Junk Box Arduino

Ten Projects in Upcycled Electronics

The Ultimate Arduino-Hacker's Playground Guide

James R. Strickland

Apress®

Junk Box Arduino

Ten Projects in Upcycled Electronics

James R. Strickland

Apress®

Junk Box Arduino: Ten Projects in Upcycled Electronics

James R. Strickland
Highlands Ranch, Colorado, USA

ISBN-13 (pbk): 978-1-4842-1426-8 ISBN-13 (electronic): 978-1-4842-1425-1
DOI 10.1007/978-1-4842-1425-1

Library of Congress Control Number: 2016944327

Copyright © 2016 by James R. Strickland

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Permission granted by ANSI on behalf of INCITS to use material from X.3221-1994. The X3.221 is a withdrawn standard and cannot be referred to as an approved document. Information about this withdrawn standard is provided for historical interest only and should not be used for new designs. All copyrights remain in full effect. All rights reserved.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr
Lead Editor: Jonathan Gennick
Development Editor: James Markham
Technical Reviewer: Terry King and Andrew Terranova
Editorial Board: Steve Anglin, Pramila Balen, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, James Markham, Susan McDermott, Matthew Moodie, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Melissa Maldonado
Copy Editor: Laura Lawrie
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or supplementary materials referenced by the author in this text is open source and available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

Printed on acid-free paper

For Marcia, who's been very patient with little bits of wire everywhere.

Contents at a Glance

About the Author xix

About the Technical Reviewers xxi

Introduction xxiii

How to Use This Book xxv

■Chapter 1: Your Shopping List 1

■Chapter 2: Cestino 29

■Chapter 3: Kick the Tires, Light the Fire 55

■Chapter 4: 8 Bit Ports 81

■Chapter 5: Collector, Base, and Emitter 103

■Chapter 6: TTL: The Missing Link 125

■Chapter 7: Logic Probe 149

■Chapter 8: EPROM/Flash Explorer 161

■Chapter 9: ATA Explorer 203

■Chapter 10: Time Out For a Quick Game 271

■Chapter 11: Z80 Explorer 303

Index 399

Contents

About the Author xix

About the Technical Reviewers xxi

Introduction xxiii

How to Use This Book xxv

■ **Chapter 1: Your Shopping List** 1

Tools and Supplies 1

An Arduino 1

Solderless Breadboards 2

Jumpers (Optional) 3

Hookup Wire 4

A Multimeter 5

Screwdrivers 6

Wire Cutters/Strippers 7

USB to 5v TTL Level Serial Cable 8

Optional: A Universal Programmer 10

Software and Documentation 10

Arduino 1.6.5 or Later 10

ATmega1284p Data Sheet 11

Texas Instruments 74xx00 Data Sheet 11

Calculator 11

Optional: Adafruit Circuit Playground/Electrodroid 11

Parts ... 11
A Word On Resistors ... 12
A Word On Capacitors ... 14
Some Words on Junk ... 16
Parts by Project ... 17
Cestino ... 17
Morse Code Practice Translator ... 19
Larson (Memorial) Scanner ... 20
Transistor Tester ... 21
TTL Tester ... 21
Logic Probe/Injector ... 22
ROM/EPROM/EEPROM Explorer ... 22
ATA Explorer ... 24
Dice Device ... 25
Z80 Explorer ... 26
Postage Is Not Your Friend ... 28
Summary ... 28
Chapter 2: Cestino ... 29
The Stuff You Need ... 30
New Parts ... 30
New or Used Parts ... 30
Anatomy of the Cestino ... 31
The ATmega1284P Microcontroller ... 32
20 MHz TTL Full Can Oscillator ... 35
Placing Components ... 36
Power Circuits ... 38

A Dab of Ohm's Law 41
How It Works 42
Examples 42
LED Circuit 43
Clock Circuit 45
Reset Circuit 46
TTL RS-232 Circuits 48
RTS-Reset Circuit 49
TX, RX, and CTS Circuits 50
Testing, Board Layout, and Static 52
Credit Where Credit Is Due 52
Further 53
■ **Chapter 3: Kick the Tires, Light the Fire 55**
The Stuff You Need 55
New Parts 55
New or Used Parts 55
Software 55
Bootloader and Core 56
Bootloader 56
Core 56
A Little History of Software Abstraction 57
Set Up the Arduino Application 57
ISP: In Circuit Programmer 58
Arduino as ISP 59
How ArduinoISP Works 59
Set Up the ArduinoISP Sketch 60
Set the Programmer and Board in the Arduino Application 60

Wire your ArduinoISP into the Breadboard 60
Preflight Check 62
Programmer Not Responding 64
Device Signature Error 65
Device Signature Yikes Error 65
Could Not Find USBtiny Device Error 66
Burn the Bootloader 66
Blink Pin 1: It's Alive! 67
Or Not So Much: Troubleshooting 69
Programmer Not Responding 69
Wrong Sketch? 70
Power or Clock Problems, Perhaps? 70
ATmega1284P Is Bad? 71
The Morse Code Practice Translator 71
Download the Arduinomorse Library and Install It 73
A Quick Introduction to Object Oriented Programming 73
Hook Up the Hardware 76
The Code 76
How to Use the Sketch 78
How the Code Works 78
Summary 80
Credit Where Credit's Due 80
Further 80
Chapter 4: 8 Bit Ports 81
The Stuff You Need 81
New Parts 81
New or Used Parts 82

A Little Binary With That 82
Counting in Binary 82
Bytes and Words 83
Bit Twiddling 84
The Cestino's Ports and How to Use Them 91
Don't Step on the Ports 92
Port Registers and Commands 93
Build the Larson (memorial) Scanner 96
The Circuit 96
The Code 99
Binary Numbers on Display 100
Further 101
Chapter 5: Collector, Base, and Emitter 103
The Stuff You Need 105
New Parts 105
New or Used Parts 105
Software 105
A Little Semiconductor Theory 105
Electron Orbitals, Bands, and a Dab of Quantum Theory 106
Copper, Revisited 107
Silicon 107
Diodes 107
Transistors, At Last 108
Kirchhoff's Laws and Voltage Dividers 110
Kirchhoff's Laws 110
Voltage Dividers 111
Transistors in Voltage Dividers 113

Build the Transistor Analyzer 113
Construction 114
The Code 117
Credit Where Credit's Due 123
Further 124
Chapter 6: TTL: The Missing Link 125
The Stuff You Need 127
New parts 127
New or Used Parts 127
What TTL Is 127
The Extended 7400 Series Family 128
Types of TTL 128
Why TTL 130
How to Read a Datasheet 131
How to Read Your IC 133
Build the TTL Explorer 134
74xx00 134
74xx92 141
Credit Where Credit's Due 148
Further—More Chips, More Pins, Automatic Configuration 148
Chapter 7: Logic Probe 149
The Need for Speed 149
The Stuff You Need 151
New Parts 151
New or Used Parts 151
Design the Logic Probe 151
Build the Logic Probe 154

Testing the Logic Probe 154
Floating Pins 155
Logic Probe Test 1 155
Full Speed Ahead 157
Debugging Ports with the Logic Probe 158
Binary Numbers for the Logic Probe 158
Credit Where Credit's Due 159
Further: Better Logic Probes and Logic Analysers 159
Chapter 8: EPROM/Flash Explorer 161
The Stuff You Need 163
New Parts 163
Used Parts 163
A Quick Introduction to Hexedecimal 163
ROM, EPROM, EEPROM, and Flash: A Recognition Guide 164
Build the EPROM Explorer 166
Getting On the Bus 168
Power, Ground, and Unused Signals 169
Data Bus 170
Address Bus(es) 170
The EPROM_Explorer Sketch 171
Output 178
Debugging 179
Build the Flash Explorer 180
Power, Ground, and Unused Signals 181
Data Bus 182
Address Bus(es) 182
A Brief Diversion: Bank Switching 182
Control Bus 183

The Flash_Explorer Sketch.. 184
Output.. 198
Credit Where Credit Is Due 200
Further.. 200

■Chapter 9: ATA Explorer.. 203

The Stuff You Need.. 204
New Parts.. 204
Used Parts 204
The Bad Old Days 206
ATA 208
Bit Width.. 209
Endian-ness 209
Anatomy of a PATA Drive 209
Control Signals 209
Registers 210
The Sector Buffer.. 213
Commands.. 213
The Physical Disk(s).. 214
Build the ATA Explorer 215
Hotwire the ATX Power Supply 216
Set Up the PATA Cable and Pins.. 217
Wiring Up 219
The Sketch 220
Preprocessor Definitions 220
Global Variables 224
Low Level Functions.. 225
Testing 228

Utility Functions 228
High Level Functions 237
The Complete Sketch 246
Output 264
Credit Where Credit Is Due 269
Further—Bigger, Faster, More Modern? 270
Chapter 10: Time Out For a Quick Game 271
The Stuff You Need 273
New Parts 273
Used Parts 273
Electronic Dice 274
Driving 7 Segment Displays 275
Time Division Multiplexing 276
Interrupts 277
Configuring Interrupts 278
Interrupt Service Routines (ISRs) 278
Interrupt Vectors 279
Timer/Counters 279
ATMega Timer/Counter Counting 280
ATMega Timer/Counter Actions 280
Configuring Timer/Counters 281
Build the Dice Device 285
The Sketch 287
The Complete Sketch 295
Credit where Credit is Due 301
The Stand-Alone version 301

Chapter 11: Z80 Explorer 303

The Stuff You Need 304

New Parts 304

New or Used Parts 304

Z80 Microprocessor Anatomy 305

Busses 306

Registers 310

The ALU 314

Instruction Decoding and Control Logic 319

Putting It All Together: Operations 319

Objects and Classes, Revisited 321

Pointers 323

Function Prototypes 326

Build the Z80 Explorer 327

Install the Z80 327

Power Circuits 328

Data Bus 328

Control Bus 328

Memory Address Bus 330

The Sketch 330

The Plan 330

The Code 332

The Full Code 356

Output 377

Assembly and Machine Language 381
Program 1: Infinite Loop 382
Program 2: Index Controlled Loop 384
Program 4: Fun with the Stack 391
Credit Where Credit Is Due 394
Further! 395
Index 399

About the Author

James R. Strickland is a professional nerd who writes science fiction, steampunk, technical books, and technical videos. When that doesn't fill his time, he builds retrocomputers, repairs antique radios, programs computers, and is still known to play role-playing games occasionally. He lives in the Denver metro area with his wife, Marcia, and a variable number of cats. He can be found at `www.jamesrstrickland.com`.

About the Technical Reviewers

Terry King has designed broadcast stations, recording studios, broadcast equipment, intelligent machines, and special computer languages for IBM, and has worked as a broadcast journalist covering elections, fires, riots, and Woodstock.

He has taught electronics at SUNY and IBM, and "Bits&Bytes" to many high schools.

Terry received an Outstanding Technical Achievement award from IBM for the software architecture of IBM Chip Test systems.

He is now "retired" and writing about Arduino/Embedded systems (`http://ArduinoInfo.Info`) and running YourDuino.com with his friend from China, Jun Peng, and his library designer wife, Mary Alice Osborne. Since "retirement" Terry has lived and taught in Africa, China, the Middle East, and Italy. Now he is "home again" in rural Vermont and working 40-plus hours a week on ArduinoInfo.Info, firewood cutting, woodworking, and electronics.

Andrew Terranova has been fascinated with electronics since childhood, and fell in love with digital circuit design while working with communication satellites. Today he is an engineer, maker, and writer who is usually found taking something apart or putting something together. Andrew makes robots, electronics, and other fun stuff whenever he gets the chance. His projects and articles have been published in *Make: Magazine and Popular Science.*

Introduction

Admit it: you have an electronics junk box.

Human beings like to hang on to stuff. We get sentimental about the computers we had in high school, or junior high school. If you're in your 20s (or younger) today, you might still have the first computer your parents let you play with when you were old enough to reach the keyboard and the mouse. Human beings haven't yet adapted to Moore's Law, where processing power doubles every 18 months, nor to the idea that everything we buy is already obsolete by the time we buy it. So we sentimentalize equipment that really has very little useful value, and it winds up in the junk box, still good, hopelessly obsolete, useless, but we just can't bear to throw it away or recycle it properly. I've been a computer nerd since the early 1980s. My junk boxes take up about a quarter of my basement.

It's easy to look at the contents of these boxes as frozen magic: tools and arcana from another time and place, distinct from the modern age because the modern equipment is so much more powerful, so much slicker, so much more magical.

