3). The extra PC board and messy cables in the middle of the photo are a temporary solution to a motor driver "meltdown" problem that I encountered early in Octavius' development. I'll explain the details when we discuss Octavius' motor-driver circuitry. In the next chapter, we'll develop a motor-control project that is a major improvement over Octavius' original circuitry and can also be used to drive your next robotic creation.

Each of Octavius' peripheral breadboards has access to a total of 30 signals, including 26 I/O lines from his PICAXE-40X2 CPU as well as the necessary power and ground lines. Routing all 30 lines to each of the eight breadboards was the most challenging aspect of Octavius' design, but it's also what gives him his power and flexibility. In addition, it gives us the opportunity to extend our "multiprocessor" approach in Part Two to the world of robotics. The circuitry on any (or all) of Octavius' eight breadboards can easily include a peripheral processor. In fact, it would even be possible to include more than one processor on a breadboard if you wanted to do so. Each of the peripheral processors can independently carry out its assigned task and easily communicate with Octavius' CPU. As a result, Octavius' power and

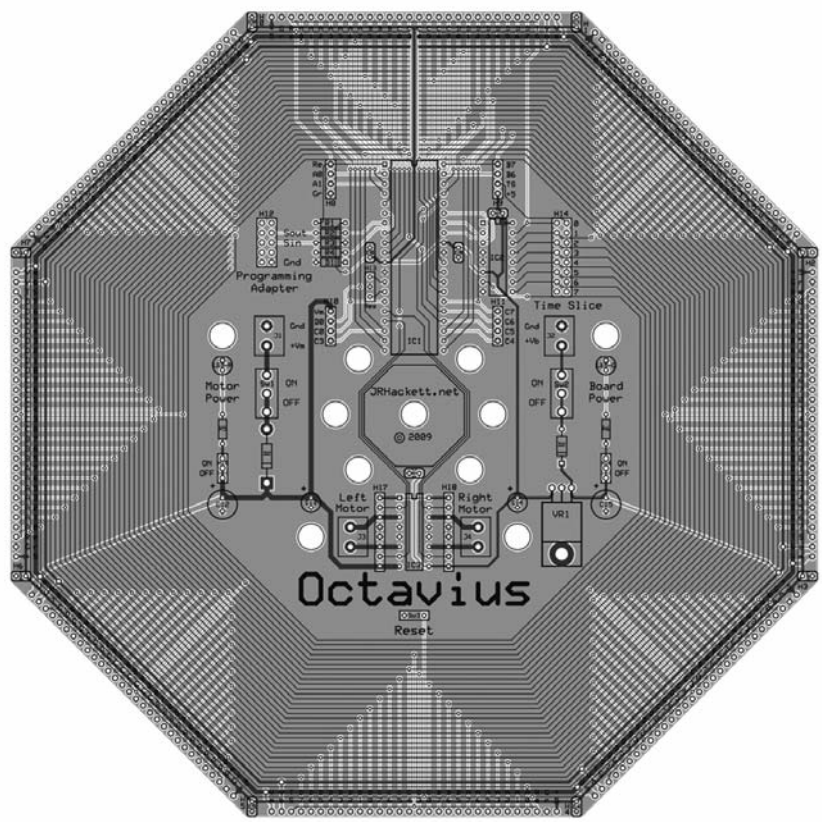


Figure 16-2 Octavius' main logic board

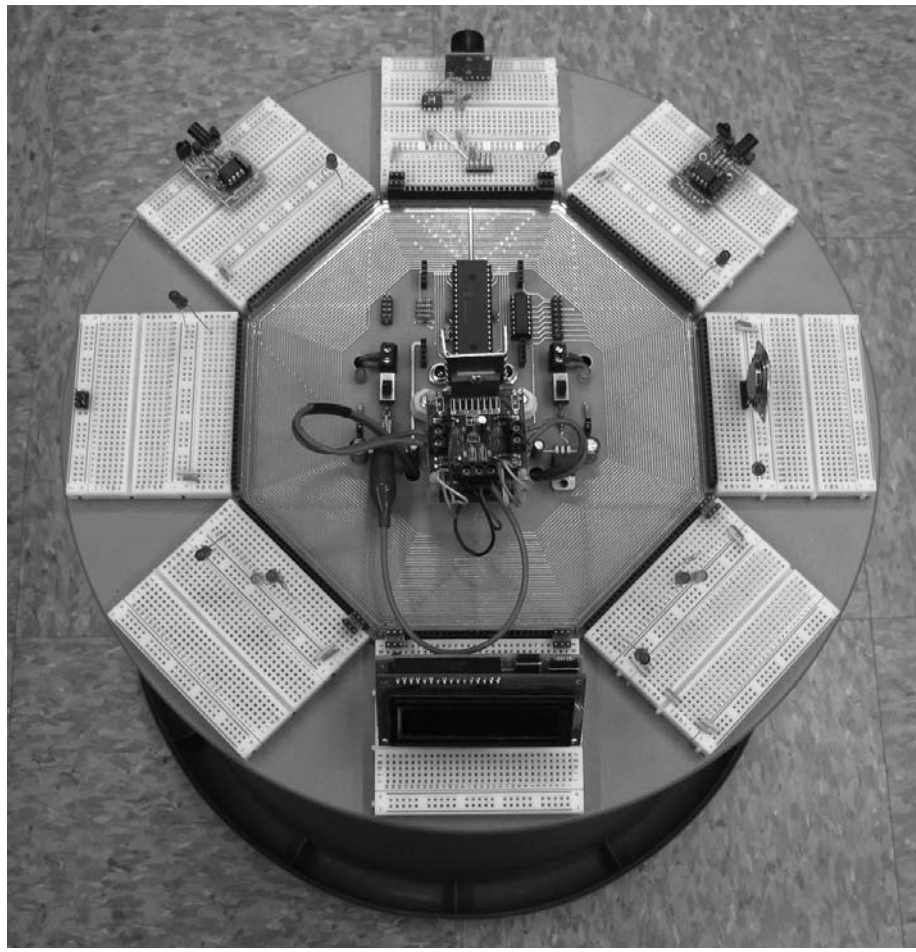


Figure 16-3 Octavius—an advanced robotics experimentation platform

flexibility are only limited by our imaginations and programming prowess!

There is one aspect that clearly differentiates Octavius from all the projects we have completed thus far in this book. Up to this point, every project was fully designed and tested before I began writing about it. In contrast, Octavius is a work in progress. If you look closely at Figure 16-3, you'll see that his main logic board is what's called a prototype PC board. For one thing, there's no silk screen (the handy white labeling on a PC board). There is also no solder mask (the protective coating that covers everything except the soldering points), although that's probably not clearly evident in the photo. I decided to approach the Octavius project in this manner so that his final design can benefit from the discoveries I make during the time I spend experimenting with and writing about Octavius.

My main purpose for including the chapters and projects in Part Three is to present what I have learned so far in my work with Octavius, especially the mistakes I have made along the way. That way, when you're ready to develop your own robotic creation, you'll have some idea of what works and what doesn't. In this chapter, I'll describe how I implemented Octavius' "body" and power system, and focus on some of the things you can do to simplify the process of designing and building a robot "body." In Chapter 17, we'll focus on constructing a motor controller circuit that's capable of delivering a considerable amount of power to your robot's motors without risking a motor controller "meltdown." In the final chapter,

we'll turn our attention to the implementation of a powerful sensory system that can be used by any robot.

If you find the material presented in Part Three helpful in the design and development of your own robot creation, I would love to hear about it. You can reach me at Ron@JRHackett.net. Also, if you are willing to share a photo of your creation, I would like to include it in a robot "pin-up" gallery on my website.

Understanding Octavius

Figure 16-4 presents the schematic for Octavius. Since it's fairly complicated, let's break it down into the following sections for our discussion:

- PC Board Power Supply
- Motor Power Supply
- Programming Adapter
- Peripheral Breadboard Headers
- Time-Slice Generator
- Motor Controller

PC Board Power Supply

Octavius' PC board power supply is a standard +5V regulated supply. It includes a charging connector for the PC board battery (a 12V, 1.3Ah, lead-acid battery). The connector is identical to the one we used in our breadboard power supply project in Chapter 3; it accepts a power plug that has a 2.1-mm ID (inner diameter) and a 5.5-mm OD (outer diameter). As I mentioned in Chapter 3, that size plug is commonly available and easily obtained from surplus vendors or obsolete answering machines, modems, etc. You'll also need a suitable lead-acid battery charger; ordinary "wall-wart" supplies will not work, and may damage the battery and even present a fire hazard. Finally, the power indicator LED can be disconnected via header H3 if you prefer to conserve battery power.

Motor Power Supply

The motor power supply is even simpler because there's no regulation involved. It also includes its own charging connector for the motor battery (a 12V, 8.0Ah, lead-acid battery), as well as a header to disconnect the power indicator LED, if desired.

Programming Adapter

On Octavius' prototype PC board, the programming adapter is connected via a 5 × 2 male header, which is an interface I have used in most of my projects for the past few years. However, as we have already discussed, the serial port is just about dead by now, so I intend to upgrade Octavius to the standard audio connector that accepts the AXE027 USB programming cable.

Peripheral Breadboard Headers

Each of the eight peripheral breadboards abuts its own 30 × 1 female header, allowing for simple jumper wire connections between the 40X2's I/O lines and the breadboard. If you look back at the photo in Figure 16-3, you will see that each breadboard unit is actually comprised of one and a half breadboards. This arrangement allows for considerable flexibility in the breadboard wiring because signals can be easily routed around processors and other components on the board. Breadboard 0 is at the top of the PC board (directly above pin 1 of the 40X2), and the breadboards are numbered consecutively (0 through 7) in a clockwise direction. The order of the 30 connections shown at the right side of the schematic is the same for all eight breadboards.

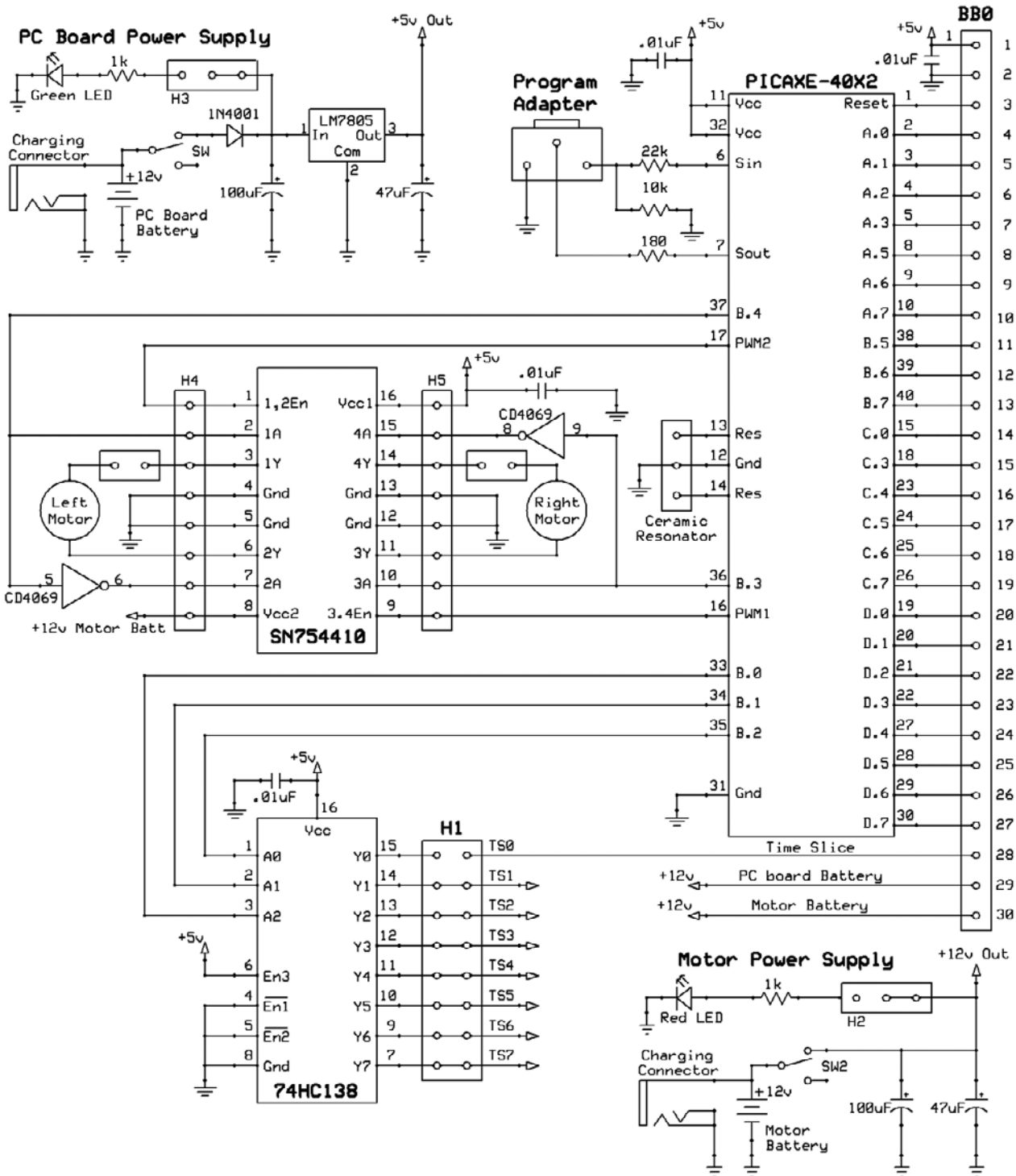


Figure 16-4 Octavius' schematic

Time-Slice Generator

The 74HC138 is a complementary metal-oxide-silicon (CMOS) three-line to eight-line inverting demultiplexer. The three-bit address placed on pins A0, A1, and A2 of the 74HC138 controls which of its eight outputs is pulled low (all the other outputs remain high). The purpose of this circuitry is to enable Octavius to coordinate all communication with his numerous sensory/motor peripheral processors and thereby simplify the software necessary for monitoring his various processes and successfully navigating his way through his environment. My original intention was to implement a main program loop whereby Octavius would sequentially signal each peripheral processor in turn to determine whether the processor had updated sensory information to report. However, from working with the ultrasonic ranging system we're going to implement in Chapter 18, I soon realized the power and simplicity of a sensor that provides a continuously updated reading on a single output.