There is no magic. There never was. If you've experimented with Arduinos already, this may have dawned on you, at least in a small way. There is no magic. The world runs on electronics and code and the machines these electronics and code control. All of those things, in turn, were designed by people, coded by people, and manufactured by people. If they can understand it, so can you.

Our junk boxes, then, have a purpose in addition to providing ballast for our homes to keep the basement end down and the roof end up. The very fact that the stuff in our junk boxes has no practical value anymore, means we can pull it apart, see what's in it. If we add an Arduino, we can talk to these parts, see what they do, and learn how they communicate. It's fun, and it dispels the magic even further. Once we understand how the old things work, we really are most of the way to understanding the modern, and to putting modern equipment in its proper place, as tools, as entertainment, as machines that do our bidding.

That's what this book is about.

How to Use This Book

The first thing you need to do in order to use this book is to read it. Seriously. Read each chapter all the way through, preferably before you sit down with the parts in front of you to put something together. Pay close attention to The Stuff You Need part of each chapter, as it lists the tools and parts you need for each project. I've burned up more electronics trying to work around a missing tool or part than any other single cause.

Please also read the chapters in order. Some of the projects may look like respins of hoary old Arduino projects, like the Larson (Memorial) Scanner. "Come on," you might say, "I've done that." Please do it again anyway. I use these basic projects to demonstrate powerful techniques and concepts that the classic versions and traditional Arduino programming stop short of. Each chapter assumes that you know the stuff in the previous chapter, that you got that project to work, and that you're comfortable with how it works.

At the end of each chapter is a section called "Further." In these, I talk about how the project might be expanded into something more complex. These are suggestions. Sometimes I'm working from directions others have taken similar projects, sometimes I'm speculating into the clear blue sky. The important point is that I haven't tested the ideas in the Further sections, and you'll be on your own figuring out if the ideas are even practical, designing the circuits and making them work.

Finally, I do make an assumption in this book: that you are already familiar with Arduinos and the Arduino software. We'll be using both. If you haven't touched an Arduino before, please stop here, flip ahead to Chapter 1, and get one of the 5 volt Arduinos that I recommend. Then hook it up, get the Arduino software working, and play with it. There are hundreds of tutorials online, and they'll get you going with the basics. Once you've got that going, you'll have what you need for this book.

CHAPTER 1

Your Shopping List

This chapter is, in essence, a shopping list of the tools, parts, and software you'll need to do the projects in this book. Most of the parts are common, and you probably have some in your junk box. A few aren't worth getting anywhere else. I'll list the kinds of gear that has the parts we need, but these lists are certainly not the only places to look.

Some parts are best purchased new, and I'll list sources for them. You should know that I live in the continental United States, and my usual suppliers are nearly all in my country, which saves me postage. I've taken care to name only suppliers I've dealt with personally or large-scale operations I would buy from without trepidation.

If you are not in the continental United States, you can still use the websites that I list to look at the parts, but it will probably be cheaper for you to use suppliers in your own country to deliver them.

As for tools, if you have some or all of the tools on this list, great. No need to buy new ones. If not, I'll go into how to pick good ones without breaking the bank.

All the software and information listed are free, and many are open source.

Tools and Supplies

Science officers from certain starships may be able to build computers from stone tools and animal hides, but I'm here to tell you that it's easier with the right tools and supplies.

An Arduino

We'll be building an Arduino-compatible device called the Cestino. (That's Italian for recycling bin.) To build it, ironically, you need an Arduino. I developed the Cestino with an Adafruit DC Boarduino and an Arduino Mega but, except for the Leonardo, nearly any Arduino or any fully compatible clone that can handle 5v circuits will do.

If you don't already have an Arduino, then there are some questions to consider. First, can you solder? Second, do you want to be able to use Arduino shields, which are add-on boards that require the standard Arduino pin arrangement?

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-1425-1_1) contains supplementary material, which is available to authorized users.

© James R. Strickland 2016

J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_1

The Cestino will not have the standard pin layout, and it cannot be used with shields without a lot of jumpering and reworking of code.

If you can solder, and you don't care about shields, I strongly recommend Adafruit's USB Boarduino kit. Once you solder it together, it can plug directly into your breadboard, and you can jumper it to the Cestino's breadboard sockets with ease. These are available from Adafruit (`http://www.adafruit.com`).

If you don't want to solder, or if it will save you a bundle to order in your home country instead of overseas from the United States, but you still don't care about shields, then the Arduino Nano will do all the same things as the Boarduino, and they come preassembled. They should be available at the usual Arduino (now Genuino) distributors, which are listed here: `https://www.arduino.cc/en/Main/Buy`. Make sure that the one you buy comes with its pins already soldered.

If you do care about shields, I suggest an Arduino Mega 2560 or an Arduino Uno R3. Like the micro and nano listed earlier, they should be available at the usual Arduino distributors. They are also the most commonly cloned boards, so non-Arduino-branded alternatives may be cheaper and easier to obtain.

Solderless Breadboards

All the projects in this book, as well as the Cestino itself, will live on solderless breadboards (Figure 1-1). If you've not encountered these before, they consist of a plastic case drilled with holes every 1/10th of an inch (2.54mm) surrounding a central trough. Each row of holes is connected together under the plastic case so that an IC plugged into the central trough has several holes beside each pin so you can plug wires or other components in and connect them to the pin.

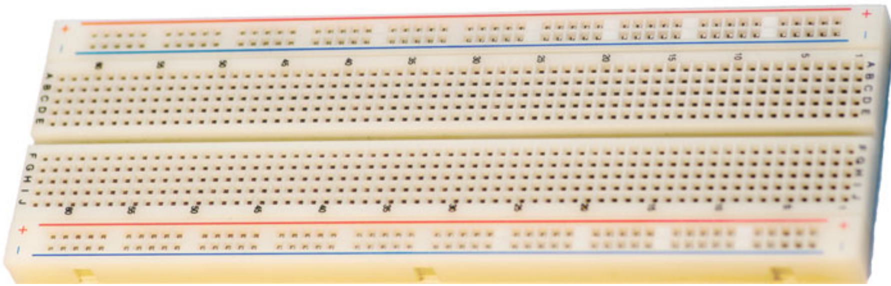

Figure 1-1. *Solderless Breadboard*

The most common size of solderless breadboard is 6.5 inches by 2.1 inches (165.1 mm x 54.36 mm) with 830 tie points (holes.) You'll need two of these, preferably the kind that can snap together along the long edge. While the pin spacing is standard and the power busses along each edge are usually standard, the connectors to snap two or more boards together are anything but. Make sure that both boards are the same brand if you have to order them.

The only extra feature on these boards that's really useful is to have the pin rows numbered. It's not absolutely essential, but you'll be working with ICs with 40 pins on a side at times, and IC makers love to put the power and ground connectors right next to each other. For sanity's sake, I urge you to pay the tiny amount extra it costs to get labeled boards. You can also order a pair of these breadboards connected together with a metal backing and nice rubber feet to keep them from sliding off your workbench, with or without a built-in power supply, but these are much more expensive.

Solderless breadboards are available at Mouser (`http://www.mouser.com`), Digikey (`http://www.digikey.com` (although it's harder to find the cheap ones there), Adafruit, and so on. They're a standard tool for prototyping circuits, so your usual electronics suppliers should have them. You might consider shopping surplus dealers like MPJA (`http://www.mpja.com`) as they sometimes have solderless breadboards considerably cheaper. It's conceivable that old boards from the junk box might have corrosion in the tiepoints that would cause problems, or that the board has been used so much that the tiepoints are loose. They do wear out.

Jumpers (Optional)

A jumper is, in its most basic form, a piece of wire with two bare ends for connecting tiepoints of your breadboards together or to connect tiepoints to sockets on your Arduino.

You don't actually have to buy jumpers like the ones in Figure 1-2. You can make your own out of hookup wire. When you get to the chapter on building the Cestino, you'll see I did exactly that for the Cestino's wiring, mostly to make it neat and easily discernible from the project wiring in the photographs. But make no mistake. Stripping each jumper on both ends is *tedious*. I recommend a set of premade jumpers with hard connectors at each end for easy grabbing. These are available at Mouser and Jameco, but they're very expensive there. Adafruit has them at a much more reasonable price. Most hobbyist electronics shops sell these as well. The more colors they come in, the better. If you're buying them, you want male-to-male jumpers.

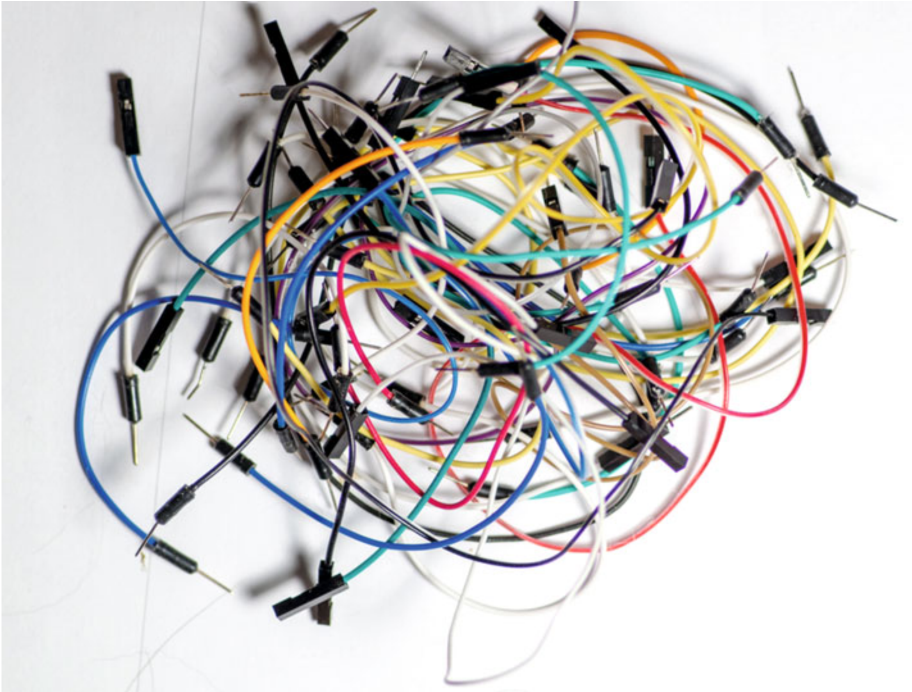

Figure 1-2. *Jumpers*

Jumpers wear out.

Maybe I abuse my jumpers, pulling them out from the middle instead of the molded connectors on each end, but I had several jumpers turn up bad during the course of developing these projects and the Cestino itself, and spent several frustrating hours debugging software problems that turned out to be bad jumpers. If your jumpers have been around a while, and something just doesn't seem to work, test the jumpers for continuity.

Hookup Wire

You need hookup wire (Figure 1-3). I recommend using it for the standing wiring of the Cestino, to keep it out of the way, if for no other reason. It also looks nicer. Also, sometimes you need a jumper of an unusual length, and having some hookup wire to make your own is very handy. There are only a couple of important specifications.

Figure 1-3. *Hookup Wire*

The wire must be 22AWG—gauge—a.k.a. 0.644 mm. The wire must be solid core. That is, it must have a single bendable copper wire inside, not a lot of little strands. They do, I'm told, make prebonded wire, which is stranded wire tinned along its length with solder. I've not tried it yet, but I'm told it makes fine, long-lasting jumpers. The bottom line is that the wire has to be rigid enough to push into the tiepoint reliably.

You'll need at least two different colors of wire, black and something else. If your electronics supplier has a multicolor pack, I'd strongly urge you to get that. It's easier to keep data lines separate from power, ground, and reset when they're different colors.

Nearly any electronics hobby shop will have this kind of wire. In quantity it gets heavy, so I don't recommend mail-ordering it, but if you must, I suggest Microcenter (`http://www.microcenter.com`) for the six-color kit from Elenco (I have this), or the similar kit from Adafruit.

A Multimeter

A multimeter (Figure 1-4) is a handy way to see inside a circuit and look for basic functionality. You need one. If you already have one, great.

Figure 1-4. *An Inexpensive Digital Multimeter*

I have a bunch: a Radio Shack folding analog meter from the early 1990s, a $70 Elenco digital multimeter from about 2004, and a new $11 MCM digital multimeter I picked up at my local Microcenter. For the purposes of these projects, pretty much any multimeter will work, but you do need to make sure that the resistance tester doesn't put more than about 5 volts into the circuit being tested, or you can fry sensitive components. One mega-ohm of impedance should be enough—more is better. None of the circuits we're working with are especially high impedance themselves. (Impedence is the AC version of resistance. For some components, it changes over frequency. At the frequencies we're dealing with through most of this book, the difference doesn't matter.)