For example, suppose you have an 08M peripheral processor that is monitoring the light level directly in front of your robot. It would be a simple matter for the 08M to maintain a pulse-width modulation (PWM) output signal and update the duty cycle to reflect the real-time light level it "sees." Whenever the master processor wanted to check the current light-level reading, all it would need to do is take a *pulsin* measurement on the appropriate input pin; no coordinated two-way communication would be required. I'm just beginning to explore the possibilities of this approach, so it's too soon to know if it's simpler than Octavius' current time-slice arrangement. If it is, I'll be able to simplify Octavius' circuitry.

Motor Controller

Octavius' prototype PC board includes an SN754410 motor controller chip, which is a quadruple half-H driver that is capable of controlling two DC motors with voltages between 4.5V and 36V, with a maximum current-draw of 1 Amp each. Headers H4 and H5 are included on the prototype PC board so that a second 754410 can be "piggy-backed" to double the current-handling capabilities of the motor control circuit. This turned out to be a good decision because when I initially designed the circuit, I forgot to include the necessary inverters. Fortunately, the headers made it easy to redesign the motor control circuit (correctly, this time!), build it on a small stripboard, move the 754410 from the PC board to the stripboard, and plug it into the headers.

Even though I recovered from that little faux pas, I still had major problems with the motor control circuit. The motors that I'm using each draw about 200mA with no load; under load, that figure increases to about 600mA. What I didn't realize is that a full-stall condition (for example, when Octavius runs into a wall and the motors continue to be powered but the wheels are not turning at all) can drastically increase the current draw to a value greater than even two 754410 chips can handle. The result was a loud crack and a lot of "magic smoke," destroying the motor driver chips in the process. Even worse, I soon discovered that one of the PWM pins on my 40X2 was also burned out.

The result of this little tragedy was that I decided to completely redesign the motor controller circuit. I'm upgrading Octavius to an L298 motor driver, which is capable of handling 2 Amps per motor continuously, and 3 Amps per motor for momentary surges. More importantly, the L298 includes circuitry that will enable Octavius (or any robot) to monitor his motors'

real-time current levels and take corrective action if they become dangerously high. We'll get into the details of the "new and improved" motor control circuit in Chapter 17.

Project 16
Building Octavius

In this section, we're going to take a detailed look at Octavius' "brain," as well as his body. Most of the following information will be helpful for any robotics project you undertake.

Octavius' Printed Circuit Board

Figure 16-2 (presented earlier) showed the printed circuit board layout for Octavius' "brain." The circuit itself only occupies about 25 percent of the PC board area; the majority of space is taken up by the routing of the 30 signals to all eight peripheral breadboards. I realize that many readers of this book may not be interested in constructing a robot as complex as Octavius, so I'm working to simplify his circuitry and decrease the size of his main PC board. The final version will definitely be simpler, smaller, and less expensive to build; I'll post updated information on my website. The following discussion will focus on a couple features of the Octavius PC board that can be easily adapted to a robot of your own design.

The four 4-pin female headers (H8 through H11) that flank the 40X2 processor are included to provide one possible means of expanding Octavius' capabilities. For example, suppose you want Octavius to be able to rapidly find the clearest path in his immediate environment. One possibility would be to add an ultrasonic ranging system to each of his peripheral breadboards. However, the cost of eight such systems could well be prohibitive for most amateur roboticists. A considerably less expensive (and more elegant) approach to the problem would be to design a scanning tower (driven by a stepper motor) that could rapidly measure the open distance in a variety of directions. All you would need to do is design a stripboard circuit for your scanning tower that includes downward-pointing male headers that are spaced appropriately to mate with H8 through H11. That way, one ultrasonic sensor could quickly get the job done.

IC2 is the 74HC138 three-line to eight-line inverting demultiplexer that implements Octavius' time-slice capability. Header H14 provides for the possibility of rerouting any one of the TS signals to an expansion board that is mounted in headers H8 through H11 by simply installing a two-pin shunt on the two pins of the time-slice signal that you want to use.

Motor Controller Options

As we have already discussed, Octavius' original SN754410 motor controller was not able to handle the power requirements of the motors that I am using. In Chapter 17, we'll discuss this issue in more detail and construct a motor controller based on the more powerful L298 dual full-bridge driver chip, which is more than adequate for the task.

Additional Modifications

There will be a couple of additional minor modifications on the final Octavius PC board. First, header H12 is currently used for the PICAXE programming connection. However, by now, the vast majority of PC users no longer have access to the "old-fashioned" serial port with which header H12 connected. Consequently, the final PC board will include the standard PICAXE audio connector for use with the AXE027 USB programming cable. If you happen to be one of the "serial port holdouts," it's a simple matter to make an audio cable to DB-9 adapter.

Also, I have found the screw-type connectors that I am currently using for the power and motor connectors (J1 through J4) to be problematic; they are a nuisance whenever Octavius needs to be disassembled and re-assembled, and it's far too easy to accidentally reverse the motor connections in the process. Therefore, I intend to eliminate them entirely and substitute direct solder connections and polarized connectors on the necessary wires.

Designing a Body

In this section, we'll discuss some of the factors that need to be taken into account when you are designing a body for Octavius (or any robot). I'll tell you what I did, including the mistakes I made along the way. Even if your robot's body turns out to be drastically different from the one I constructed, I think you'll find a fair amount of helpful information in this section.

Figure 16-5 is a photo of Octavius on my "operating table" just after completion of a minor modification to his body. Let's begin by discussing what didn't work so you can avoid making the same mistakes I did. When I first built Octavius, I didn't include the round, raised platform you can see about ten inches above the lower platform. In his first incarnation, I had attached Octavius' outer shell (actually, a large, plastic flower pot!) directly to the bottom platform with some fairly complicated hardware that allowed me to adjust the levelness of his body. This approach turned out to be much more difficult than I anticipated; even worse, whenever I needed to remove the shell for any reason, it was a time-consuming process.

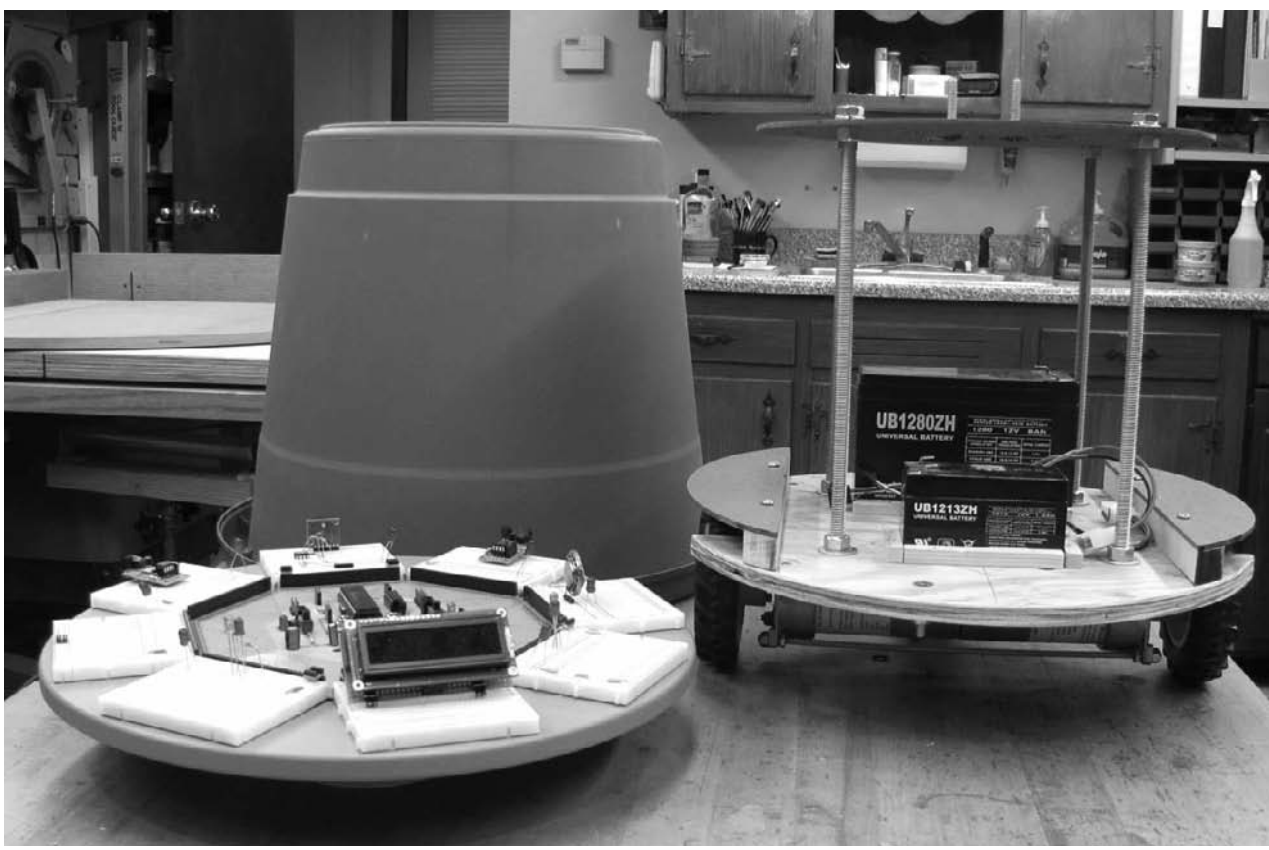


Figure 16-5 Octavius on the "operating table"

My second design is simpler and makes it much easier to access Octavius' innards whenever the need arises. The top platform (1/8-inch fiberboard) is supported by four pieces of 3/8-inch threaded rod, but I'm sure 1/4-inch would work just as well. It was tedious to initially level the top platform at exactly the right position to give Octavius the 1/4-inch ground clearance that I wanted, but it only had to be done once. Now, it's a quick and simple process to remove the shell and place it back at exactly the same position. The two bolts that you can see jutting up from the middle of the top platform fit through corresponding holes in the flowerpot and the round base that I used to mount the PC board and the breadboards. When everything is in position, two nylon nuts are all it takes to secure everything in place, and disassembly is just as easy.

Four sets of cables connect the two batteries and the two motors to the PC board. I used polarized power connectors on each cable to make it easier to disassemble everything when necessary. Be sure to clearly label each connector, especially if they are all identical. Also, make sure the cables are long enough to allow the shell to be lifted high enough to access the connectors to separate them. In Figure 16-5, you can see two small "fenders" above the protruding tops of Octavius' wheels. They turned out to be a necessary addition in order to protect the excess cable length from getting tangled in the wheels as Octavius was roaming about. On the next base that I construct, I'll make sure the wheels are entirely below the lower platform, which will eliminate the need for the protective fenders.

Figure 16-6 shows the details of Octavius' motor mountings. As you can see, they just barely fit in the circumference allowed by his flowerpot shell. The extra rectangular piece of plywood at the top of the photo is there because I originally

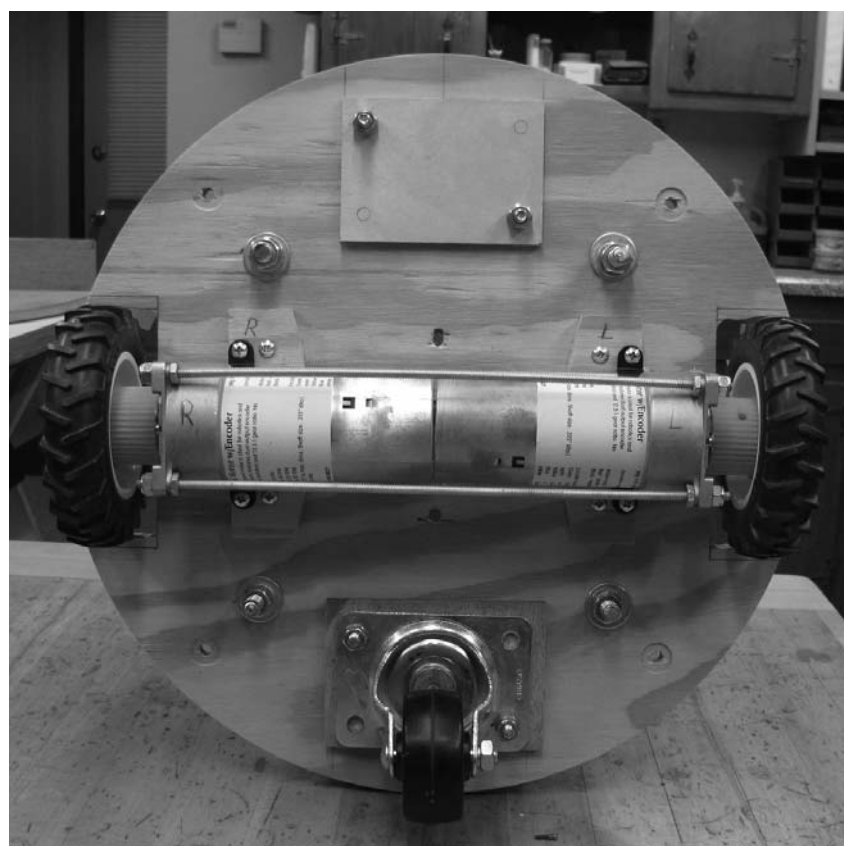


Figure 16-6 Octavius' motor mounts

had a second caster wheel installed. However, that turned out to be unnecessary as long as the bulk of Octavius' weight (i.e., his motor battery) is positioned directly above the single rear caster.

When you're ready to design your own robot, I would suggest that you begin with the motors and that you test them thoroughly before building the complete chassis. Make sure that they are capable of powering your robot's approximate weight, and also test their maximum current draw under full-stall conditions. (If I had taken my own advice, I probably wouldn't have destroyed Octavius' original motor controller circuitry and damaged two PICAXE-40X2 processors!) If you need more information on robot motors and drive trains, check out *Building Robot Drive Trains* by Dennis Clark and Michael Owings (McGraw-Hill, 2003).

I think I have included sufficient information on the construction of Octavius' body to enable you to design and construct a robot of your own. In the next chapter, we're going to cover the details of designing a motor control system that's powerful enough for a robot of Octavius' size and weight, and simple enough to be used with smaller robots as well. That way, if you would rather begin by designing a smaller and simpler robot, you could use our 20X2 master processor as its "brain," build a simple one-level platform, and add the motor controller we are about to discuss and construct. If you decide to take this approach, you will need to make one small sacrifice. The L298 chip requires two PWM signals—one for each motor; however, it's also possible to drive both L298 PWM input lines from the single PWM output on the 20X2. Your robot will still be able to move forward and backward at varying speeds, as well as turn in any direction by rotating in place. Making a gradual turn by moving along a curved path is the only ability that is sacrificed by this approach. If you don't want to lose that capability, you could switch to a 20M2 processor (which has four PWM outputs) or a 28X2, which has two PWM outputs and four additional I/O pins.

CHAPTER 17

Driving Octavius

When Octavius experienced his unfortunate motor controller meltdown that I described in the previous chapter, I knew his SN754410 motor controller chip wasn't up to the task of driving him around his subterranean home, so I decided to upgrade him to a more powerful propulsion system. I chose the L298 motor controller chip for three reasons. First, the L298 is capable of providing 2A per motor continuously and 3A per motor for momentary surges, so it's more than powerful enough for Octavius. Also, its I/O interface is identical to that of the SN754410, so there was no need to change Octavius' wiring or software. Most importantly, the L298 includes dual current-sensing capability, so Octavius (or any robot) can monitor the current draw of each of his motors in real time and, if necessary, implement corrective action to remedy the situation before any damage is done to his circuitry. For example, the most likely cause of excessive current draw would be that Octavius has run into an immovable object. If that were to happen, his wheels would either not be turning at all (a complete motor stall) or they would be drawing excessive current to overcome friction and spin in place. In either case, Octavius could detect the excessive motor current and immediately correct the situation by backing up a bit, turning, and going in a different direction.

In this chapter, we're going to take a brief look at the theory of an H-Bridge motor controller such as the L298 (or the SN754410, for that matter), followed by a more in-depth look at the L298. Finally, we'll construct a complete L298 dual-motor controller circuit and conduct experiments to demonstrate its major capabilities.

H-Bridge Motor Control Circuits

An H-Bridge motor controller is a simple device that enables us to easily control the direction in which a DC motor spins. Essentially, it consists of four "switches" that are arranged as shown in Figure 17-1. In the diagram on the left side of the figure, all four switches are in the open position so the motor is not powered. In the center diagram, switches 1 and 4 are closed so the DC current flows from left to right through the motor, causing it to spin in one direction. In the diagram on the right, switches 2 and 3 are closed so the DC current flows from right to left, causing the motor to spin in the opposite direction. Of course, in our simplified diagram, closing switches 1 and 2 (or 3 and 4) is an invitation to disaster because a direct short is produced between the V+ supply and Ground. Fortunately, all H-Bridge motor controllers include additional circuitry that prevents that possibility.

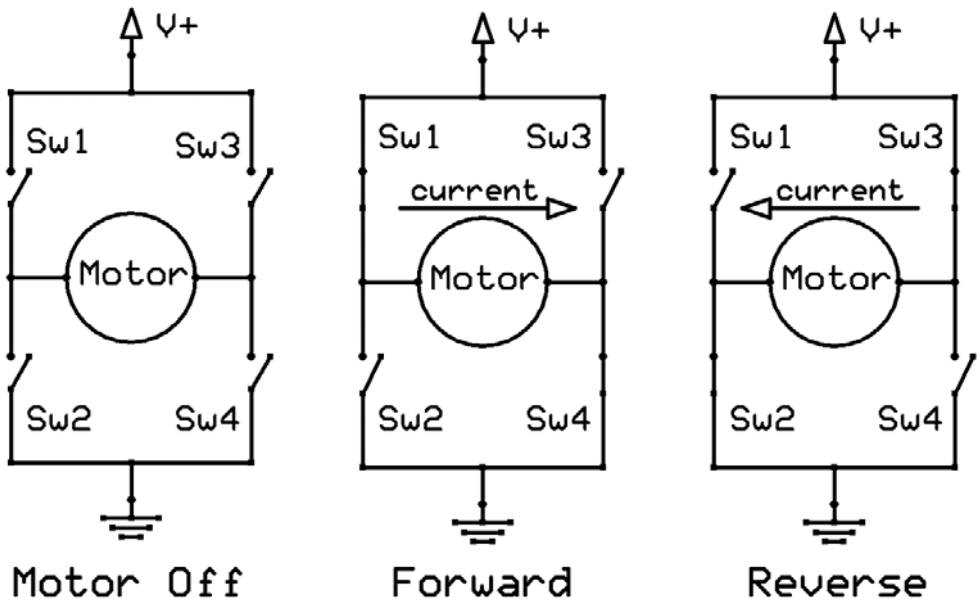


Figure 17-1 Simplified H-Bridge diagram

The L298 Dual H-Bridge Driver

The L298 datasheet, which is available on my website and elsewhere, lists the following specifications that are relevant for its use as a DC motor driver:

- Supplies 2A per motor continuous (3A surge)
- Output voltage range from 7.5V to 46V
- Dual output current-sensing capability
- Thermal protection (automatic shutdown if overheated)
- Standard 5V logic inputs
- High noise immunity

As you can see, the L298 can handle a considerable amount of power. I doubt that I would ever need to run a motor at more than 24 volts, but it's nice to know the capability is there. Also, a continuous 2A per motor output should be plenty for Octavius, and the output current-sensing capability is a great safety feature for whenever Octavius gets himself in a jam.

The L298 pin-out is presented in Figure 17-2. The chip is installed vertically so the large metal tab is available for attaching a heat sink, which is recommended for higher power capability. So far in my testing, the L298 hasn't gotten overly warm, but I'll add one anyway, just to be safe.

The L298 pin-out isn't very breadboard- or stripboard-friendly; the spacing between adjacent pins is 0.1 inches (2.54 mm), which is perfect, but the two rows are staggered. As a result, all the pins in one row would have to be bent 0.05 inches (1.27 mm) in order for both rows to fit into a breadboard or stripboard. The project we are about to construct avoids this problem by using a PC board (available on my website) that's specifically designed for the L298 pin-out (see Figure 17-3). The extra-wide traces for the output lines ensure that the board is able to handle all the power that the L298 can provide.

If you don't want to construct the complete project but would like to conduct the experiments that follow, you may want to bend the L298 pins so that they can be inserted into a stripboard, and add two male headers to insert the stripboard

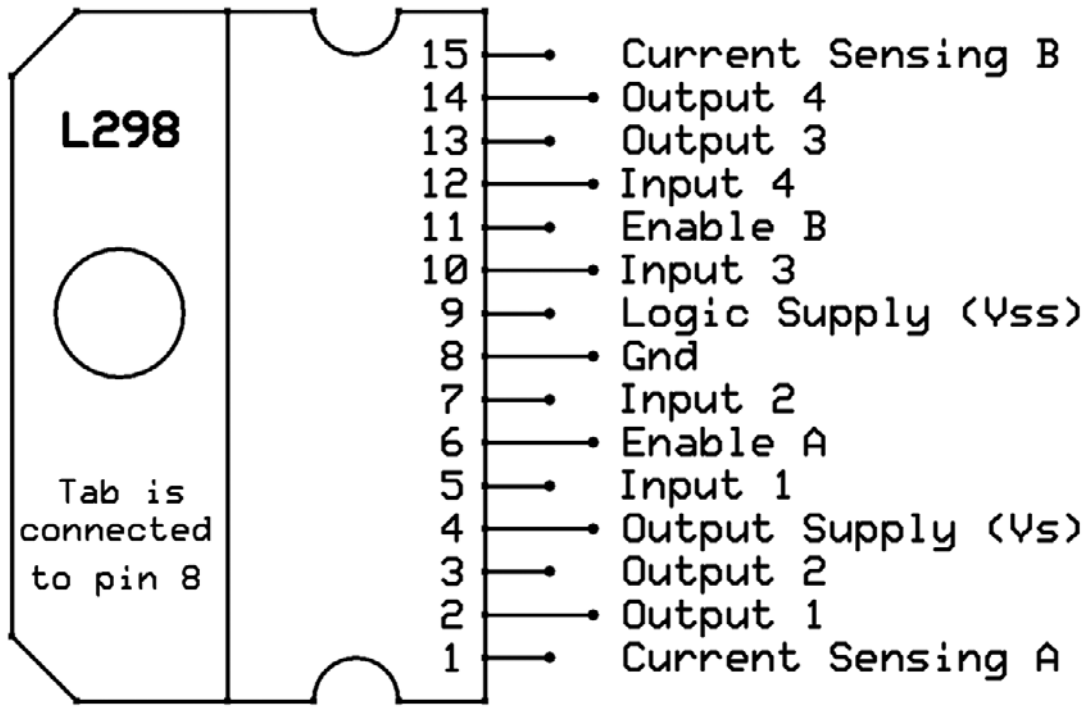


Figure 17-2 L298 pin-out

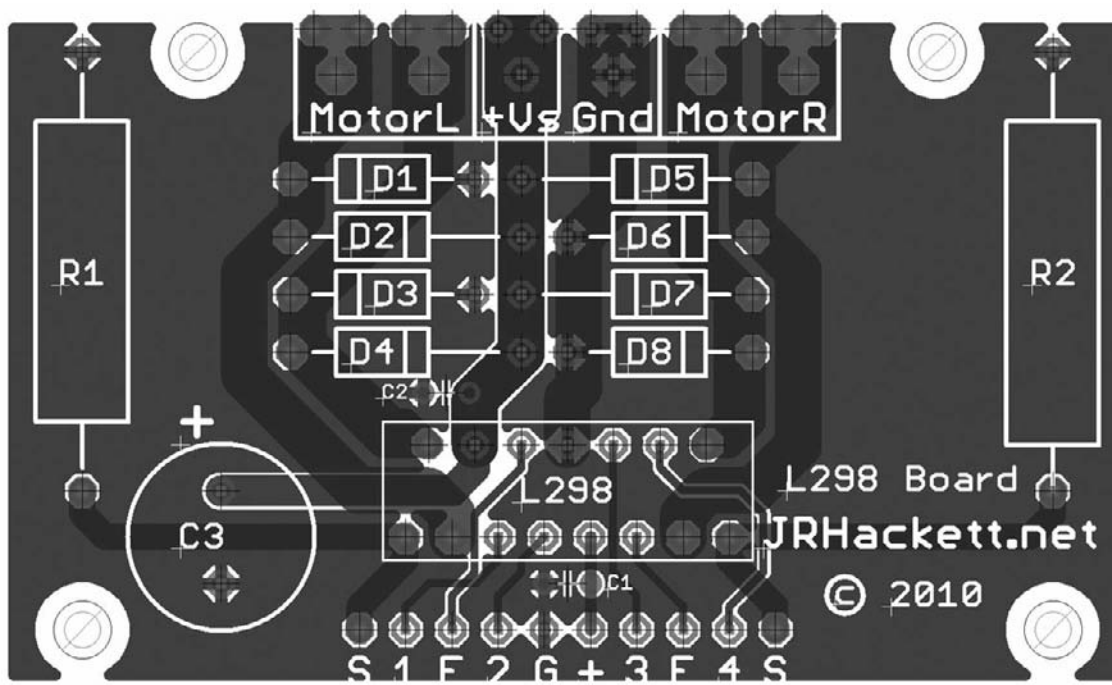


Figure 17-3 The L298 motor controller PC board

adapter into a breadboard. This arrangement should work fine, but probably only for the low-power motors that we're going to use in our experiments. For driving motors with higher power requirements, I would strongly recommend the PC board.

Project 17
Constructing an L298 Dual DC Motor Controller Board

Figure 17-4 presents the schematic of the L298 motor controller board. Diodes D1–D8 are necessary to protect the L298 from the voltage spikes that occur whenever power is applied to or removed from a DC motor. (Note that they are installed with their anodes, not their cathodes, pointing toward Ground.) This phenomenon is known as counter electromotor force (CEMF), and is a consequence of the inductive characteristics of DC motors. We're not going to get into the details here, but if you are interested in learning more about CEMF, a quick web search will produce an abundance of references. C3 is a 470µF electrolytic capacitor that's included to help minimize the voltage fluctuations whenever the motor power is switched on or off. R1 and R2 are each 0.5, 3-watt power resistors that are used in the current-sensing circuitry—we'll discuss them when we get to that topic.

Constructing the L298 PC Board

The parts list for our project is presented in the Parts Bin. All parts are available on my website.

Assembling the PC board is a quick and easy process. As usual, read through the complete list of assembly instructions that follows to be sure you understand the entire procedure before beginning.