If you don't already have a multimeter, there are some considerations you should make before buying. The traditional view of tools is that if you buy good tools once, you can use them for a lifetime. That's still true, but it's expensive. If you want to go this route, head to `https://www.youtube.com/watch?v=gh1n_ELmpFI`, EEVblog's Digital Multimeter Buying Guide. Watch the whole thing. I learned a lot when I did. Good, well designed, safe meters start at about $70, and the sky is the limit.

Really, we're dealing with low power circuits. The maximum voltage any of the projects in this book should ever see is 12v, and that's only in chapter 9, the ATA Explorer, to power the hard drive spindle motor. As long as you promise never to plug your meter into wall power, no matter how much the documentation says it can handle it, a cheap meter will do the job just fine.

The specs it must have are: at least 1 mega-ohm of impedance, so you can peek at the voltage of a logic line without changing it; a resistance/ohms setting that doesn't throw out more than 5 volts (less is better); a continuity beeper, that beeps at resistances of a few tens of ohms; and it needs to take normal batteries, such as double or triple As or a standard 9-volt battery. Cheap meters tend to eat batteries, so their batteries should also be cheap.

Doing a web search right at this moment, I found meters that look promising at Harbor Freight (`http://www.harborfreight.com`) and in my local Walmart, (`http://www.walmart.com`), but if you want to spring for a better one for not much more money, I'd look at Sparkfun (`http://www.sparkfun.com`) or Adafruit. The latter two can probably tell you what the impedance and ohms testing voltage are, whereas the former are unlikely to know.

Screwdrivers

There are a couple of uses for screwdrivers in the projects in this book. The one that makes them a requirement is (gently) prying ICs off the breadboard when you're done with them, so you'll need a small, flat-bladed screwdriver for that. Disassembling junk, by contrast, may require quite an assortment of screwdrivers. If you're buying new, most hardware stores sell inexpensive assortments of screwdrivers at a discount price, with a lifetime warrantee. If there's no warrantee, it means they've made them out of such cheap steel that they'll bend or strip the head of the screw the first time you use any force on one. I really recommend against buying these online, as they are quite heavy, but if you insist, Home Depot has a six-piece set in their Husky brand. I have a couple of those (`http://www.homedepot.com`). As you're probably expecting by now, Adafruit has screwdrivers, too.

Wire Cutters/Strippers

Unless the car has one wire to splice and you will be eaten by plague bunnies *right now* if you don't get the thing running, don't use a pocket knife to cut and strip wire. It's bad for the knife, and worse for your thumbs.

Wire cutters and strippers (Figure 1-5) are the right tool for the job. Fortunately, after screwdrivers, pliers, and duct tape, they're the most common tool on Earth. You probably have at least a couple pairs of wire cutters already. Any of them will do the job for cutting.

Figure 1-5. *Wire Cutters and Strippers*

Stripping wires is a different question. I've been known to strip wires with my wire cutters in a pinch, and for thick wires it works ok. On 22 AWG wire, not so much. If you nick the copper inside the insulation, the wire will break after only a few bends. You want some kind of wire stripper. The simplest are a cutter with a notch sized to the copper itself. More complicated ones cut the insulation and remove it in one motion. Pick the kind you like.

A further consideration is that wire cutters and especially wire strippers wear out. While we're not asking a lot for even dull cutters to cut 22 AWG wire, the wire stripper needs to do the job smoothly and neatly or it defeats the purpose. I wore out the (really cheap) wire cutters and strippers in Figure 1-5 in the process of writing this book.

If you're buying new, I suggest getting separate wire cutters and strippers. For the wire cutters, flush cutters are a good choice. They're cheap, sharp, and when you move on to printed circuit boards, you'll need them anyway. Wire strippers come in a myriad of shapes and sizes, so assuming that they have a setting for 22-gauge wire, pick the kind you like. Mouser and Digikey carry both, as do Adafruit, Sparkfun, and so forth. Name brands cost more and usually last longer.

USB to 5v TTL Level Serial Cable

You need an FTDI USB to 5v TTL level serial cable (Figure 1-6). What, you ask, is that? Read on.

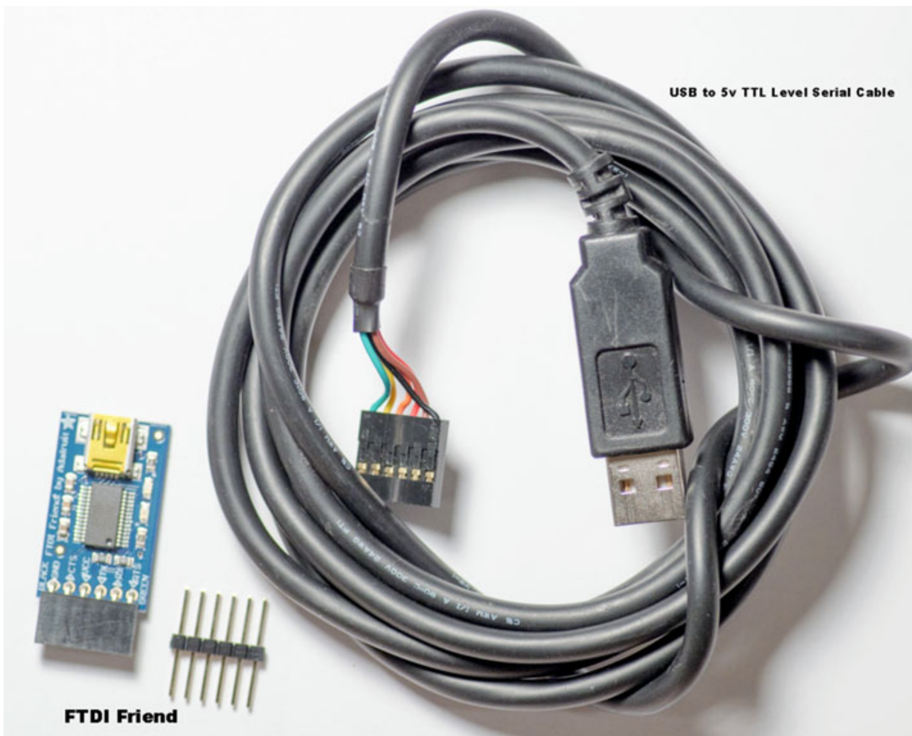

Figure 1-6. *USB to 5v TTL Level Serial Cable and FTDI Friend*

The earliest Arduinos communicated with their host computers over RS-232 serial. RS-232 is a very old protocol, dating back to 1962. It's the kind of serial line we used to connect terminals to minicomputers with, and in more recent times we connected modems to computers with them. The ATmega and AVR micro-controllers used in Arduinos usually have the hardware to generate these signals built in. RS-232 is slow, but it's simple, and fairly robust, not least of which because its signal voltages are as large as +15 volts for high and -15 volts for low. Therein lies the first of our problems.

The ATmega 1284p microcontroller, like most of its ATmega and AVR relatives, operates at about five volts. The voltage on any input can be no higher than 1 volt above the operating voltage. If we feed raw +-15 volt RS-232 to any pin on the ATmega 1284p, the result will likely be smoke and a damaged or dead microcontroller. Even if it could handle the voltages coming in, the computer on the other end is expecting +-13 volts (the most common real voltages in PC RS-232 ports), and the signals are coming out of the microcontroller are inverted anyway.

The solution to this problem is a level shifter, either as a dedicated IC, or a network of transistors and capacitors that shifts the signals up on transmit and down on receive. The original 2005 Arduino used this latter method, and if the Cestino were communicating to the host computer over RS-232, the Cestino would need a level shifter, too. It doesn't have one because of the second problem.

RS-232 ports are rapidly fading into extinction. Macintosh computers were the first to be too cool for RS-232, but the venerable ports have largely vanished from laptops and notebooks, and were never present on tablets. They're still occasionally found on desktop PCs, although they may not be wired right, and boards to add them are becoming few and far between. The world has gone to USB. Sadly, the ATmega-1284p doesn't speak it.

One thing I've noticed in the time I've been tinkering in electronics: when there's a widespread problem or group of problems, someone will make an IC that solves it or them. In the case of translating inverted TTL level RS-232 signals to USB, there are several. The most common is made by a company called FTDI, and they make a cable that plugs into your USB port, powers the internal conversion chip, and generates inverted, 0-5v ttl level RS-232 signals that the microcontroller can understand. Because USB also provides power, in the form of a +5 volt line, which is exactly what the microcontroller needs, the Cestino is powered by USB as well. This, then is the USB to 5v TTL Serial cable you're looking for. But there are other solutions.

Adafruit gets mentioned a lot. They're one of the North American distributors for Arduino, and they have these cables. They also have a little board of their own design they call the FTDI Friend, which does the same thing, but isn't built into the cable. Better still, the FTDI friend comes with a set of extra-long pin headers for plugging into your breadboard and then plugging into the FTDI friend. It also comes with LEDs tied to the transmit and receive lines so you can see when the Cestino is communicating.

If you already have an FTDI Serial TTL-232 cable, as I do, you can either get a strip of protoboard and solder two sets of pins to it, connected together in pairs and use this to plug the cable into your breadboard, or you can order the extra-long pin-headers from Adafruit or Schmartboard (`http://www.schmartboard.com`) and plug them in without soldering. I bent mine to a right angle so the connector wasn't quite so precarious.

There are two flavors of FTDI cable: the 5-volt version and the 3.3-volt version. You need the 5-volt version. The FTDI Friend is, by default, set up to provide 5-volt power and 3.3-volts on the communication lines, which is fine.

If you're buying new, get the FTDI Friend. These can be had at Adafruit, of course, and second-sourced at Evil Mad Scientist (http://shop.evilmadscientist.com). Similar boards can also be found on the Arduino store (https://store.arduino.cc/) and Seeed Studio (http://www.seeedstudio.com). These do not, however, come with the extra-long pin headers. One could certainly unsolder the pin sockets and solder normal pin headers to these boards instead. If you by the FTDI Friend or similar board, you will also need a standard USB cable with a Mini-B 5-pin connector on one end.

If you want the more traditional cable and you're okay with figuring out how to connect it to your breadboard, Adafruit has them, Sparkfun has them. Mouser has them for about twice the price.

Optional: A Universal Programmer

A universal programmer, also known as a ROM burner, allows you to place programmable ICs in it, and program them. It will not work directly with the Arduino software, and if everything goes smoothly you shouldn't need one for any of the projects in this book. (Although having one does make the EPROM explorer a lot more fun.)

I'm including it here because once in a while the bootloader installation process goes off the rails, and leaves the ATmega1284P "bricked"—caught in a catch-22 situation where it's configured wrong and can't communicate. Being able to reset the ATmega to its factory state will fix it, and a universal programmer is the tool for the job. If you have one, and it can program the ATmega1284P, you're good to go. If you don't have one, and you'd like one, make sure that the one you're looking at can also program the ATmega1284P, among other things. Universal programmers can cost hundreds or thousands of dollars from companies you've heard of in electronic supply houses.

By contrast, I have a MiniPro TL866 universal programmer that I got directly from China on Ebay. It cost about $60 plus shipping, and I've been quite satisfied with it. I should mention that the TL866 only has windows software, but it runs nicely in VirtualBox under Windows 7 and 10. There is also open source software available for the TL866 (https://github.com/radiomanV/TL866) but I've not used it at any length.

Software and Documentation

This book is mostly about hardware, but it's software and documentation that make the hardware useful. This isn't a comprehensive list of either one—there are more things to download in the projects themselves—but here's what you need to get started.

Arduino 1.6.5 or Later

You need the Arduino software. We will be using Arduino 1.6.5. Later versions should work unless they change the configuration file layout *again*. The best place to get it is https://www.arduino.cc/en/Main/Software. If you already have Arduino installed, that's fine, but make sure it's version 1.6.5 or later. It will say in the top bar of the Arduino window.

■ **Note** Linux users, your Linux distribution may provide a version of Arduino, but particularly if your distribution is based on Debian, it's probably way out of date. I strongly encourage you to go to the Arduino site listed earlier and download the current version. Versions prior to 1.6.5 won't work with the Cestino.

ATmega1284p Data Sheet

You need the datasheet on the Atmel ATmega1284p. It's here, as of this moment, on Atmel's Site: (`http://www.atmel.com/images/doc8059.pdf`).

Big companies do occasionally reorganize their websites, so if you can't find it, do a web search on ATmega1284p data sheet. I strongly suggest downloading and saving a copy, as with all the datasheets you download, but it's 372 pages long, plus appendices, so for now, you only need to print page 2, where the pinout diagrams are.

Texas Instruments 74xx00 Data Sheet

We'll use the 74xx00 quad-nand gate IC in a project, but it's also used in discussions of how to read datasheets. The TI datasheet is well written and well organized, and while there are lots of 74xx00 datasheets out there, the TI one is the one to get. It's available on Texas Instruments' website: `http://www.ti.com.cn/cn/lit/ds/symlink/sn74ls00.pdf`. It has a part number of SDLS025B.

Calculator

Any calculator will do. If it is a scientific or programming calculator that can translate in and out of binary, it might save you some work. The best of these run as applications on your computer, or your phone.

Optional: Adafruit Circuit Playground/Electrodroid

Although it's not necessary, if you have an IOS (Apple) smartphone, Adafruit's Circuit Playground is free and extremely useful for calculating circuits, working out resistors, and so on. I'll tell you how to do these all by hand, but this isn't school, so you can use an app if it's easier. For Android users, there's Electrodroid, which appears to do many of the same things.

Parts

Parts for these projects can come from a wide range of sources. New is fine, used parts are usually okay. Some parts aren't worth harvesting used.

A Word On Resistors

A resistor is a device that conducts electricity but not very well. As a result, it converts some of the energy passing through it into heat. On the face of it, it doesn't seem like resistors would be very useful, but they are the most common electronic component of all. They come in a wide array of packages, but the most common are axial lead resistors with the values painted on, like the ones in Figure 1-7.

Figure 1-7. *Resistors*

Thanks to Ohm's and Kirchoff's laws, we can use resistors to reduce the voltage of a given circuit. We can use them to limit the current of the circuit. More miraculously, we can raise the voltage of a circuit relative to ground by putting a resistor between the circuit and ground. We'll get into Ohm's and Kirchoff's laws in Chapters 3 and 5, and these applications will make more sense, but for now it's important to understand that resistors deal in ohms (resistance) and watts (energy or heat).

Resistors are also cheap. Really cheap. Resistors are so cheap that they're not worth harvesting from junk, and even if your junk box has loose resistors that are old, with badly corroded leads, you might want to consider replacing them. In any case, when dealing with resistors, it's a good idea to switch your multimeter to its ohms (Ω) setting and make sure the resistor is reasonably close to its advertised value.

Resistors are measured in ohms, named after the german physicist Georg Simon Ohm, who wrote Ohm's law. This is the symbol for ohms: Ω.

The resistor on the left is a 1 watt resistor. The one on the right is a 1/4 watt, which is the type we'll be using for these projects. They're the same resistance value, but the one on the left can dissipate four times more energy as heat before it burns up than the one on the right.

■ **Tip** Approach the full wattage ratings of resistors with caution. It's a good idea to rate your resistors at twice the wattage you expect them to face.

Figure 1-8 is a handy table showing how you read the value of a resistor.

Color	Ones/ Tens	Powers of 10		Tolerance
Black	0	$10^0\Omega$ =	1Ω	
Brown	1	$10^1\Omega$ =	10Ω	1.00%
Red	2	$10^2\Omega$ =	100Ω	2.00%
Orange	3	$10^3\Omega$ =	1000Ω (1kΩ)	
Yellow	4	$10^4\Omega$ =	10kΩ	
Green	5	$10^5\Omega$ =	100kΩ	0.50%
Blue	6	$10^6\Omega$ =	1000kΩ (1MΩ)	0.25%
Violet	7	$10^7\Omega$ =	10MΩ	0.10%
Grey	8	$10^8\Omega$ =	100MΩ	0.05%
White	9	$10^9\Omega$ =	1000MΩ=1GΩ	
Silver				10.00%
Gold				5.00%

Figure 1-8. *Resistor Color Codes*

Looking at the resistors in Figure 1-7, you can see there are four color bands. Some resistors have five bands, but those are not very common. From the top, we have an orange, a second orange, and a brown band. If you look at the table in Figure 1-8, you see that the first two bands always indicate the numeric value, so orange orange is 3 and 3. The third band always indicates the power of ten to apply to the first two bands. Our third band is brown, which means 10 to the first power, or 10. 33x10 is 330 ohms. A 470k (thousand) ohm resistor would be yellow, for 4, violet for 7, and yellow for ten to the fourth, or 10,000, giving us a total of 47x10,000 or 470,000 ohms. And so on. The maximum this notation is capable of would be a resistor with three white bands, 99 times 10 to the 9th power, or 99 giga-ohms. By the way, a black band is always zero, so black-brown-black would be 0 1 times 10 to the zeroth power, which is 1, thus, 1 ohm.

The fourth band is the tolerance, that is, how close to its rated value is this resistor required to be. The widest tolerance is 20 percent, which means our 330Ω resistor might go range from 264Ω to 396Ω. These won't have a fourth band at all. A silver band means the resistor is a 10 percent tolerance, so from 297Ω to 363Ω. Gold is 5 percent – 313Ω to 346Ω. Bear in mind that the price of resistors goes up dramatically from 0.8 cents each (in bulk orders of 5000) for 5 percent, quarter-watt, 330Ω resistors, to 1-7 cents each (in bulk orders of 5000) for 1 percent to 61 cents each (bulk orders of 5000) for .1 percent, which is as close a tolerance as Mouser will sell me. There are color codes for these too, brown and purple, respectively. Five to ten percent is fine for our purposes. I've chosen resistors that give us good, middle-of-the-road values in the circuits they're in. If we're ten percent high or low, the circuits will still work, and for digital applications rounding up to the nearest standard value of resistor will be fine, so if your calculations say you need 320Ω, use a 330Ω resistor.

I've called out what resistors you *must* have for each project, but the truth is, I recommend buying resistor kits if you're just starting out. These are assortments of resistors, and they're cheaper that way. The important specifications are: through-hole mounting (because surface mount won't do any good on a breadboard) and 1/4 watt or 250mW (milliwatts – thousandths of a watt), mostly because anything else is either more expensive than you'll need for digital electronics or too small to see easily.

Spark Fun has 1/4 watt resistor kits, (`https://www.sparkfun.com/products/10969`) as does Amazon. I got mine at Radio Shack, back in the day.

A Word On Capacitors

Capacitors are simple, passive electronic devices that store electrical energy in the form of an electrostatic field. Literally, the charges are stuffed into the dielectric between two conductors, and once the capacitor charge equals the voltage of the charging circuit, the capacitor will cease to conduct until something changes—either the charging voltage goes higher, or lower.

Capacitors are very useful for separating alternating signals from direct current levels. If a capacitor is charged to the level of DC in a given wire, but an AC voltage is superimposed on the wire, only the AC voltage will emerge from the other side of the capacitor. If, instead, you want to keep AC out of a circuit, you put a capacitor from the circuit to ground, and most of the AC in the circuit will be shorted to ground. We use capacitors in both these ways in the Cestino build.

Capacitors are measured in Farads, technically, but a Farad is an enormous amount of capacitance and is not common in the wild. Usually in digital electronics, one sees microfarads—μF, and picofarads—pF. Occasionally things will be labeled in nanofarads (nF), and if your schematic is from before 1960, when the international system of units was adopted, you might see micro-microfarads (μμF). A nanofarad is 1000pF, and a micro-micro-farad is a picofarad. In Australia, a picofarad is sometimes called a puff.

Like resistors, capacitors are extremely common, cheap components that you see everywhere in electronics, in a wide variety of formats (Figure 1-9). Like resistors, they are not worth harvesting from old equipment. Although capacitors may also be labeled with color coding, it's much more common for them to be stamped with a numeric value.

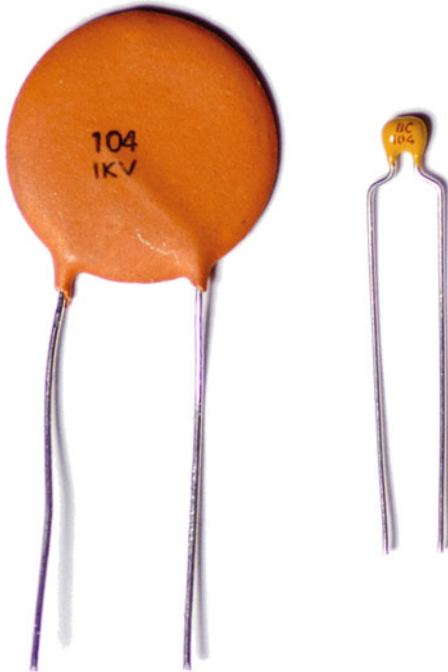

Figure 1-9. *Capacitors*

The capacitors in Figure 1-9 are both .1μF capacitors. The one on the left is a ceramic disk, rated at a thousand volts, the other is a tantalum capacitor rated at 50 volts.

The way you read this notation, 104, is the first two digits are the value, and the last digit is the number of zeros after it, in picofarads. So 104 is 10 times 10,000, or 100,000pF, aka 0.1μF.

Not coincidentally, 0.1μF capacitors are the only type we'll be using in the projects in this book, usually to short AC noise to ground, although there's one in the reset circuit of the Cestino as well.

If you're buying new, I suggest getting a few dozen 0.1μF 50v tantalum capacitors. They're tiny and easy to find room for.

Some Words on Junk

So if resistors and capacitors aren't worth harvesting, what is? And what kinds of junk should you look in?

The best kind of junk for these projects dates back to the 1980s-1990s. In boards of that vintage, most of the chips are still of the through-hole variety—that is, their pins go all the way through holes in the board and are soldered on the back. Quite a few will be socketed, where they can be removed with a screwdriver. If you can find this vintage of equipment at yard sales and thrift stores, they're the way to go.

As the years wore on and surface-mount (SMD) electronics and large scale integrated circuits (where more and more functions were packed in fewer and fewer ICs) became the norm. Through-hole ICs became fewer and further between. Nowhere did this happen faster than in computers.

Computers aren't the only game in town, though. As the computer revolution spread, older (cheaper) ICs and through-hole manufacturing were—and are still—often used in other devices, like thermostats, car electronics, VCRs, and so on. These can be goldmines of old parts that are worth having. There is a catch, of course, and that is that you have to unsolder the parts. There are lots of tutorials online, but the upshot is it requires patience, a soldering station (so you don't overheat the IC) and a hand-held, spring-powered solder-sucker. There are tutorials online for extracting ICs from old boards.

If you intend to do this kind of work on a regular basis, you might want to invest in a powered desoldering tool like the Hakko FR300. I have a Hakko 808, the older model, and a stash of spare parts for it. I wouldn't go back.

The components I've called out are, for the most part, common as dirt (ROMs, 7 segment LED displays, tactile buttons, speakers, individual LEDs, ATA Hard Drives, etc.) and most are readily available new from big name electronics supply houses (Mouser, Digikey, and so on.) If you don't have one, you might have to hunt around online, or ask friends for old ATA drives, as they're pretty much extinct, and certainly not worth buying new. The same goes for old ROMs/EPROMS. You could certainly buy new EPROMs (EEPROMS will also work) but unfortunately they ship blank, and the project as designed won't write to them. You'd be better off digging the BIOS chip out of an old PC motherboard, even if it's a modern 4 or 6 pin type, and reading up on how it communicates. If it speaks SPI, a serial protocol, you're in luck. The ATmega1284p has SPI hardware built in. The other possibility is that it speaks I2C, another serial protocol, for which there is an Arduino software library. This book is about tinkering. Don't be afraid to tinker. This stuff is, after all, junk, destined for the recyclers.

If you really can't find the parts for a given project, the important thing is to understand what that project is doing.

When you do find junk that seems like a good candidate, the first thing to do is look up the datasheet on that part. Most parts, especially as you reach back in computing history, are standard, and the datasheets are still around. You shouldn't have to pay for them, either. If the part is soldered, you might want to look at the datasheet first, before you go to the trouble of unsoldering it. Parts designed for anything other than 5 volts won't work for these projects.

Once you have the datasheet, save it. Datasheets have the habit of disappearing off the internet as a part goes more and more obsolete, or as companies get sold and resold and these ancient parts that haven't been in production in years are deemed not worth adding to the new company's website.

Parts by Project

If you are in the electronics store looking at this book and wondering what parts you need, project by project, you're in the right place. The same goes if you're shopping online.

Cestino

The Cestino, star of this book and all the projects in it, is a mostly-Arduino-compatible experimenter's system, built on a breadboard. In order to build one, you will need the components shown in Figure 1-10.