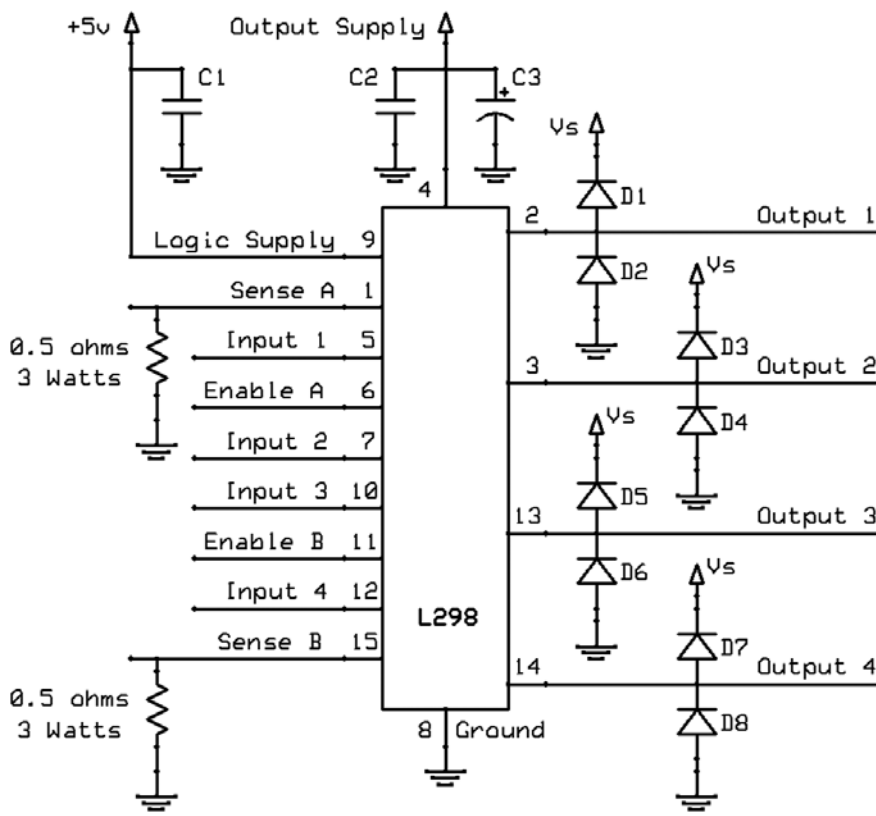


Figure 17-4 L298 motor controller schematic

PARTS BIN	
ID	**Part**
—	L298 printed circuit board (see text)
L298	L298 dual full-bridge driver
C1, C2	Capacitors, 0.1μF
C3	Capacitor, electrolytic, 470μF
D1-D8	Diodes, 1N5818
R1, R2	Power resistors, 0.5 ohms, 3 W
—	Male header, straight, 10 pins
—	Screw terminals, 5-mm pitch, 6 pins

1. Install capacitors C2 and C3; solder and snip the leads.
2. Install diodes D1–D8 (observing the "reverse" polarity); solder and snip the leads.
3. Install the two power resistors; solder and snip the leads.
4. Install and solder the six-pin screw terminals.
5. Install the 470uF electrolytic capacitor (be sure to observe the correct polarity); solder and snip the leads.
6. Install and solder the L298 in place.
7. File or sand all the cut leads on the bottom of the board.
8. Install the ten-pin male header from the bottom of the board; solder the short end of its leads on the top of the board.
9. Clean the flux from the bottom of the board and allow it to dry.
10. Inspect the board carefully for accidental solder connections or other problems.

When you have completed the assembly, we're almost ready to test the PC board. Before we can do that, however, we need to discuss the relationships between the logic levels on the L298's input pins and the resulting motor behavior. These relationships are summarized in Table 17-1. Each motor is controlled by the state of three input pins: Enable A and inputs 1 and 2 control one motor (the one labeled "left" on the schematic); Enable B and inputs 3 and 4 control the "right" motor. As you can see, the relationship is the same for both motors.

To begin, we need to clarify an essential point: Forward and reverse are relative terms; it all depends on which way you wire the motor. When actually constructing a robot, the simplest approach to this issue is to randomly connect the two wires from the left motor to the left motor outputs and randomly connect the two wires from the right motor to the right motor outputs and see what happens. If you program your robot to move

TABLE 17-1 L298 Bidirectional DC Motor Control

Enable A (B)	Input 1 (3)	Input 2 (4)	Motor Behavior
High	High	High	Fast Stop (Braking)
High	High	Low	Forward Motion
High	Low	High	Reverse Motion
High	Low	Low	Fast Stop (Braking)
Low	High	High	Coasting Stop
Low	High	Low	Coasting Stop
Low	Low	High	Coasting Stop
Low	Low	Low	Coasting Stop

forward and he spins in a circle, the connections to the motor that are turning in reverse (or the corresponding software definition of "forward") need to be reversed. Of course, if your creation runs in reverse, the connections to both motors (or both software definitions of "forward") need to be reversed.

Assuming that issue has been resolved, let's examine the data in Table 17-1. As you can see, the enable input has the highest priority. If your robot is in motion when the *enable* input is driven low, then power to the motor is turned off. The levels of the associated inputs don't matter—a "coasting" stop will occur, which means that your robot will gradually come to a stop. When *enable* is high, the situation is more complex: If the two associated inputs are at different levels, then the motor is powered either in a forward or reverse direction, depending on which input is "high." If the two inputs are driven to the same level when the robot is in motion, he will come to a "fast" stop. In other words, the "brakes" will be applied and he will stop more quickly than the "coasting" stop I already mentioned. (In actual practice, I haven't found much difference between the two types of "stops.")

The remaining two situations (forward and reverse motion) are relatively simple. When *enable* is driven high and the two associated inputs are at different levels, the motor will spin in one direction or the other. As we discussed earlier, once you have determined which combination of input levels produces the forward motion, you're all set.

We're going to conduct four experiments with our motor controller board, all of which will require the same setup on our 20X2 master processor board. The schematic for our experimental setup is presented in Figure 17-5. As you can see, output C.5 on the 20X2 is connected to both enable pins on the L298. This is because

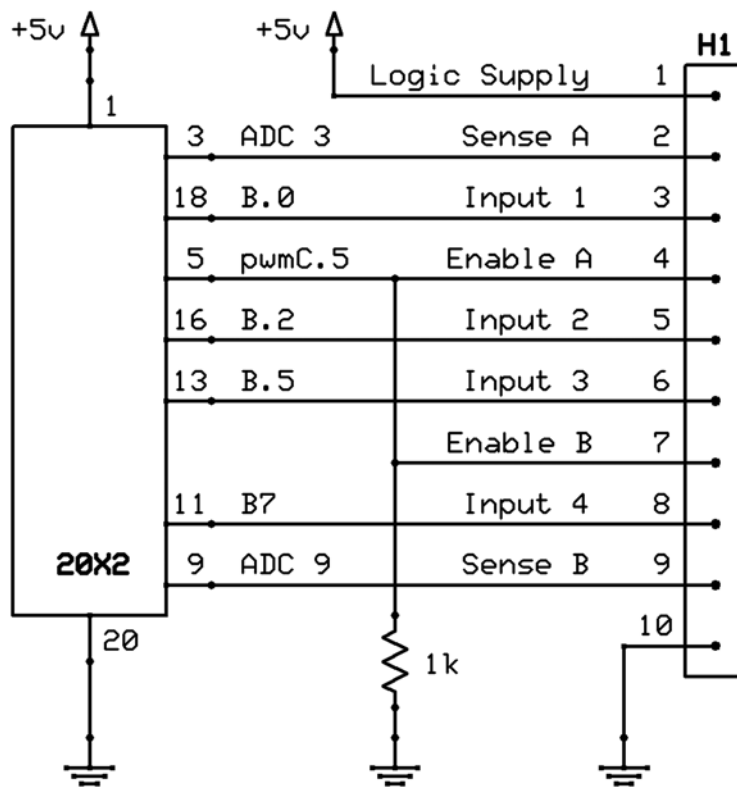


Figure 17-5 Schematic for 20X2 master processor motor controller circuit

C.5 is the 20X2's only PWM pin, so we're going to use it for both *enable* inputs to the L298. Also, if you're wondering about the 1k pull-down resistor on the C.5 line, it's there to make sure the line remains low during the brief time between applying power to the circuit and the point at which the software configures pin C.5 as a low output.

Experiment 1: Testing the Completed L298 Circuit

Before we actually connect a motor to our L298 board we're going to test it first, using a simple LED setup. Figure 17-6 presents the schematic for the test circuit, which we're going to assemble on a separate breadboard and power with a 9V battery because, as we discussed earlier, the L298 requires at least 7.5V to operate correctly. In the schematic, the six screw-terminal connections of the L298 board are at the left, with the "left" motor outputs at the top. For each pair of motor outputs, the polarity of the two LEDs (red and green) is reversed, so only one LED in each pair can light at a time. I have arbitrarily assigned the green LEDs to light when forward motion would be produced if a motor were actually attached; the red LEDs, of course, signify reverse motion.

Figure 17-7 is a photo of my master processor breadboard setup for Experiment 1, which includes the wiring for the motor controller circuit (Figure 17-5) and the auxiliary LED breadboard circuit (Figure 17-6). Using both schematics and the photo of the complete setup shown in Figure 17-7, assemble the circuit for Experiment 1.

CAUTION **Make sure you observe the correct polarity for the 9V battery connections.**

The software for Experiment 1 (Exp1&2-DirectionTest.bas) is presented in Listing 17-1. The main program loop sequentially calls each subroutine and transmits the corresponding motor directions to the terminal window. Each subroutine simply establishes the correct voltage levels on the L298 inputs to produce the desired voltage levels

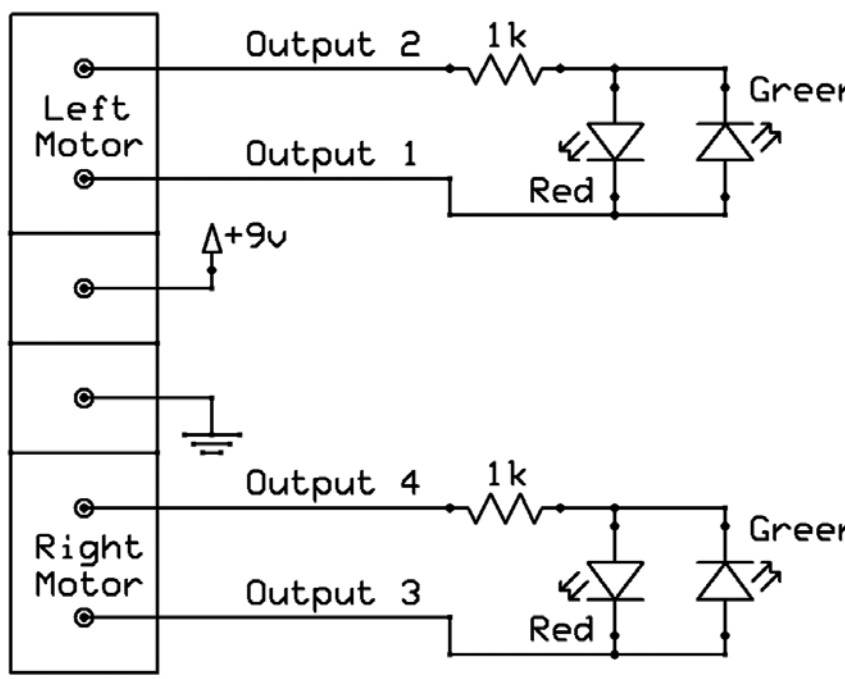


Figure 17-6 Schematic for the auxiliary breadboard circuit of Experiment 1

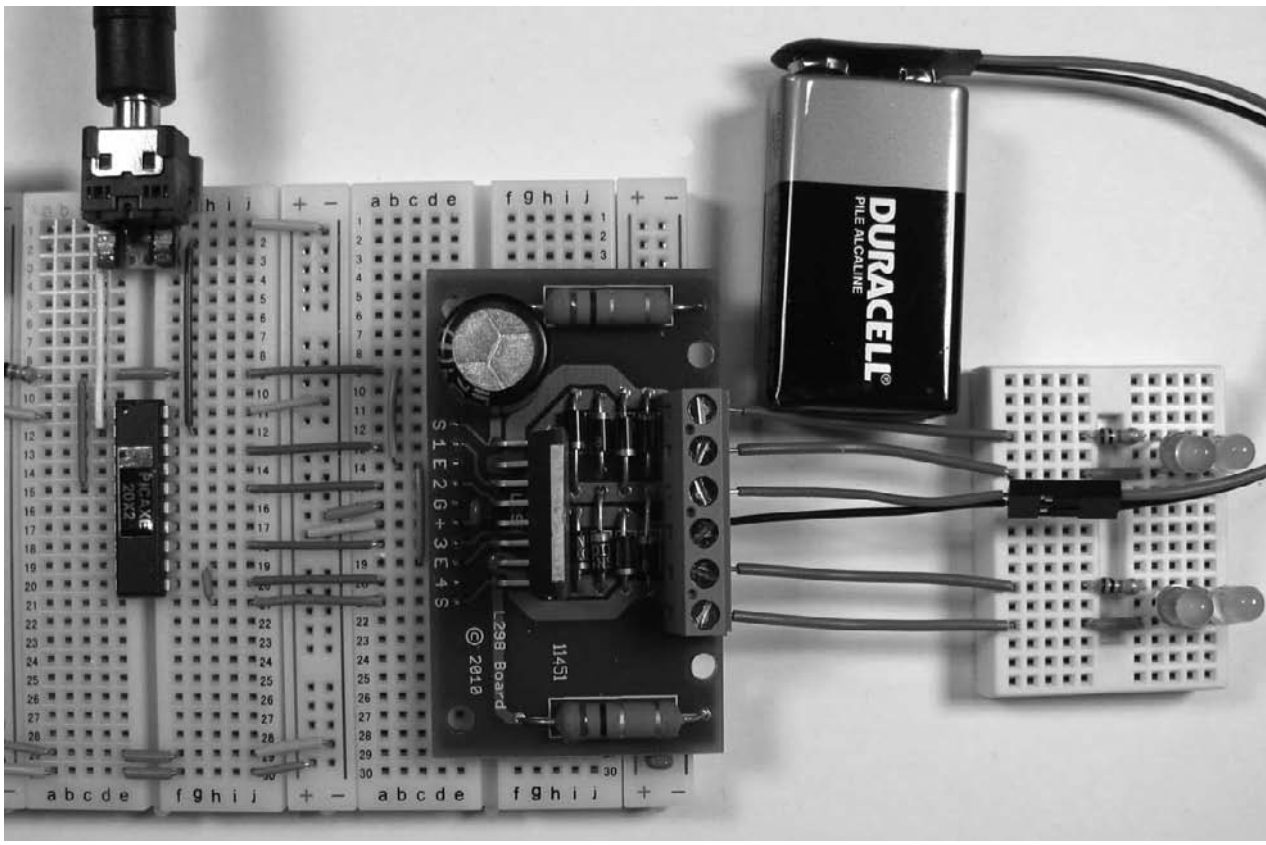


Figure 17-7 Breadboard setup for Experiment 1

LISTING 17-1

```
' ========================== Exp1&2-DirectionTest.bas ==========================
' This program runs on the 20X2 master processor. It demonstrates L298 motor
' direction control.

' === Constants ===
  symbol enable = C.5

' === Directives ===
  #com 4                                ' specify com port
  #picaxe 20X2                          ' specify processor
  #no_data                              ' save time downloading
  #no_table                             ' save time downloading
  #terminal 9600                        ' open terminal window

' ============================ Begin Main Program ============================
do
  sertxd ("Forward",CR,LF)
  gosub fwd
  wait 2

  sertxd ("Reverse",CR,LF)
  gosub bak
  wait 2
```

LISTING 17-1 (continued)

```
  sertxd ("Left",CR,LF)
  gosub left
  wait 2

  sertxd ("Right",CR,LF)
  gosub right
  wait 2
loop

' ==== End Main Program - subroutines follow ====
fwd:                                    ' go forward
  gosub stopp                           ' always stop first
  high B.0                              ' setup for forward
  low  B.2
  high B.5
  low  B.7
  high enable                           ' enable output
  return

bak:                                    ' go in reverse
  gosub stopp                           ' always stop first
  low  B.0                              ' setup for reverse
  high B.2
  low  B.5
  high B.7
  high enable                           ' enable output
  return

left:                                   ' rotate left
  gosub stopp                           ' always stop first
  low  B.0                              ' setup for left turn
  high B.2
  high B.5
  low  B.7
  high enable                           ' enable output
  return

right:                                  ' rotate right
  gosub stopp                           ' always stop first
  high B.0                              ' setup for right turn
  low  B.2
  low  B.5
  high B.7
  high enable                           ' enable output
  return

stopp:                                  ' stop
  low enable                            ' disable output
  wait 2
  return
```

on the two pairs of motor outputs and transmits the relevant data to the terminal window to assist in debugging the circuit. If you compare the level definitions implemented in any one of the subroutines with the data presented in Table 17-1, the correspondence will be obvious.

NOTE **Remember, a turn is actually a rotation; therefore, one motor is running forward and the other is running in reverse in any turn.**

The only aspect of Exp1&2-DirectionTest.bas that requires an explanation is the fact that each subroutine calls the *stopp* subroutine before implementing the desired motor directions. For our current LED test, this is totally unnecessary, but we're going to use the same software in Experiment 2 with a small DC motor; at that point, it will be important. Whenever a DC motor is running in one direction and it is immediately switched to running in the opposite direction, the motor's current draw increases dramatically for a brief period. In fact, it can easily double that of a full motor stall. Obviously, this situation could be disastrous for our L298 board. In addition, it puts a considerable strain on the motor and the gearbox, which will ultimately shorten the life of both. The safest approach to this issue is to always include a call to a *stopp* subroutine at the beginning of each of your motor direction subroutines. (I have a nagging suspicion that I may have forgotten to take my own advice and that's what led to Octavius' original motor controller meltdown.) The moral of the story is:

CAUTION **Never change motor directions without first stopping both motors for a second or two!**

When you have assembled the LED test circuit, download Exp1&2-DirectionTest.bas to your master processor. The pattern of the lit LEDs should be as follows:

- **Forward:** Both motor LEDs = green
- **Reverse:** Both motor LEDs = red
- **Left turn:** Left motor LED = red; right motor LED = green
- **Right turn:** Left motor LED = green; right motor LED = red

If you get a different pattern, you will need to check your wiring and the polarity of the LEDs.

Experiment 2: Controlling a Small DC Motor

When Experiment 1 is functioning correctly, we're ready to use the same setup to actually drive a motor. Any small DC motor that will run at 9V will be fine for Experiment 2. The following motors from my local Radio Shack both did the job: RS #273-255 (9V to 12V) and RS #273-256 (9V to 18V). Of course, larger motors will drain your 9V battery faster, so for this experiment you may want to use the smallest 9V motor you can find. If you happen to have a 12V motor and battery, you can substitute those as well.

The setup for the remaining three experiments is the same; simply disconnect the auxiliary LED breadboard from the L298 board and connect your motor to the right motor screw terminals on the L298 board. (However, don't dismantle the auxiliary breadboard yet; we're going to use it again in the next chapter.) When the motor is powered, it will tend to jump around a bit, so you will either need to hold onto it or attach it to something that is heavy enough to hold it in place. I used a heavy piece of steel from my shop (see the photo in Figure 17-8).

When your setup is complete, run the Exp1&2-DirectionTest.bas program again; the motor should change direction each time a new subroutine is executed. If your motor doesn't come to a full stop between each direction change, you may want to lengthen the *wait* statement in the *stopp* subroutine until you find a value that works.

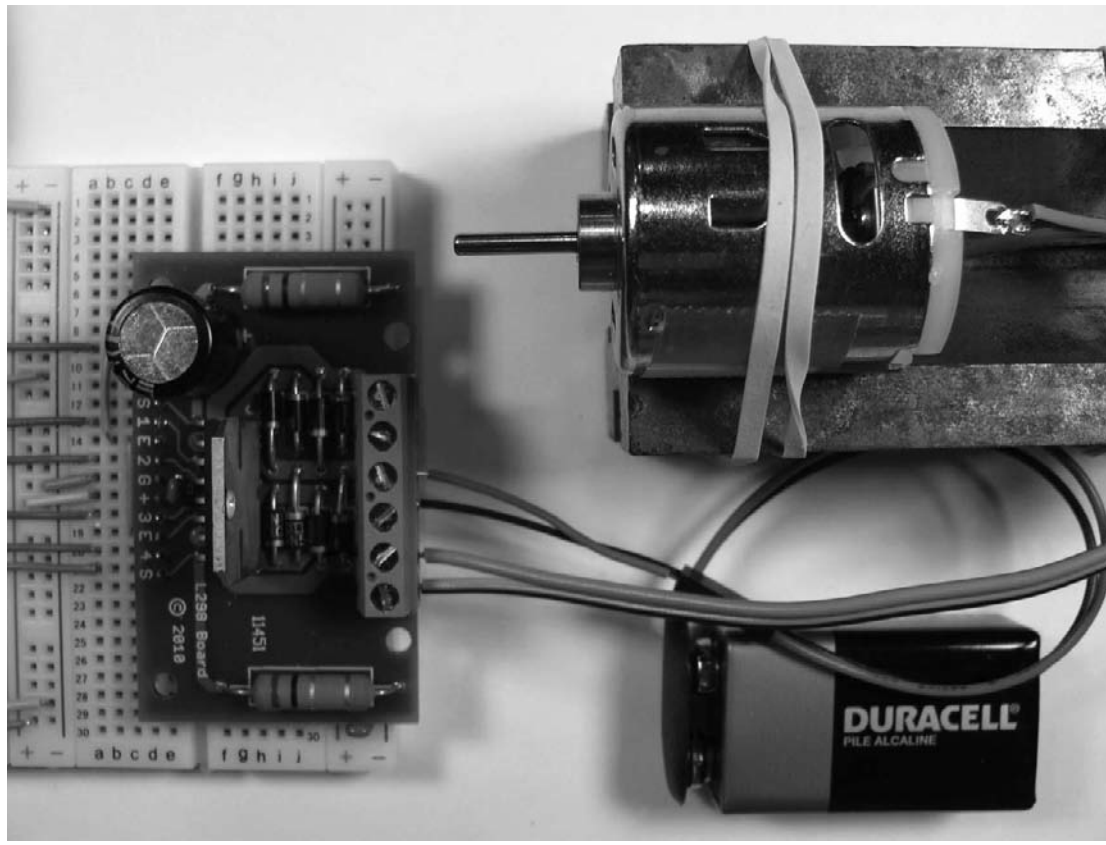
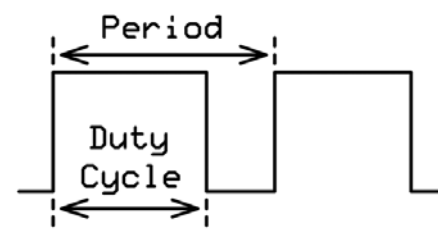


Figure 17-8 Breadboard setup for Experiments 2 through 4

Experiment 3: Implementing Variable Speed Control

Now that our little motor is running correctly at full speed, we're going to experiment with the *pwmout* command to adjust its speed. PWM is an acronym that stands for *pulse width modulation*, which refers to the technique of adjusting the portion of time that a continuous pulse train is driven high. The complete syntax for the *pwmout* command is *pwmout pin, period, duty cycle,* where *pin* specifies the I/O pin to use (C.5 on the 20X2), *period* refers to the amount of time required for one complete cycle of the waveform, and *duty cycle* refers to the amount of time each cycle is driven high. The following illustration should help to clarify the relationship between these terms:



The *pwmout* documentation in the manual includes formulas to compute the values of *period* and *duty cycle* for the specific waveform you want to generate, but it's easier to use the Pwmout Wizard in the Programming Editor; just choose PICAXE | Wizards | Pwmout in the menu structure. Be sure to select the clock speed that you're using, or the wizard's values will be way off the mark.

Pwmout differs from most other PICAXE commands—once your program executes a *pwmout* statement, the specified waveform is continuously generated in the background until the command is cancelled or modified. This feature is what makes *pwmout* so useful for motor control; whenever the processor wants to change your robot's speed or direction, it just needs to issue another pair of *pwmout* commands to do so (one for each motor). However, there's an even better way to accomplish the same goal: the *pwmduty* command. Once you have issued an initial *pwmout* command, *pwmduty* enables you to update the duty cycle whenever necessary. The main advantage of the *pwmduty* command is that it doesn't reset the processor's internal timer each time it's used, as the *pwmout* command does, so the transition from one *duty cycle* value to another is simpler (and probably a little faster). The syntax is also a little simpler: *pwmduty pin, duty cycles.*

The software for Experiment 3 (Exp3-SpeedTest.bas) is presented in Listing 17-2. In it, I have configured the *pwmout* command to generate a 5kHz signal at the 20X2's default clock speed of 8MHz. You certainly can lower that if you want, but some motors produce an annoying hum or buzz at lower frequencies.

Download Exp3-SpeedTest.bas to the same master processor setup we just used in Experiment 2; your motor should repetitively cycle through two speeds (fast and slow) in the same direction. If it either runs too fast or not at all when 75 percent PWM is applied, you will need to change the duty cycle value to obtain the desired results; use the Pwmout Wizard to adjust the value as necessary. (Don't forget to set the clock speed to the 20X2's default of 8MHz.)

When you have the program operating correctly, you may want to experiment with various *duty cycle* values. If you do, you'll discover that there's a significant range of values at which your motor doesn't run at all. The reason is that the motor outputs on the L298 are turned off during the low portion of every PWM cycle; in effect, the motor is operating on a reduced voltage that's directly proportional to the duty cycle value. When the duty cycle is at 100 percent, the motor receives the full 9V (usually 12V in a real-world robot). However, when the duty cycle is at 50 percent, the motor effectively receives only half the battery voltage, which may not be enough to run it at all.

CAUTION **Don't be fooled by motor controller boards that advertise "255 speeds in forward and reverse" because as many as three-fourths of the "speeds" won't run your motor at all!**

Experiment 4: Monitoring Motor Current

Our fourth and final experiment is a simple demonstration of the L298 current-sensing feature. We'll again use the same master processor setup, but with different software (Exp4-CurrentTest.bas), which is shown in Listing 17-3. The program is simple; all it does is send the real-time ADC current-sensing value to the terminal window. Download it to your master processor and while it's running, place a load on the motor by gently pinching its shaft; you should be able to at least double the ADC reading.