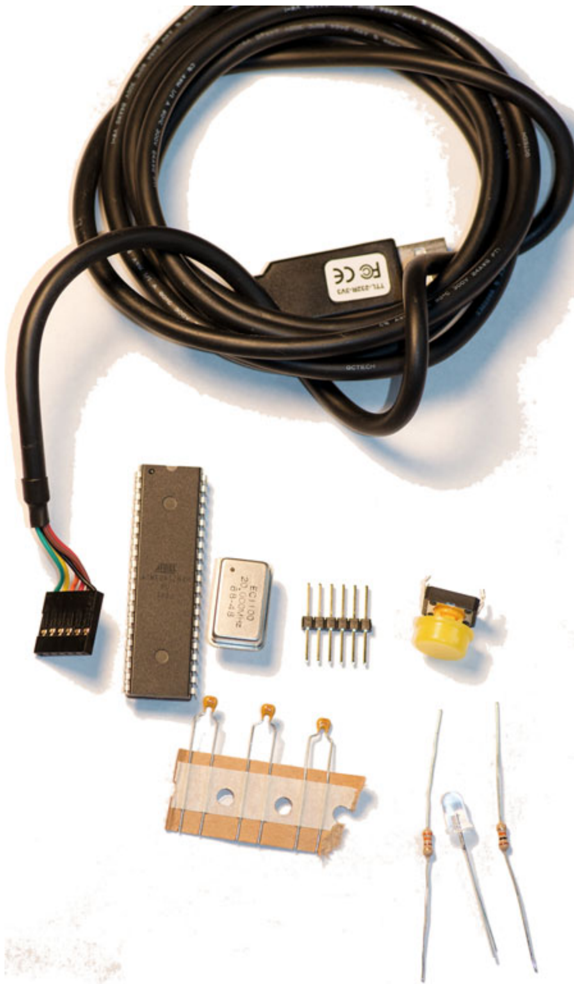

Figure 1-10. *Cestino Parts*

New Parts

- One Atmel ATmega1284P sometimes called the ATmega1284P-PU microcontroller. You want the PDIP/40 pin DIP or through-hole version.

These are available at Mouser, Digikey, and Futurelec (`http://www.futurlec.com`). Futurelec's shipping tends to be leisurely, but they usually have the best prices and a very wide selection. Amazon has them as well, but they're charging a more than most other suppliers. These ICs seem to be quite robust, so only get a spare if you want one.

■ **Warning** Some sellers also list the ATmega1284, sometimes known as the ATmega1284-PU. (Note the missing P before the hyphen.) It's not quite the same microcontroller, and the Cestino firmware will not load on the ATmega1284. I have two of them that don't work to prove it. The PU in both ICs refers to the PDIP (plastic dual in-line pins) package rather than the pico-power capabilities of the micro controller, as in the ATmega 1284P-PU

- Four 0.1uF capacitors, any type. See: A Word on Capacitors.

One of these couples the reset circuitry to the TTL-RS232 cable to keep DC levels from being passed. The others are decoupling/bypass caps to keep AC noise out of the IC's power supplies.

- One 10kΩ resistor. See: a Word on Resistors.

This is the reset pull-up resistor.

- One 330Ω resistor. See: a Word on Resistors.

This is the current-limiting resistor for the LED. (If your LED is a real weirdo, you may need to resize this. My board has a 220Ω resistor because I chose a very bright LED that has a higher voltage drop. All this is explained in Chapter 3.

- Hookup Wire. See Tools and Supplies.

New or Used Parts

- One 20MHz Full Can TTL crystal oscillator

This is the Cestino's clock. Without it, the Cestino won't do anything. Any brand is fine, and Mouser, Digikey, and Futurelec all have them for about the same price. Make sure you get the full can type as shown in the photo. I had two of these in my junk box. The one shown in the picture is the one that works. They're not expensive. You might want a spare.

- One Tactile button, momentary-on (only on while pressed).

Mine have big flashy plastic buttons on them, but it's not a requirement. As always, you want a through-hole mounting. This is the reset switch for the Cestino. If you've looked ahead, there's another one just like it used for the Morse Code Translator.

- One USB to 5v TTL level Serial Cable/board/whatever. See Tools and Supplies.

This can't be shared with your existing Arduino, as the Cestino gets its power from this cable.

- One Arduino. See Tools and Supplies.

Used to burn the bootloader onto the Cestino.

Morse Code Practice Translator

The Morse Code Translator is a simple project rolled into Chapter 3, wherein we test the Cestino to make sure it works properly. Its parts are shown in Figure 1-11.

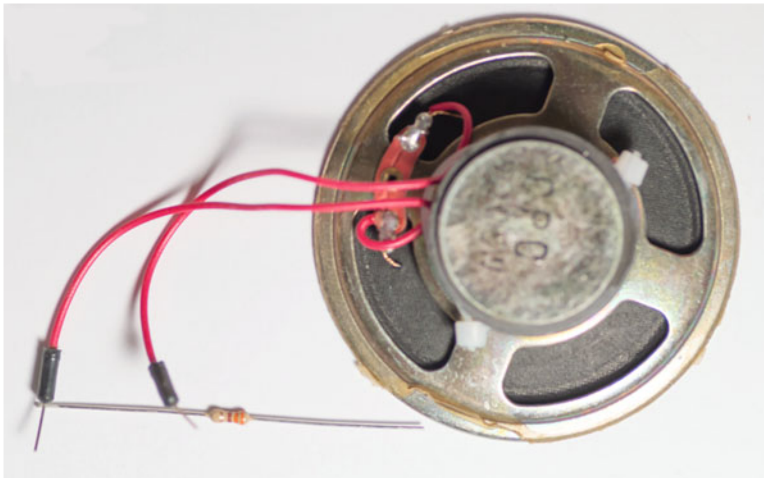

Figure 1-11. *Morse Code Practice Translator Components*

New Parts

- One 100Ω resistor. See A Word on Resistors. A 330Ω will also work, but it's quieter.

New or Used Parts

- One small speaker, 8Ω or higher.

If you solder, soldering some hookup wire to the tabs on the speaker will make it easier to connect to your breadboard, but twisting the hookup wire on tight will work. I sacrificed a jumper to connect mine. Literally any small speaker will do. If it's designed for a car sound system or is much larger than your hand, it might not make enough sound to hear with the energy we'll be giving it.

Larson (Memorial) Scanner

The Larson (Memorial) Scanner is a classic Arduino project named after the late TV producer Glen A. Larson. It's done here with a twist. Its components are shown in Figure 1-12.

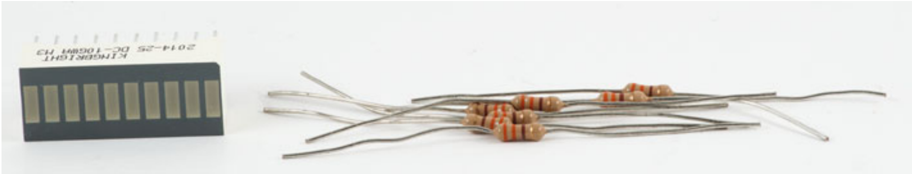

Figure 1-12. *Larson (Memorial) Scanner Parts*

New Parts

- Eight 330Ω resistors. See A Word on Resistors.

These are current limiting resistors to keep the LEDs from being fried, and the ATmega1284P from having its outputs overloaded. These should be good for any color of LED about this size If your LEDs are weirdos (very small or very large,) you may need to resize. Other colors than red are fine, and can use the same resistors, since their forward voltages are higher.

New or Used Parts

- Eight LEDs, preferably the same size, shape, and color or one LED bar graph (any orientation) with at least eight LEDs in it. Common cathode or common anode bargraph LEDs will work, but the wiring will be different.

If you're buying new, make sure to download and read the datasheet on your LEDs. They can't have a forward voltage drop of more than 5 volts (that's all the ATmega1284P can provide) and they need to be happy on less than 20mA each, so we don't overload the ATmega1284P's outputs or overall current limit. Don't worry, I'll explain in Chapter 4.

These can be bought new at Mouser, Digikey, SparkFun, Adafruit, and just about anywhere that sells electronic components.

Transistor Tester

The Transistor Tester is a simple project to get familiar with pulse-width modulation and analog inputs. Various transistors in different packages are shown in Figure 1-13.

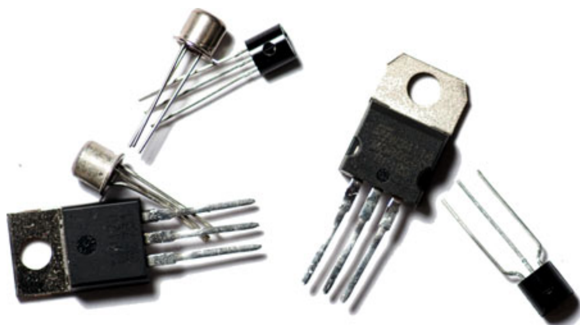

Figure 1-13. *Transistors*

New Parts

- Three 330Ω resistors.

These are the load resistors to convert the PWM pulses into a variable voltage.

Used Parts

- Small signal BJTs (bipolar junction transistors) and/or Darlington transistors.

If the leads will fit in the breadboard, it has a chance of working. I have an assortment from Microcenter, but if there's one part that should be in everyone's junk collection somewhere, it's these.

TTL Tester

7400 series TTL ICs have held the computer revolution together from their introduction in the 1960s right through today. The TTL tester explores these wonderful, simple ICs. (Figure 1-14)

Figure 1-14. *7400 Series TTLs*

New or Used Parts

- Any 74xx series logic IC of any vintage. 7400s, 74LS10, 74F? Bring 'em on. As long as they're 5-volt logic, at the speeds we're going, they should all work fine.

If you must buy new, Futurelec has by far the best assortment. The 74xx00 quad-nand gate is the most fun, and we use it in another project anyway. Mouser and Digikey have these as well. As always, get the DIP, PDIP and/or through-hole versions.

Logic Probe/Injector

New Parts

- 2 680Ω resistors.
- 1 0.1uF capacitor.

New or Used Parts

- 1 74xx00 IC, where xx is F, LS, HCT, or any other type of 5v 7400 capable of sinking at least 1.5mA, and whose inputs go high around 2v.

If you bought a 74xx00 for the TTL Tester project, it's perfect.

ROM/EPROM/EEPROM Explorer

The ROM/EPROM/EEPROM explorer lets you peek inside 8 bit ROM ICs, like the ones in Figure 1-15.

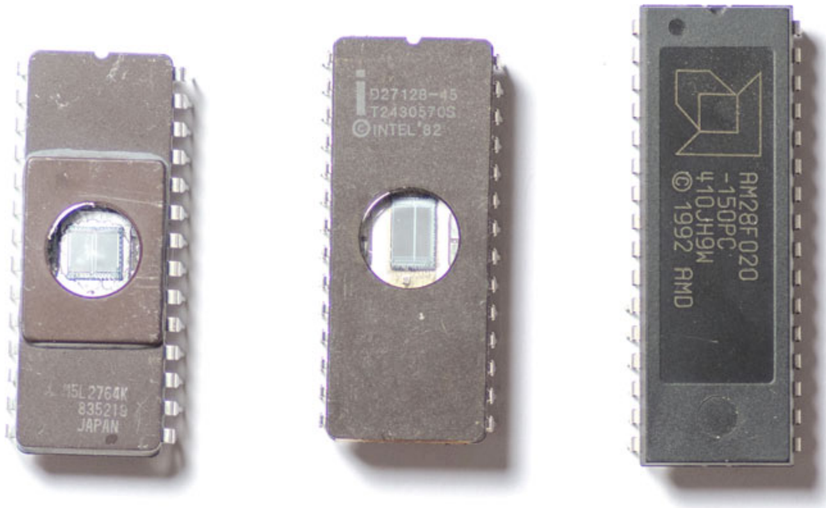

Figure 1-15. *EPROMS and Flash*

New Parts

- One 0.1µF capacitor. This is used to decouple the ROM's power supply from AC noise.

Used Parts

- Any 8 bit parallel (multiple data lines) ROM, EPROM or EPROM. It must be happy on a single supply voltage of 5v. For EPROMs, this is anything made after about 1980, particularly 27xx series EPROMs, preferably not a 2716. Parallel EEPROMs and Flash will work too.
- A datasheet for your ROM/EPROM/EEPROM/Flash. Their pinouts can be different, so you'll need to look yours up.

There's really no point buying EPROMs for this project new, as they will be blank, unless you have the optional universal programmer (See: Tools and Supplies). If you do have the programmer, or a friend does, any 2764 EPROM will work, and Futurelec has them, but unless the old school, quartz window EPROMS intrigue you as much as they do me, I'd suggest getting a 28C series EEPROM or 28/29F series Flash IC. They interact more or less the same way.

There's a second setup and sketch in this project for writing to a 39SF020A flash, which is a common type and still readily available from supply houses today. If you're buying new, look for this part number: SST39SF020A-70-4C-PHE. Many other types of flash IC will work, but may involve rewiring and/or extensive rework to the sketch to accomidate different commands, registers, or fussy timing.

ATA Explorer

The ATA explorer lets you look at the contents of an old parallel ATA/IDE drive. Components are in Figure 1-16.

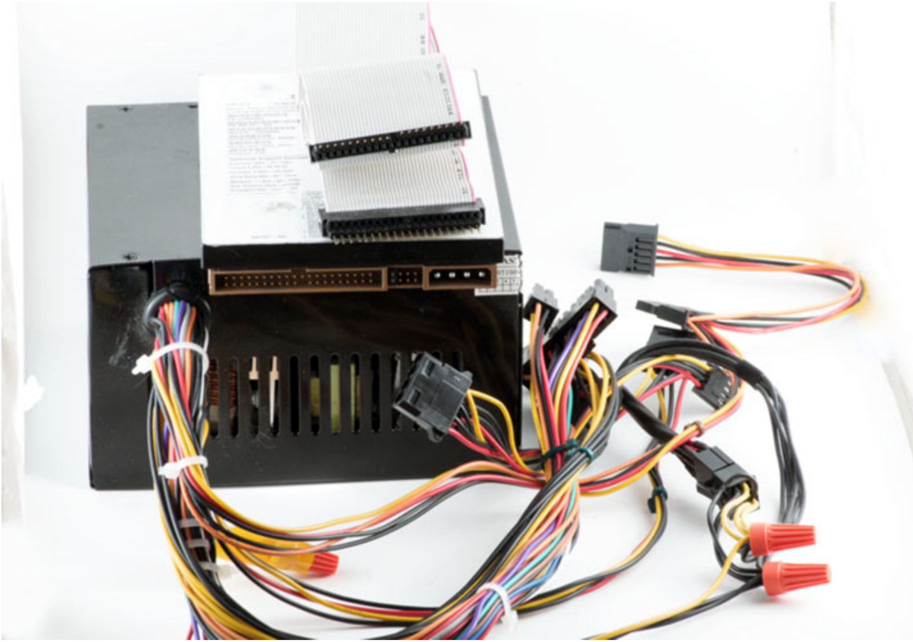

Figure 1-16. *ATA Explorer Components*

New parts

40 pins worth of extra long pin headers, ideally in two rows of 20. These are available at Adafruit or Schmartboard.com. Make sure they're the kind with equal lengths of pin on both sides of the plastic divider, just like we used on the Cestino itself for the TTL-232 connector.

Used parts

- One ATA/IDE drive. Needs to be operational.

The sketch only knows LBA24, so the drive needs to be less than 128GiB in order to work. Drives from the late 1990s to the early 2000s are the best choices.

Notebook drives should work, if you have a notebook to desktop adapter.

- 1 40 pin PATA/IDE cable with two drive connectors and no twists.
- One ATX power supply (Working)

Hard drives require 12-volt power for the spindle motor. Even notebook drives require more current in 5 volts than we can safely draw from the Cestino's existing power system. Fortunately, any scrapped computer in the last 15 years or so will have an ATX power supply. Ideally, it will have a power switch in the back, but this isn't necessary. The newer your ATX power supply is, the better, as the filter capacitors dry out over time, which makes for a noisy power supply.

Dice Device

This project builds a dice rolling device for a well known role playing game system popular in my youth. Parts shown in Figure 1-17.

Figure 1-17. *D20 System Dice Device Parts*

New Parts

- 2 330Ω resistors
- Optional: 8 150Ω resistors (these improve readability of the display)

New or Used Parts

- Two tactile button switches, momentary on. One for rolling the dice, one for setting the size.
- Two 7 segment LED displays, any color, any size that will fit comfortably on the breadboard. Common Anode or Common Cathode are fine. One double-digit 7 segment display will also work. If you're buying new, it will save you some rewiring of the project to get the LTD 4608JG, which is what I used.
- 1 ULN2803 or similar Darlington transistor array, or 2 TIP120 Darlington transistors.

Just about any NPN transistor capable of switching a continuous 80mA or more will do the job, with appropriate base and emitter resistors.

Z80 Explorer

The Z80 Explorer is the last and most complex project in the book. In it, we'll hook an old 8 bit microprocessor (a Z80; see Figure 1-18) up to the Cestino, build it some virtual memory, and get it to execute some instructions.

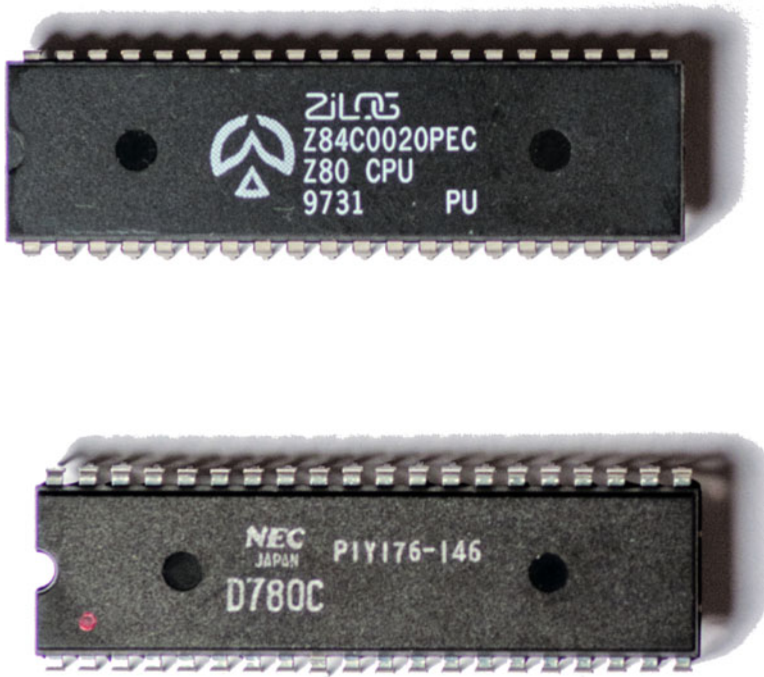

Figure 1-18. *Z80 Microprocessor and a Clone*

New or Used Parts

- Any "modern" Z84 prefix Z80 or fully compatible clone microprocessor. Its datasheet must list either "fully static design" or a maximum clock period of infinity. The D780C shown may work, according to the datasheet, but mine turned out to be dead.

Even though they were first released in 1976, Z80s are still cheap and plentiful, and available at Mouser and Digikey at speeds up to 20MHz. The 20MHz unit is twice the price of the 10MHz one, but if you are considering building a working computer around this CPU at some point in the future, get the 20MHz version. It makes old Z80 software fly. If you have a µPD780C or similar from NEC, it's worth trying, at least.

Postage Is Not Your Friend

You've probably noticed that my list of suppliers overlaps a great deal from part to part. This is not an accident. I tend to use the same suppliers over and over again, and yes, reliability matters to me. The other reason I overlapped them is that shipping is expensive. It's very easy to pay more to ship tiny electronics components than the components themselves cost.

It's a strange economy. Although your first instinct might be to shop around and get the lowest price for any given part, consolidating your orders into as few boxes as possible often saves more on shipping than the difference in part prices.

Summary

These are the parts you need for all the projects in the book. I'll list the parts again briefly in each project.

CHAPTER 2

Cestino

Arduinos have a lot of useful stuff baked into them, so much so that they seem a little magical. I'm here to tell you that there is no magic. More than that, I'm going to prove it to you.

I'm a writer with a background in computer science. One thing I've found, as I've pursued technical topics over the years, is that the best way for me to understand a given technology is to dive right in and use it. Breaking up the magic and exposing how the given thing really works goes a long way, at least for me, toward understanding it, and being able to create new projects with it. So it was with Arduino. My first one was a kit. If I seem, occasionally, to be an Adafruit fanboy, it's because that first Arduino kit came from them. It was the first piece of digital electronics I ever soldered that worked. I would not be writing this book without having built that kit, and the first time the LED flashed under program control on that kit *was* kind of like magic. More so because I did it myself.

In light of that experience, I thought building an Arduino-derived board on the breadboard would be a great first project. Flashing the LED on an Arduino you bought is fun. I do it on all of them as a basic health check. Flashing the LED on one you built? That's more fun.

There is another benefit. Most Arduino topics deal with digital and analog pins, digital_read and digital_write. Most of the electronics in your junk box probably communicate in eight-bit parallel, and to really talk to them efficiently, we need to have the Arduino speak eight-bit parallel as well. It can do that. Most ATmega microcontrollers have eight-bit ports, but there's a catch. While the Arduino core software can re-map the pins of nearly any AVR microcontroller to the standard pin layout, the ports themselves are hardware devices. They are where Atmel says they are. If you need more ports than the ATmega328p (used in the Arduino Uno r3 and many, many others) can provide, your ports may literally be anywhere, and clone boards may wire them differently still. If you ever wondered why most Arduino topics pretty much ignore eight-bit ports, this is why. With Cestino, we can bypass that problem entirely. You'll have exactly the same hardware on your breadboard that I do, and your ports will be wired the same way.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-1425-1_2) contains supplementary material, which is available to authorized users.

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_2

There's yet another benefit. We're going to use the Cestino's hardware a lot more deeply than normal Arduino programming. Since I had to develop a configuration for both the bootloader and the Arduino core software (I'll talk about both of these in chapter 3), I changed the pin layout to correspond exactly to the pinout of the ATmega1284P microcontroller. Pin 1 on the chip is digital pin 1. The analog pins are backwards, just as they are on the pinout diagram from Atmel, starting at 40 for analog 0 and going to analog 7 on pin 33.

There are some downsides. The usual Pin 13 blink sketch won't work without modification. The Cestino has no digital pin 13. Pin 13 is where the clock signal goes in. And so on. Since we're developing our own sketches, it won't get in our way too much, and it's not difficult to convert normal Arduino sketches to use the Cestino's pin layout.

The Stuff You Need

The Cestino is not much more than the ATmega microcontroller, an oscillator, some passive components and some LEDs. The parts list is fairly short.

New Parts

1 Atmel ATmega1284P sometimes called the ATmega1284P-PU microcontroller, in PDIP 40 (40 pin through-hole.) See the warning in Chapter 1 about ATmega1284P-PU vs ATmega1284-PU. You want the P-PU version.

4 0.1uF capacitors, any ceramic or tantalum that will fit on the breadboard. See A Word on Capacitors.

1 10kΩ resistor.

1 330Ω resistor.

Hookup Wire.

USB to 5v TTL level Serial Cable/board/whatever.

New or Used Parts

1 20 MHz Full Can TTL crystal oscillator.

1 Tactile button, momentary-on (only on while pressed)

1 existing Arduino.

2 Solderless Breadboards: 6.5 inches by 2.1 inches (165.1mm x 54.36mm) with 820 tiepoints (holes.)

The ATmega1284P Datasheet. It's available at `http://www.atmel.com/images/doc8059.pdf`

Anatomy of the Cestino

The Cestino, shown in schematic form in Figure 2-1, is an Arduino derivative, stripped down to the bare minimum components it needs to run.

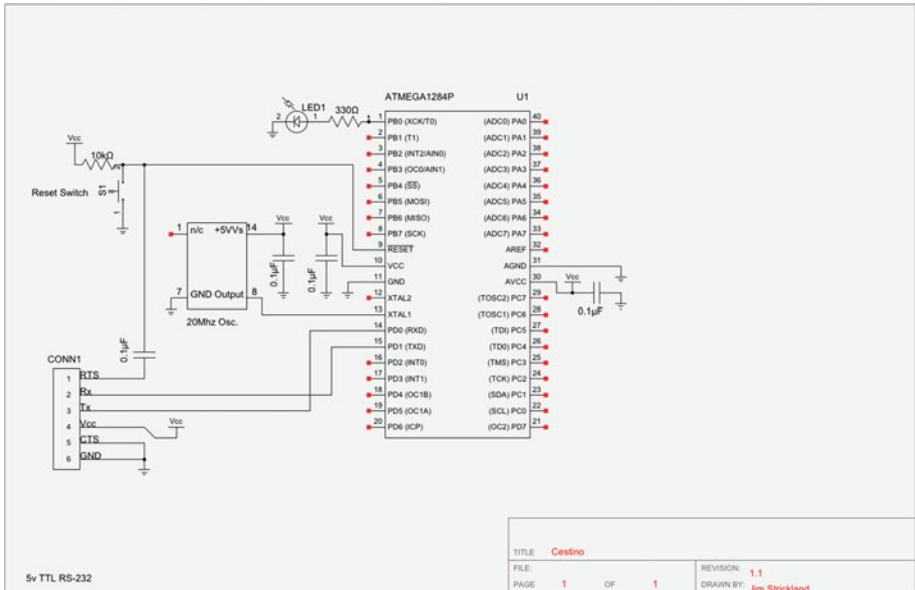

Figure 2-1. *Cestino Schematic*

Unlike a modern Arduino, say, the Uno r3, Cestino does not have its own voltage regulation circuit for external power. It does not have a USB-Serial solution built in. It does not share the standard Arduino pin arrangement, either. What it has is an Atmel ATmega 1284P microcontroller, a 20Mhz TTL crystal oscillator to generate the clock signal, a reset button and a resistor to generate the reset signal, and a pin header to connect an external USB to TTL level RS-232 cable or board, wired to the same standard used by most Arduinos, which also provides power to the Cestino.