LISTING 17-2

```
' ============================== Exp3-SpeedTest.bas ==============================
' Program demonstrates DC motor speed control using the pwmout and pwmduty
' commands.

' === Constants =====
  symbol enable = C.5

' === Directives ====
  #com 4                                ' specify com port
  #picaxe 20X2                          ' specify processor
  #no_data                              ' save time downloading
  #no_table                             ' save time downloading
  #terminal 9600                        ' open terminal window

' ============================ Begin Main Program ================================
low enable                              ' make C.5 an output
pwmout enable, 99, 0                    ' set up PWM (0% duty)
                                        ' (5kHz PWM @ 8MHz clock)
do
  gosub stopp
  high B.0                              ' set up for forward
  low  B.2
  high B.5
  low  B.7
  sertxd ("Forward Fast",CR,LF)
  pwmduty enable, 400                   ' 100% duty
  wait 2

  gosub stopp
  high B.0                              ' set up for forward
  low  B.2
  high B.5
  low  B.7
  sertxd ("Forward Slow",CR,LF)
  pwmduty enable, 300                   ' 75% duty
  wait 2
loop

' ====== End Main Program - subroutines follow ======

stopp:                                  ' stop
  low enable                            ' disable output
  wait 1                                ' wait for motor to stop
  return
```

LISTING 17-3

```
' ====================== Exp4-CurrentTest.bas ======================
' Program demonstrates L298 current-sensing.

' === Constants ======
  symbol enable = C.5

' === Variables ======
  symbol Vin = b0

' === Directives =====
  #com 4                                ' specify com port
  #picaxe 20X2                          ' specify processor
  #no_data                              ' save time downloading
  #no_table                             ' save time downloading
  #terminal 9600                        ' open terminal window

' ====================== Begin Main Program ======================
low enable                              ' make C.5 an output

high B.0                                ' setup for "forward"
low  B.2
high B.5
low  B.7
high enable                             ' start motor

do
' (Pinch motor shaft to see change in ADC value)
  readadc10 11, Vin                     ' get ADC value
  sertxd (#Vin, CR, LF)                 ' send it to Terminal
  pause 500
loop
```

When I ran the program with the motor shown in Figure 17-8 (RS# 273-255), I obtained an ADC reading that fluctuated between 18 and 25. When I pinched the motor shaft, that figure easily doubled. If I completely stalled the motor, it increased to about 90. To see what those results actually mean, let's do some calculations using the maximum value of 90. To begin with, the ADC measurement is of the *voltage* that the L298 is placing across the 0.5-ohm power resistor. Using the same proportional approach as we did in Chapter 6: Vin / 5 = 90 / 1023. If you solve that equation for Vin, it turns out to be 0.44V. (The L298 datasheet specifies a maximum value of 2V for this figure.) To determine the current draw, we just need to apply Ohm's law: I = E / R or I = 0.44 / 0.5 = 880mA.

So, our little "toy" motor draws almost a full Amp when stalled. Don't forget—that figure can easily double if you immediately switch the motor from full forward to full reverse, as I may have accidentally done to poor Octavius. Happily, the current-sensing capability of the L298 motor controller board will enable Octavius (or your robot) to avoid any disasters in the future.

CHAPTER 18

Programming Octavius

Developing an autonomous robot is a challenging and sometimes frustrating experience. It usually goes something like this: First, you spend numerous hours designing and constructing a chassis, power system, and main logic board. Next, you research, select, and implement one or more sensory systems and then write a complex and elegant program that's carefully designed to handle any eventuality. After investing huge amounts of time, energy, and money, you're finally ready to test your new creation. You hit the power switch and what happens? Your pride and joy darts off and runs into the nearest obstacle he can find! If he's a little guy, like my first- and second-generation robots, this really isn't all that much of a problem. You just run after him, pick him up, and try to figure out what went wrong. This is when the frustration (and the real learning) begins. With little or no information at your disposal, you have to greatly simplify your software and test everything one step at a time. It's a time-consuming endeavor—reprogramming and retesting in a seemingly infinite loop—not fun to say the least!

If you're working with a robot like Octavius—one that weighs 25 pounds and has motors that can draw two or three Amps (or more) when stalled—it can also be an expensive process. When Octavius experienced his power train meltdown, I had to design and build the more powerful motor controller that we constructed in the previous chapter. However, I also learned a valuable lesson from this little disaster: Just like a human infant, a newborn robot can't be left entirely on his own during the early stages of development. My first attempt at parental control (a large "kill" switch mounted on Octavius' rear breadboard) worked well enough, but I soon tired of chasing him around my basement workspace.

My second approach to the problem was a little more sophisticated and it enabled me to avoid some unnecessary exercise. I decided to apply our TV-IR remote input module (Project 8) to the world of robotics. In order to understand how helpful this approach can be in the process of Octavius' early development, we first need a sensory system that can give him the illusion of autonomy while we carefully monitor his activity in the background. Because of its simplicity and power, I chose a MaxBotix ultrasonic ranging system to enable Octavius to take his first baby steps into the real world, so let's begin there.

The MaxBotix LV-MaxSonar Ultrasonic Range Finders

MaxBotix (www.MaxBotix.com) offers three different lines of ultrasonic range finders. The LV-MaxSonar line includes five sensors that differ primarily in the beam width for object detection (see Figure 18-1, which is adapted from the MaxBotix documentation). In order to provide Octavius with the widest possible range of

"vision," I chose the EZ0 unit because it offers the widest beam width as well as the longest range of object detection. As you can see in Figure 18-1, with a 5V supply, the EZ0 is capable of detecting a 1-inch (2.5-cm) dowel at a distance of almost 10 feet (3 meters).

NOTE **At this point, you may want to visit the MaxBotix site and download the EZ0 datasheet for reference during the following discussion.**

The LV-MaxSonar units can be operated at either +3.3V or +5V with a surprisingly low current draw of 2mA. They automatically update their ranging data every 50mS and simultaneously output the data in three different formats (analog voltage, PWM, and standard serial data) that are all easy to interface with any PICAXE processor. I think the analog voltage output is the simplest of the three formats because it doesn't require any coordination with the master processor; the latest reading is always available for the processor to input as an analog voltage whenever it's convenient. In addition, the math involved in converting the ADC reading to a distance is about as easy as it gets because the scaling factor for the sensor's analog voltage output is V_{cc} /512 per inch. Since the *readadc10* command divides V_{cc} into 1024 steps, we just need to divide the ADC result by 2 in order to obtain the distance to the nearest object in inches. Finally, the LV-MaxSonar units can easily be daisy-chained so that multiple sensors can be operated simultaneously without interfering with each other. Although we won't be using that feature in this chapter, keep it in mind

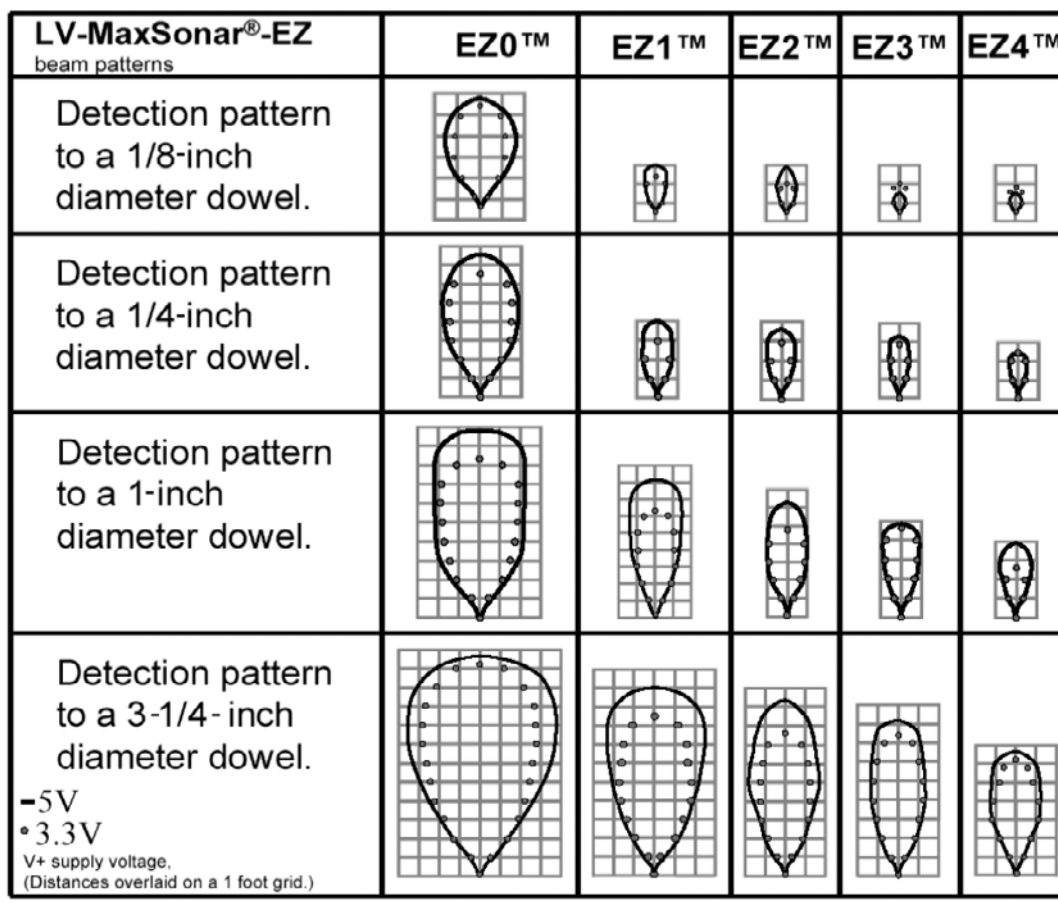


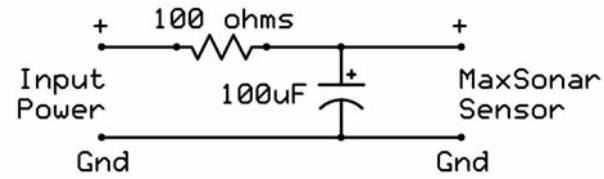
Figure 18-1 LV-MaxSonar detection-beam patterns

for any future plans you may have for your robot. (I already have two more EZ0 sensors on order as a present for Octavius' first birthday!)

Experiment 1: Testing the MaxSonar-EZ0 Sensor

The LV-MaxSonar units are relatively small (less than 1 sq. in.) and easy to interface. The composite photo shown in Figure 18-2 actually shows an EZ1 unit, but all the LV sensors have the same form-factor. In the photo, you can see that the I/O pins are clearly labeled on the back of the PC board. (MaxBotix also included a little gratuitous proselytizing, but these sensors are so good that I forgive them!) Since we're going to use the analog voltage output (labeled ADC in the photo), we only need to make the three indicated connections to interface the range finder with our master processor.

MaxBotix recommends filtering the MaxSonar power supply in electrically noisy environments to improve the stability of the output data. Even though Octavius' DC motors are on a separate supply, I also included the recommended components as shown in the following illustration, just in case.



Of course, the 100Ω resistor drops the voltage a little (200mV at a current draw of 2mA), but the MaxSonar units operate all the way down to 2.5V, so I assumed that a little voltage drop wouldn't affect their operation at all. (That assumption turned out to be wrong—we'll discuss that shortly.) In any case, the filter ensures a clean power supply to the sensor. Figure 18-3 is a photo of my completed EZ0 circuit installed on Octavius' front breadboard. (I made the tiny stripboard circuits you see in the photo to help align the breadboards with the PC board before sticking them in place.) If you aren't yet building a robot, you can easily install the EZ0 on our master processor breadboard. Simply connect its analog output to the ADC5 input (pin 15) on the 20X2. Don't forget to include the EZ0 power line filter as shown earlier in the illustration.

When you have completed the circuit, we're ready to test the MaxSonar unit. I did all my