There are only two complex components in the Cestino: the ATmega1284P itself (Figure 2-2) and the crystal oscillator, and I'll talk about them next.

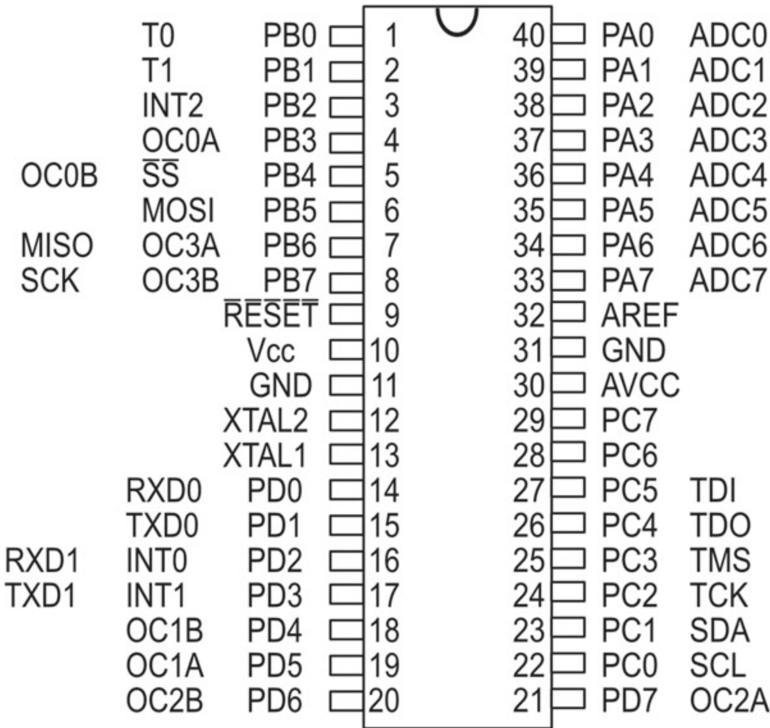

Figure 2-2. *ATmega1284P Pinout Diagram*

The ATmega1284P Microcontroller

Meet the Atmel ATmega1284P AVR microcontroller in the PU, or plastic, 40 pin dual inline pin package, hereafter referred to as the ATmega or ATmega1284P. According to the datasheet, it is an eight bit AVR RISC microcontroller, has 128K Bytes of flash, 4K Bytes of EEPROM, 16K Bytes of internal SRAM, that can run at speeds up to 20Mhz, producing about 20MIPS of performance.

That's great. What does it mean? Let's break it down.

Microcontroller

A microcontroller is like a CPU, but there are important differences. Most microcontrollers, the AVRs included, use a modified Harvard architecture. This means that the ATmega has two different memory systems, one for instructions, and one for

data. The two meet only under very controlled circumstances, and the ATmega can access them simultaneously. You can see this very clearly in the specs, where the ATmega has 128K Bytes of flash for program memory. This is not a flash drive. The ATmega does not load the program from flash into RAM and then execute it. It reads its instructions (the program) straight from the flash memory and executes them.

The 16K Bytes of internal SRAM (Static RAM), by contrast, are for data only. No instructions can be executed from SRAM. Data and Instructions remain separate. The AVR is a modified Harvard architecture, which allows data to be copied from flash into RAM, but there is no way to write the data back to flash, and data in SRAM will go away when the power is turned off. What if you have configuration data you need to store?

That, in fact, is what the 4K Bytes of EEPROM (Electronically Erasable Programmable Read Only Memory) are for: a place to stash data, like software configuration, that persists even if you turn the power on and off. You can't store instructions there either. They won't be executed. The EEPROM, like the SRAM, is for data only.

The thing to remember most about the Harvard architecture is that the program can't modify itself or the flash it is stored in. (Well, it can, but it's complicated to get at.) Want to create a flash data drive out of empty space in the internal flash? You can't. Want to have one program load another into flash? You can't. A lot of the capabilities (and vulnerabilities) we take for granted with our desktop computers, our phones, tablets, and so on, are not possible on Harvard Architecture designs. Hardware configuration is likewise out of reach. It is set with a series of "fuses" which we'll cover in chapter 3.

■ **Note** If you're curious, the other architecture, the one that your desktop computers, phones, tablets, and so on, use is called the Von-Neumann or Princeton architecture. This is the familiar "All the memory is one big bucket, and keeping memory separate from program instructions is a software job" setup you're familiar with.

Microcontrollers also typically have a lot of peripherals built in. RS-232, I²C, and SPI serial communication, timer/counters, and so on. What is a peripheral and what is on-chip is a fuzzy line in modern times, since a system-on-a-chip (SOIC) such as the one that drives the Raspberry Pi, is a Von-Neumann architecture CPU with almost all the functionality of the system rolled up into one piece of silicon. In days gone by it was an important distinction. When we get to Chapter 11, we'll be using the ATmega's peripherals to communicate with a vintage microprocessor, born with no peripherals at all.

RISC

RISC is one of those terms that used to mean something very specific, but has become fuzzy over the years. It's an acronym that stands for Reduced Instruction Set Computing. It was introduced in the 1980s. To understand RISC, you need a little background.

The Clock

We tend to think of the CPU or microcontrollers we deal with in automotive terms, as the engine of the computer. It's really not quite like that. A CPU is a series of tasks that are useful for computing, implemented in electronics. It's more like an electronic program itself than an engine. Each task, or instruction, takes a certain amount of time to accomplish. That time is determined by the physics of the transistors, resistances, and capacitances of the circuits that carry it out. Each one uses resources inside or outside the CPU, and most need exclusive access to those resources. If one instruction is copying a byte from memory into a register, there's no way to know what the result would be if another instruction's mechanism simultaneously erased that byte. That would be bad. On the hardware level, everything in a computer must be predictable at all times.

The clock breaks up the time available into discrete chunks, so that instructions don't mess each other up. If copying a byte from memory into a register in the CPU takes less than 50 nanoseconds(ns) then, on our 20 MHz ATmega, it can be done in one clock cycle, and one clock cycle will be allocated to doing that job. When that clock cycle is done, another instruction can be carried out. More complex instructions often take more than one clock cycle to be carried out, and so more clock cycles are allocated.

Reduced Instruction Set

In the old days, RAM was astonishingly expensive. Every byte had value, so you wanted your programs to use as little as possible. Processor and microprocessor designers went to great lengths to give you a list of instructions that did complex operations in a single assembly instruction. Some of these could take lots of clock cycles to execute, and they took a lot of the available transistors on a given IC to implement, so niceties like cache and so forth weren't an option.

By the 1980s, it was clear that the price of RAM was plummeting, the amount of RAM a given microprocessor might have available was skyrocketing, and programmers were using compilers (like AVR GCC, which we'll use to program the Cestino) to generate code instead of using assembly language. Since compilers seldom used the entire breadth of a complex instruction set, the RISC pioneers decided that they could reduce the instruction set to the bare minimum required. This allowed them to optimize each instruction's underlying circuitry. RISC was fast, for most applications. Most RISC instructions could be executed in one clock cycle. Contrast this, when you get to Chapter 11, with the fact that the Z80, a CISC (Complex Instruction Computing) microprocessor we'll be using, can do nothing in less than three clock cycles. Most operations take many more.

The trade off, at least traditionally, is that some operations take a lot more instructions to get done. More instructions, more clock cycles, more memory.

Today, the difference is fuzzy indeed, as the instructions a given CPU provides may be in microcode, which is more or less software executed by even faster electronics. Transistor counts are in the billions and climbing, and caching and pipelining allow even a "complex" instruction set (CISC) CPU to execute an instruction every clock cycle under ideal conditions.

When applied to the Atmel AVR line, of which our ATmega1284 is a member, RISC mostly means a small instruction set, capable of executing each instruction in one clock cycle. Because AVR is also a Harvard architecture, it can read an instruction and execute the previous one at the same time. When they say the ATmega1284P can produce

20 MIPS (million instructions per second) with a 20 MHz clock (20 million hertz, or cycles per second) this is what they're talking about, and how they got there.

Eight Bits

The bit-width of a computer is another of those terms that gets bandied about without much understanding. Most people talk about it in terms of how much RAM a system can have. That's not really the important part of the story.

Eight-bit computers like the ATmega1284P can act upon 8 bits of memory simultaneously. That's it. That's all that's required. Typically, they also have 8 bit instruction sets, that is, most instructions will have a code from 0 to 255 that designates them in memory. Eight bit computers also process large numbers (such as 32 or 64 bit floating point numbers) more slowly because they must deal with them in 8 bit chunks, requiring more instructions per computation.

Bit width can be related to memory size, but it isn't always. Even the ancient Z80 we'll use in Chapter 11 has two bytes of memory address space (64K Bytes), and 16 bit registers inside for memory addresses internally. The fact that certain 16 and 32 bit processors limited their memory registers to the bit-width of the CPU was a design decision. It probably conserved transistors or improved performance (or both). It didn't have to be that way. One need look no further than the ATmega1284, which addresses 128K of flash, to see that it needn't be so.

Package

The *ATmega1284P-PU*, the one you have, looks like a flat, 40 legged bug with shiny metal legs. That is not actually the chip. That's the package. It's called a DIP (Dual Inline Package), a DIL (Dual InLine), or a DIPP (Dual Inline Pin Package), a PDIP (Plastic Dual Inline Package – a CDIP would be ceramic) or more precisely as a DIP40 (A Dual Inline Package with 40 pins). According to Wikipedia, DIPs were invented in 1964. The actual chip, or die as it's more properly known, is probably about the size of the word "DIP" on this page, depending on font size.

The largest number of pins commonly used in the DIP format was 64, with 40 pin DIPs being much more common. Most new integrated circuits need more leads than that, including larger and more powerful ATmega microprocessors. Why did I choose the ATmega1284P? It was the largest pin-count ATmega in a DIP package that is breadboard friendly.

20 MHz TTL Full Can Oscillator

The 20Mhz TTL full can oscillator, also known as a clock oscillator, a crystal oscillator, or simply an oscillator, depending on who writes the datasheets, is a combination of a quartz crystal resonator and the drive electronics to produce a square wave, 50 percent duty cycle clock signal at 20Mhz, +- 100PPM.

What does that mean? Simply this: When you connect the oscillator to power and ground, every 0.05 microseconds (μsec), it will generate one full off pulse and one full on pulse, that the on and off pulses will be of equal length (50% duty cycle), and that as a clock, it's accurate to 0.01 percent. In human readable numbers, it will gain or lose a little

more than a second every hour. That's close enough. The on or high pulses will be at least 4.5 volts, and the low, or off pulses will be at maximum of 0.5 volts. Running on 5 volts, as we'll be configuring it, the ATmega's clock input pin will take low voltages up to 0.5 volt, and high voltages as low as 3.5 volts, so those values are fine.

Placing Components

So let's get started. If you haven't already snapped your two breadboards together into one wide breadboard, now's a good time. Things get a little fragile once the little components and wires are in place. My breadboards have double-face tape on the back, so I stuck them to a piece of cardboard for stability.

Take your ATmega1284P and look closely at it. You'll note a notch in one end. This is one of several standard notations for where pin 1 is. With that notch up, pin 1 is on the top left, two is below it, and so on to pin 20. Pin 21 is on the bottom right, continuing to pin 40 at the top right. In that orientation find tiepoint row 11 on your left breadboard (refer to Figure 2-3). and gently press the ATmega into the tiepoints with its pin 1 in row 11. The ATmega is wide. It will reach two columns beyond the channel on the other. I recommend placing it so that you have three columns of open tiepoints on the left and two on the right, to make room for the Cestino's internal wiring (which is almost all on the left) and still have some tiepoints open on that side for jumpers later.

■ **Note** you don't have to follow my layout for your Cestino. That's the beauty of breadboards. You can put components in where you want. The really important thing is that the connections are the same and that the high frequency line from the oscillator to the ATmega is as short as possible. Weird signal distortions from too-long high frequency wires are not something we want on the Cestino's clock.