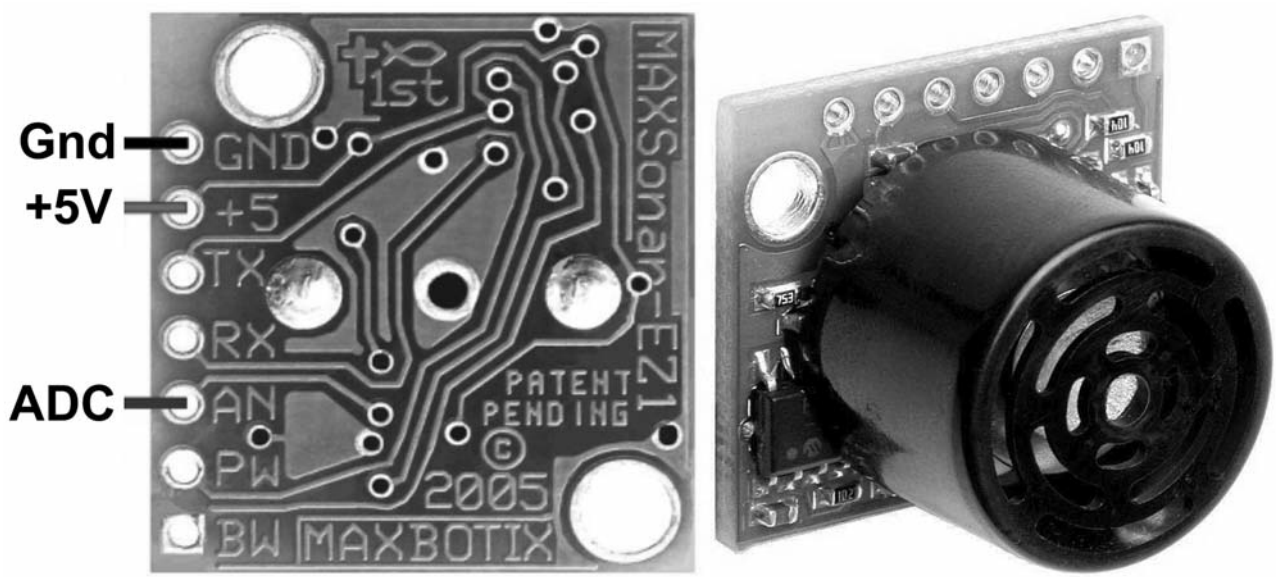


Figure 18-2 MaxBotix MaxSonar-EZ1 composite photo with pin-out

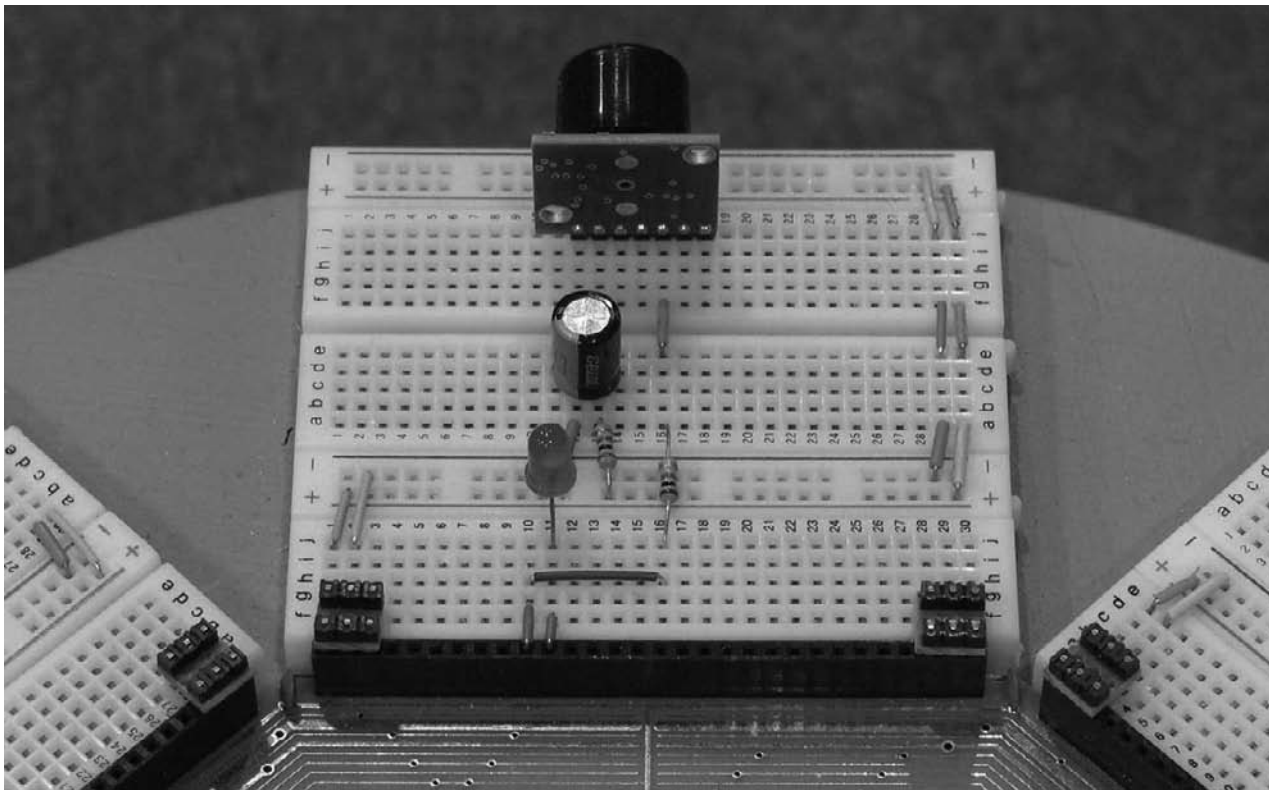


Figure 18-3 Octavius' new "eye"

testing on Octavius, but the following software (EZ0toTerm.bas; shown in Listing 18-1) is written so that you can conduct your tests directly on the 20X2 master processor board. When you run the program, move an obstacle to various distances from the EZ0. Measure the distance each time and note how closely it compares to the EZ0 data. In case you haven't read the datasheet, I should point out that 6 inches (15 cm) is the minimum distance that the EZ0 can measure; an object closer than that will still be reported as being 6 inches away.

When I first ran Experiment 1, my EZ0 output was consistently two or three inches less than the actual measurement to the obstacle. My first thought was that the small voltage drop produced by the power line filter that MaxBotix recommends might be the cause of this minor discrepancy, so I replaced the 100Ω resistor with a jumper wire. Sure enough, the EZ0 output became much more accurate (i.e., within one inch of the correct measurement every time). In spite of the error introduced by the filter, I decided to reinstall it in the circuit because of the potentially disruptive effects from the electrical noise that may be produced by Octavius' DC motors. The error introduced by the filter is easy to correct in the software—just add 2 to each distance measurement (right after dividing the raw data by 2).

Experiment 2: Adding a Four-Digit LED Display

As I have already mentioned, part of the difficulty in diagnosing the problem when a robot "misbehaves" stems from the fact that we have very little information at hand at the moment the problem occurs. To improve this situation, I decided to add a display to Octavius. Since I wanted to be able to read the display from a distance, I chose to add a four-digit LED display in spite of its relatively high power consumption. Naturally, I first thought of using the SPI display

LISTING 18-1

```
' ================================ EZ0toTerm.bas ================================
' Program uses a MaxSonar-EZ0 to measure distance to nearest object &
' sends results to the terminal.

' === Constants ===
  symbol EZ0 = 5                          ' EZ0 on ADC5
  symbol LED = 5                          ' debugging LED on B.5

' === Variables ===
  symbol dist = w0                        ' word data from EZ1

' === Directives ===
  #com 3                                  ' specify serial port
  #PICAXE 20X2                            ' specify processor
  #no_data                                ' save time downloading
  #no_table                               ' save time downloading
  #terminal 9600                          ' open terminal window

' ============================ Begin Main Program ============================
setfreq m8
dirsB = %11110111                         ' configure pinB.3 (ADC5) as input
adcsetup = %0000000000100000              ' setup for ADC5 (see Manual)
do
  high LED                                ' for debugging
  wait 1
  readadc10 EZ0, dist                     ' get EZ0 raw data
  dist = dist / 2                         ' convert to inches
  sertxd ("Distance = ",#dist,cr,lf)
  low LED
  wait 1
loop
```

we constructed in Chapter 12. However, I didn't want to use three of Octavius' I/O pins for this purpose, so I opted for a serial LED display that I had developed earlier for the "PICAXE Primer" column in *Nuts and Volts* magazine. If you are interested in using this display, the bare PC board is available on my website.

However, for Experiment 2, I modified the program that I used with Octavius so that it runs on our 20X2 master processor board with the SPI LED display. My breadboard setup for this experiment is shown in Figure 18-4. I realize that the angle of the photo is a little steep for viewing the LED display, but I wanted the necessary connections to the display to be visible (hspi data out = C.1, clock = B.7, and load = C.2). The analog voltage output from the EZ0 is connected to ADC6 (B.4), and there's a debugging LED on A.0. Also, the filter recommended by MaxBotix is included in the power line to the LED. (The 100Ω resistor is visible in front of the display, and you can see the top of the 100µF electrolytic capacitor sticking out behind it.)

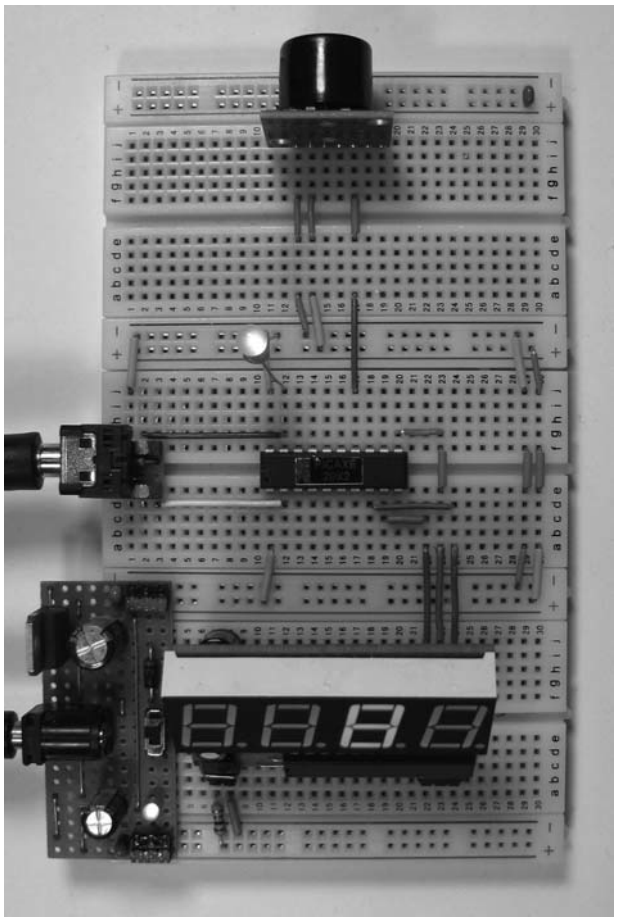


Figure 18-4 Serial LED display mounted on Octavius' rear breadboard

When you have completed your breadboard setup, download the EZ0toLEDs.bas program, which is presented in Listing 18-2. You should see the real-time distance measurement being continually updated on the LED display. Since the filter is included in the circuit and the correction is in the software, the results should be fairly accurate. However, if a small constant discrepancy remains, you may want to adjust the value of the correction in the program.

Who's in Charge Here?

Now that we have Octavius' ultrasonic range finder operating properly, we're ready to implement the TV-IR remote system that I mentioned at the beginning of the chapter. We've already developed and tested the necessary hardware back in Project 8, so let's see how we can use our IR-TV input module to exercise a little supervisory control over Octavius (or any robot, for that matter).

LISTING 18-2

```
' ================================ EZ0toLEDs.bas ================================
' Program uses MaxSonar EZ0 to measure the distance to the nearest object
' & sends results to the EG SPI LED.

' === Constants ===
  symbol EZ0    =    6                  ' EZ0 analog on ADC6 (B.4)
  symbol LED    = A.0                   ' debugging LED on A.0
  symbol load   = C.2                   ' 7219 "Load" pin
  symbol blank =   15                   ' used to blank an LED

' Register addresses for the MAX7219
  symbol decode =  9                    ' decode register
  symbol brite   = 10                   ' intensity register
  symbol scan    = 11                   ' scan-limit register
  symbol on_off = 12                    ' 1 = display on; 0 = off

' === Variables ===
  symbol dist = w0                      ' data from EZ0
  symbol ones = b2                      ' for LED display
```

LISTING 18-2 (continued)

```
  symbol tens = b3                          ' for LED display

' === Directives ===
  #com 3                                    ' specify serial port
  #PICAXE 20X2                              ' specify processor
  #no_data                                  ' save time downloading
  #no_table                                 ' save time downloading
  #terminal off                             ' disable terminal window

' ============================ Begin Main Program ==============================
  setfreq m8
  dirsb = %11101111                         ' EZ0 is B.4 (ADC6)
  dirsc = %10111111                         ' C.6 is input only
  adcsetup = %0000000001000000              ' enable ADC6 (B.4)
  hspisetup spimode00,spislow               ' set up hspi

' Initialize MAX7219
  hspiout (scan,3)                          ' set scan for digits 0-3
  pulsout load,1
  hspiout (brite,5)                         ' set brightness to 5
  pulsout load,1
  hspiout (decode,15)                       ' set BCD for digits 0-3
  pulsout load,1
  hspiout (on_off,1)                        ' turn display on
  pulsout load,1

do
  high LED                                  ' for debugging
  wait 1
  readadc10 EZ0,dist                        ' get EZ0 raw data
  dist = dist / 2                           ' convert to inches
  dist = dist + 2                           ' adjust for power filter
  tens = dist / 10                          ' isolate the tens digit
  ones = dist // 10                         ' isolate the ones digit
  if tens = 0 then                          ' for zero-blanking
    tens = blank
  endif

' Send data to the four LED digits
  hspiout (1,blank)
  pulsout load,1
  hspiout (2,tens)
  pulsout load,1
  hspiout (3,ones)
  pulsout load,1
  hspiout (4,blank)
  pulsout load,1
  low LED
  wait 1
loop
```

Experiment 3: Controlling the L298 Board with a TV Remote

In this experiment, we're simply reusing the same 20X2 master processor setup that we used in Experiment 1 of the previous chapter to test our L298-based motor controller board and adding the TV-IR input module from Chapter 8 in order to be able to remotely control the functioning of the L298 board. Figure 18-5 is a photo of my breadboard setup for the experiment. As you can see in the photo, the input of the TV-IR input module is connected to pin C.4 of the 20X2 and its output is connected to pin C.1. These connections differ from the ones we used in Chapter 8 because pin C.5 is now being used for the PWM output to the L298 board, so it's no longer available for use with the TV-IR input module. Also note that I have bent the pins of the PNA4602 so that the sensor is pointing directly up; that way, when I install the unit on one of Octavius' breadboards, I'll be able to control him from any direction.

We're going to use five commands to control our motors, as shown in Figure 18-6. As you can see, I have taken advantage of the typical diamond-shaped pattern of the keys that we will be using so that the remote control will be as simple as possible to operate. (When Octavius is about to crash into something, I don't want to have to look at the remote to find the key to stop him.) If you are interested in the specific remote control in Figure 18-6, it's available for less than $5 from The Home Depot (SKU #372731).

The driver software that we used for the TV-IR input module in Chapter 8 needs to be slightly modified to accommodate our new command structure. The resulting program (IRtoOctavius.bas) is shown in Listing 18-3. As you can see, it's even simpler than the version we used in Chapter 8 because we're not using the number keys this time.

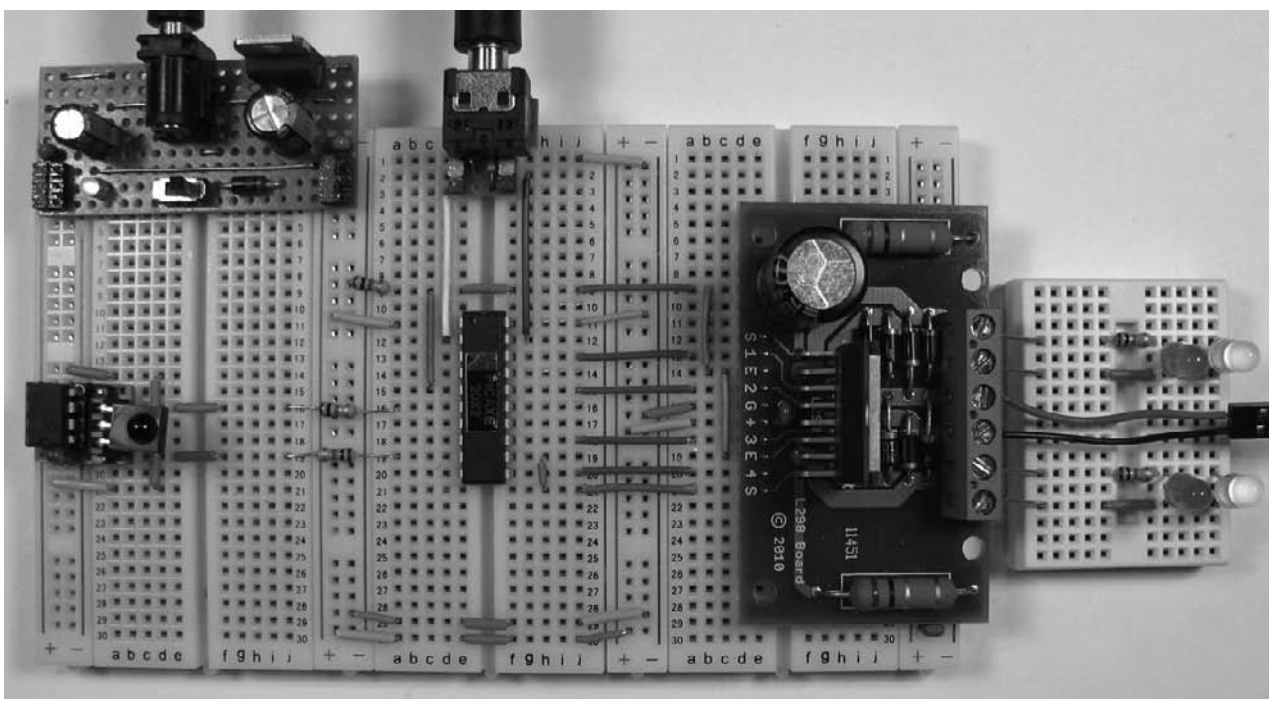


Figure 18-5 Breadboard setup for TV remote control of the L298 board

Command	Key	Value
Forward	Channel Up	16
Reverse	Channel Down	17
Turn Right	Volume Up	18
Turn Left	Volume Down	19
Stop	Mute	20

Figure 18-6 TV remote control and commands used in Experiment 3

LISTING 18-3

```
' ============================== IRtoOctavius.bas ==============================
' Driver program for 08M2-based TV-IR input module (Ch. 8).
' It waits for a keypress from a SIRC TV remote control
' and then transmits the corresponding value to Octavius.

' === Constants ===
  symbol toOct = C.2                    ' serout line to Octavius
  symbol frOct = C.4                    ' input line from Octavius

' === Variables ===
  symbol junk = w0                      ' word required by pulsin

' === Directives ===
  #com 4                                ' specify com port
  #picaxe 08M2                          ' specify processor
  #terminal off                         ' disable terminal window

' ============================ Begin Main Program ============================
do
  irin C.3, infra                       ' wait for IR input

  select case infra
    case 16 to 20                       ' 4 arrows & mute
      high toOct                        ' "ready to send"
      pulsin frOct,1,junk               ' junk is junk
      low toOct
      serout toOct,N2400_4,(infra)      ' send it
  endselect

  pause 500                             ' slow it down a bit
loop
```

When you have completed your breadboard setup for the experiment, remove the 08M2 from its socket, install it on a second breadboard, reprogram it with the IRtoOctavius.bas software, and reinstall it on the TV-IR module.

The software that we need to install on our master processor (IRmotorControl.bas) is presented in Listing 18-4. As you can see, it's essentially a combination of the software we used in Chapters 8 and 17, with two minor

LISTING 18-4

```
' ============================ IRmotorControl.bas ==============================
' Program runs on the 20X2 master processor. It receives
' data from the IR-TV module and controls 2 DC motors.

' === Constants ===
  symbol fromIR = C.1
  symbol toIR   = C.4
  symbol enable = C.5

' === Variables ===
  symbol cmnd = b0
  symbol IRflag = pinC.1                ' as usual, this is a var

' === Directives ===
  #com 3                                ' specify com port
  #picaxe 20X2                          ' specify processor
  #no_data                              ' reduce download time
  #no_table                             ' reduce download time
  #terminal off                         ' disable terminal window


' ============================ Begin Main Program ==============================
do
  pause 100                             ' pretend to be busy
  if IRflag = 1 then
    gosub getData
    select case cmnd
      case 16
        gosub fwd
      case 17
        gosub bak
      case 18
        gosub right
      case 19
        gosub left
      case 20
        gosub stopp
    end select
  endif
loop
```

LISTING 18-4 (continued)

```
' ==================== End Main Program - Subroutines Follow ====================
getData:
  pulsout toIR,10                       ' 50uS "send it" pulse
  serin fromIR,N2400_8,cmnd             ' get data
  return

fwd:                                    ' go forward
  gosub stopp                           ' always stop first
  high B.0                              ' setup for forward
  low  B.2
  high B.5
  low  B.7
  high enable                           ' enable output
  return

bak:                                    ' go in reverse
  gosub stopp                           ' always stop first
  low  B.0                              ' setup for reverse
  high B.2
  low  B.5
  high B.7
  high enable                           ' enable output
  return

left:                                   ' go left
  gosub stopp                           ' always stop first
  low  B.0                              ' setup for left turn
  high B.2
  high B.5
  low  B.7
  high enable                           ' enable output
  return

right:                                  ' go right
  gosub stopp                           ' always stop first
  high B.0                              ' setup for right turn
  low  B.2
  low  B.5
  high B.7
  high enable                           ' enable output
  return

stopp:                                  ' stop
  low enable                            ' disable motor output
  wait 2
  return
```

differences: First, the *fromIR* and *toIR* constants are defined differently, as I mentioned earlier; second, the main loop simply executes the appropriate subroutine for each of the five button-presses on the TV remote; all other buttons are ignored.

Download IRmotorControl.bas to your master processor. Each button-press on the TV remote should produce the appropriate LED pattern on the auxiliary breadboard. (There will be a slight delay due to the *stopp* subroutine we have included at the beginning of each of the other subroutines.) You may also want to experiment with placing your breadboard setup on the floor in the middle of a room to determine whether you have full 360-degree control of the motor outputs.

Project 18
Hail, Octavius!

Now that we have tested all the necessary components (L298 motor controller board, TV-IR input module, EZ0, and serial LED display), we're ready to integrate them into a system that will allow us to exercise supervisory control over Octavius' behavior as he pursues his quest for independence. Figure 18-7 is a photo of the completed system installed on Octavius' breadboards. As you can see, I have also upgraded his motor control circuitry with our L298 board.

The program that controls and integrates all the new hardware that we have developed (HailOctavius.bas) is too long to include here; you can download it from my website. Due to its

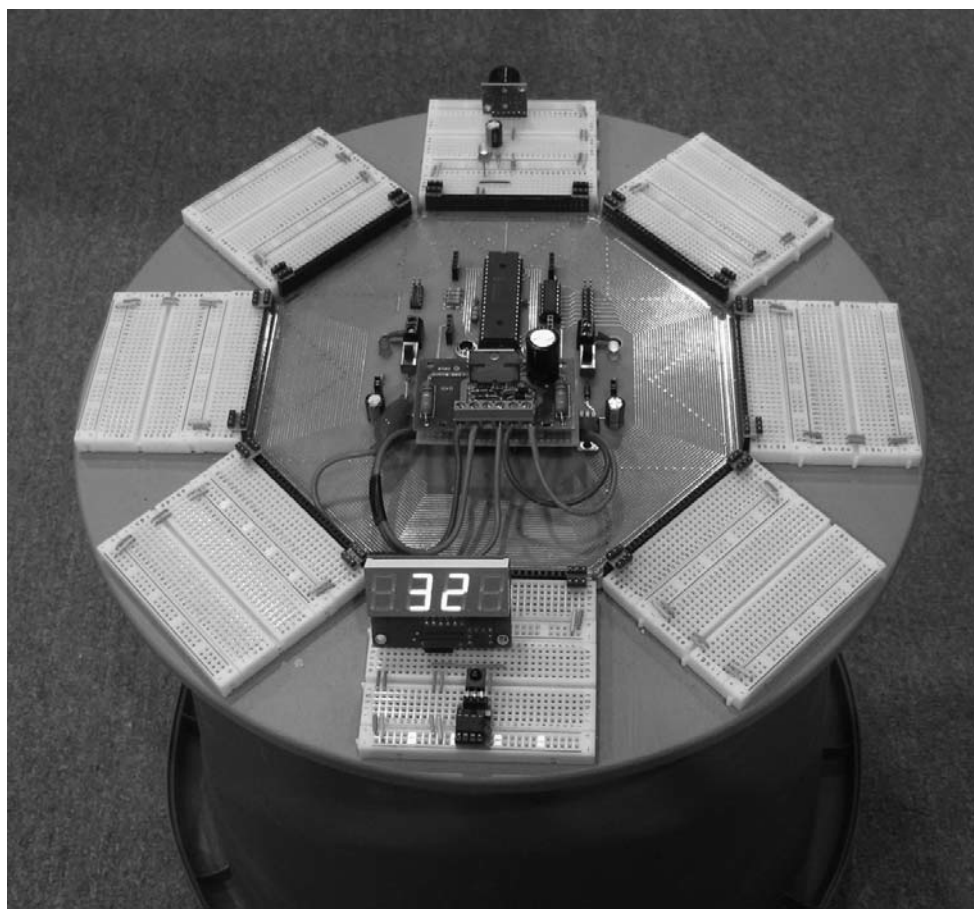


Figure 18-7 Hail, Octavius!

length, it may seem complicated, but it actually contains very little code that we haven't already discussed. Essentially, the program allows us to change Octavius' direction whenever we want, or to stop him and discontinue his exploratory behavior completely. All the while, Octavius is continually updating his ultrasonic range-finding data. The best part is that the program can be easily modified whenever I add new sensory capabilities to Octavius.

HailOctavius.bas can also be easily modified to run on our master processor, so if you decide to build your own 20X2-based robot, it would make a great starting point for exploring the word of PICAXE robotics. Of course, our "Hail, Octavius!" project is just the beginning. Its main purpose is to provide a framework within which to safely endow a robot with additional sensory/motor functions as he grows and develops. We'll discuss some of the many possibilities in the Epilogue, along with what's in store for Octavius as he continues his journey of exploration and development.

This page intentionally left blank

EPILOGUE

What's Next for Octavius?

In robot years, Octavius is still in his infancy; he's just beginning to toddle about, and he hasn't even learned to talk yet! (Although, as we're about to see, that's high on my list of priorities.) Now that I have completed the manuscript for this book, I've been thinking about some of the capabilities that I would like to add to Octavius. The following list comes readily to mind, and I'm sure you could easily make several additions to it:

- **Additional EZ0 ultrasonic ranging sensors:** Just as I began writing this Epilogue, the two additional EZ0 ultrasonic ranging sensors that I recently ordered arrived in my (snail) mailbox. It's been a struggle to continue writing, but as soon as I complete the Epilogue, Octavius and I are going to have some fun!
- **Speech synthesis:** To help Octavius learn his first words, I'll probably use the Devantech SP03 Speech Module. Although I recently read that it may have been discontinued, I happen to have one on hand, so it's my best choice for now. The main function that I have in mind for the SP03 is as a replacement for Octavius' seven-segment LED display. It would provide a much more flexible output device for real-time status reports and other helpful information as we go along.
- **Battery monitoring and status reporting:** So far, I have been quite impressed with how well Octavius' batteries have been functioning. In spite of the fact that I have been spending a considerable amount of time with Octavius lately, his batteries seem to be able to run for weeks at a time without needing to be recharged. Even so, it would be a real convenience if I enabled him to verbally report his battery status from time to time. It's a really simple project (just interfacing a couple of his ADC inputs and voltage dividers with his two batteries), so it won't be long before Octavius will be able to tell me when he's "hungry."
- **Self-charging capability:** Of course, the next logical step would be to enable Octavius to "feed himself" whenever his batteries are running low. My first thought is to implement an infrared beacon that Octavius can use to find his "refueling" station. I'm sure it will require a fair amount of thought and experimentation, but it's a project that I'll definitely tackle before long.
- **Wireless communication link to a Mac or PC:** I have done some experimenting with inexpensive wireless links, and I find them to be frustrating, to say the least. Recently, I began working with the xBee Communication Modules. They are more expensive than the typical 315MHz and 434MHz units that seem to be available everywhere, but they are also much more reliable. Since I'm willing to spend the extra cash to foster my boy's development, that's what he's going to get!

- **Scanning tower:** I haven't quite figured out what I will do with it, but I know that a scanning tower is in Octavius' future. (Did someone say "wireless video link?")

I could go on, but you get the point—Octavius and I have a lot to learn, and we're going to have a lot of fun doing it!

Working with the EZ0 ultrasonic ranging system has made me realize that multiple I/O connections between Octavius' CPU and his peripheral processors can actually be coordinated much more simply than I originally thought. As a result, I have been working on a redesign of Octavius' main logic board that utilizes the PICAXE-28X2 in place of his 40X2 CPU. I think that the 28X2's 22 I/O lines will be more than adequate for motor control and communication with his peripheral processors. The resulting PC board is less than one-third the size of the current prototype board, which will decrease its cost considerably. By the time you are reading this, the redesigned board should be available on my website.

In the meantime, you certainly don't need the power and sophistication of Octavius to get started in the word of PICAXE robotics. All you really need is a 20X2 or 28X2 processor and a couple of breadboards and motors, along with the motor controller circuit we developed in Chapter 17—that and a little creativity and imagination.

With that in mind, and with apologies to Henry Wadsworth Longfellow ("The Children's Hour"; 1860), I'll leave you with this inspiring little snippet…

Between the daylight and the dark,
When observing his processing power,
Comes a pause in mundane tasks
That is known as the Robots' Hour.

I hear in the chamber below me,
The whirring of little wheels,
The sound of computation,
And the pride creation feels.

…and a few final words of warning:

CAUTION **Don't anthropomorphize robots—they hate it when you do that!**

Index

Note: *Italicized* page numbers indicate tables and figures.

O