Skip a row of tiepoints so you can get a screwdriver under the ATmega later if you need to remove it.

Now take the 20 MHz TTL full can oscillator, hereafter referred to as the oscillator, and look at it closely. Three of its corners are rounded. One is not. The sharp corner is where pin 1 is. There may also be a dot painted on the shell on that corner as well. Put pin 1 of the oscillator into row 33 or so of your breadboard, and again, I'd put it so that it lines up with the ATmega on the left, but this is mostly for aesthetics.

Skip a couple rows of tiepoints to leave room for wires, and the all-important screwdriver access. Then install the momentary tactile button below the oscillator so that two contacts that will connect when the button is pushed are to the left of the valley in the center of the breadboard, as shown in Figure 2-3.

My tactile buttons are weirdos. Instead of having connections from pins 1 to 4 and 2 to 3 as you'd expect, they connect pin 1 to 2 and 3 to 4 when closed. You need to read the datasheet on yours to find out which pins need to go where, or you can figure it out with your multimeter in the Ohms setting. Your reset circuit may wind up looking different from mine.

If you have the extra-long pin headers, as from Adafruit and others, you want six pins worth, and stick them in column E, the one closest to the channel on the left, starting at tiepoint row 1 on the breadboard. This leaves just enough room for the reset circuitry the LED and its resistor. We'll get to those right after we get done wiring the power circuits. You might want to bend these pins over so the TTL RS-232 cable will rest in the empty space on your breadboard. If so, don't do it in the breadboard. The little metal connectors in the breadboard weren't designed to take much stress. I suggest two pairs of pliers, and be sure to hold all the pins on both sides of the plastic connector, or you'll snap the pins out of the plastic connector.

You could also leave them straight. I bent mine, as shown in Figure 2-3.

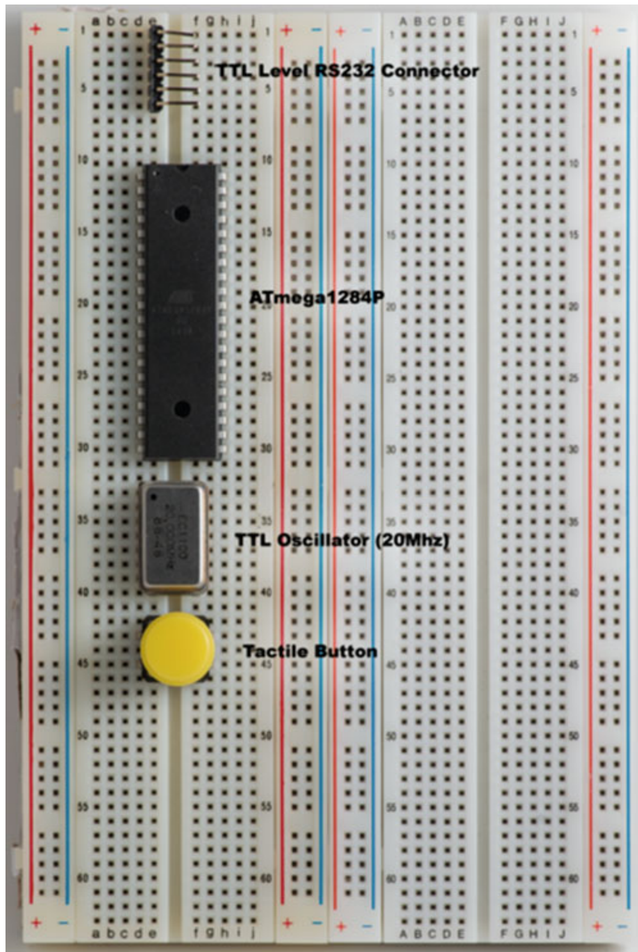

Figure 2-3. *Breadboard Layout*

Power Circuits

There are no unimportant circuits in the Cestino, except maybe the LED, but if the power circuits don't work right, nothing else will either.

If you look along the edges of your breadboards, you'll (probably) note that there are two columns of tiepoints, one with a red line next to it, and a + marking and one with a black or blue line and a - marking. There is probably a duplicate pair on the opposite edge. If yours don't have the lines, that's ok. If you have older breadboards with only one column of connectors, wiring the power circuits will be a little more complicated for you, but your breadboards will still work.

These columns of tiepoints are called busses. All the tiepoints in each column are connected together. The busses on each edge of each breadboard are not connected to each other, and obviously the busses of the left breadboard aren't connected to the busses of the right breadboard. That's the first job we're going to do. It's not a big deal (refer to Figure 2-4). At the bottom of your first breadboard, run a piece of hookup wire from the + bus on one side to the + bus on the other side of the same board. Convention is to use red wire for the + bus. Run another from the + bus on the right of your first breadboard to the + bus on the left of your second. Run a third from the + bus on the left of your second breadboard to the + bus on the right. All the + busses should be tied together.

Do the same thing with the - busses. Go from the left edge of the first breadboard to the right edge, to the left edge of the second, to the right edge. All the - busses should be tied together, too. Convention is to use black wire for the – bus.

These connections will make wiring the Cestino easier, and as we do projects, they will make powering the project ICs and components much easier as well. I try to leave four or five millimeters of bare wire to reach down fully into the tiepoint. If the wire's not quite long enough, it can result in a flakey connection, and those can be a real pain to track down.

■ **Note** if you're using other types of breadboards, the busses may also be divided on the vertical center, with one set on the top of the breadboard and one on the bottom. If you don't have the manual, grab a couple jumpers, set your multimeter for ohms, and see if you get continuity from the top of the bus to the bottom. If you don't, add a jumper or a piece of wire to bridge the spot where they divide.

Although we're on the subject of power (not to mention messing with the red and black hookup wire), let's go ahead and connect up power to all the components we've put in place.

Start with the ATmega. Looking at the pinout diagram on page 2 of the ATmega1284P datasheet, we can see that pin 10 is Vcc and pin 11 is GND, or ground. Wire the GND pin to the nearest - bus, and Vcc to the nearest + bus. These pins provide 5 volt power to the ATmega. If you use the tiepoints closest to the busses, it will leave room for the capacitor you will install later.

■ **Note** Vcc, Vdd, and so forth are, for our purposes, all the same. They indicate the peak voltage applied to the pin, and that it is the positive voltage. The subscript—cc, dd, and so on—actually call out various parts of different transistor technologies. The first c in this case is for the collector (we'll cover transistors at length in Chapter 5), and the second c means it's a supply voltage. This comes from a standard, IEEE 255, but is often misrepresented in digital schematic software.

On the other side of the ATmega, on pin 30 is the AVcc pin, and on 31 is another GND pin. (Remember, count from pin 21 up the right-hand side.) These supply power to one of the ports, and to the analog to digital converter. Wire pin 30 to the nearest + bus and pin 31 to the nearest - bus. We'll talk about ports in Chapter 4, and the analog to digital converter in Chapter 5.

■ **Note** that pin 30 is AVcc and 31 is GND. They are internally connected to Vcc and GND, and they are backward from the Vcc and GND terminals on the other side. I mixed them up once, and got an ATmega hot enough to burn my finger on contact. I haven't tested that ATmega to see if it's damaged or not. I won't be surprised if it is.

Next, we'll wire up the oscillator. The diagram in the data sheet for the ATmega doesn't really show you the pins for the oscillator, but you'll find that this is a standard part. The datasheet for the oscillator itself should have the pin assigments listed. It's only got four pins, and it's oriented just like it is in Figure 2-1.

I've represented the oscillator in the schematic as though it has 14 pins, as it fits in a 14 pin socket, but of these, only four pins actually exist, and three are used. Starting from the pin 1 marker on the top left, where the sharp corner and possibly a black dot are, pin 1 is not connected to anything. Pins 2, 3, 4, 5, and 6 don't exist. Pin 7 is ground. Pin 8 on the other side at the bottom is the output pin that we'll use later, and pin 14 on the top right is the power pin, called Vcc. Wire pin 7 to a - bus, and pin 14 to a + bus.

Next is the TTL Level RS-232 connector. It's the pin header at the top of your breadboard, and it's labelled CONN1 on the schematic in Figure 2-1.

■ **Note** the pinout shown in Figure 2-1 and described below are how the FTDI USB to 5 volt serial cable is wired. This is a de facto Arduino standard but check your datasheet. If your USB to serial cable is wired differently, you will need to wire this connector differently.

Connect Vcc out to the nearest + bus and GND to the nearest - bus. These connections will power the Cestino from the USB to TTL Level Serial cable/board/whatever.

Finally wire one side of your tactile button to a - bus. It doesn't make any difference which side, but it makes things easier to wire if the ground connection is furthest from the ATmega.

Done? Almost. There's one more thing.

If you look at the schematic, you'll see a number of capacitors (c1 through c3) that go from a Vcc pin to ground. That little symbol with three lines of decreasing length? That's ground. These are 0.1 μF (microfarad) decoupling capacitors.

As I mentioned in Chapter 1, capacitors only conduct alternating, or varying voltages. When the voltage on both sides hits equilibrium, they stop conducting. We're using that feature here.

The world is a noisy place, electrically speaking. There are as many ways for electronic noise to get into your breadboard as there are tiepoints on the breadboard. Our ICs—the ATmega and the oscillator—expect DC. Noise will produce unpredictable results, which are the very last thing you want in an electronics project.

Noise, by definition, is fluctuating voltage. A capacitor will conduct it. These decoupling capacitors, then, take any variable voltage signal on the positive terminal of the IC and short-circuit it to ground. DC, on the other hand, does not fluctuate, so the capacitors won't conduct it at all. That's exactly what we want. The DC arrives at the ATmega and the oscillator with its AC or variable voltage noise damped down.

Go ahead and put those capacitors in now, as they're part of the power system. Put one from pin 10 to pin 11 of the ATmega, one from pin 30 to pin 31 of the ATmega, and one from pin 14 to pin 13 of the oscillator.

But wait, you say. There is no pin 13 on the oscillator.

You're right. We're using that row of tiepoints as a convenient place to connect the capacitor. Wire a piece of hookup wire from pin13 to the nearest - bus. Now the capacitor forms a circuit from power to ground, but only for varying voltage, or AC noise.

Figure 2-4 is photo of my board. You can see the power wires tying the busses together across the bottom, and the various power and ground connections from the oscillator, the ATmega, and the TTL RS-232 connector.

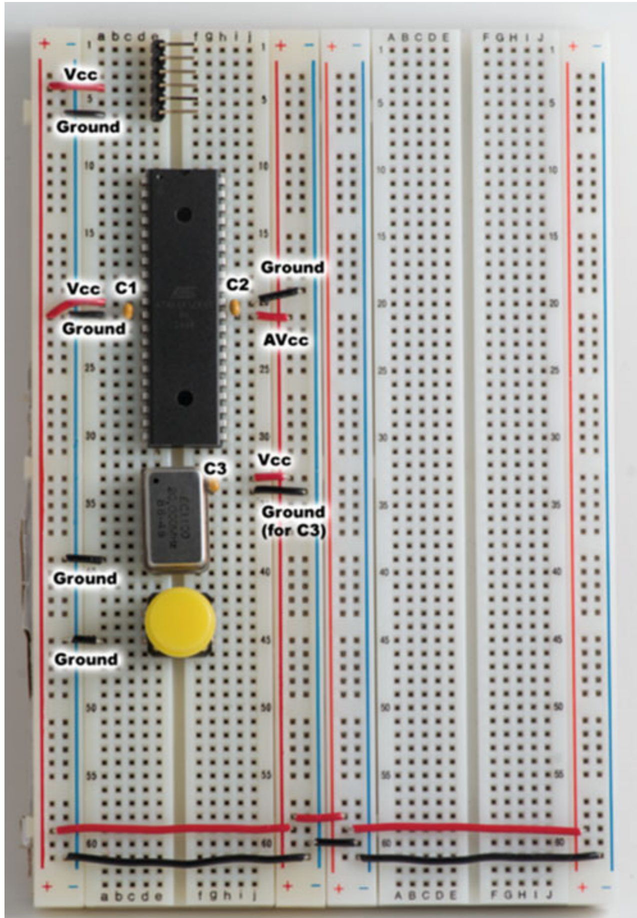

Figure 2-4. *The Power Circuits*

A Dab of Ohm's Law

I've thrown around the word voltage a lot, made mention of current in a number of places, and said that 1/4 watt resistors are fine for all of our projects. Before we go any further, I need to be clear on what those words actually mean, and how they're interrelated. It's time for a dab of Ohm's law.

Ohm's law states that the current (amps) between two points on a conductor is proportional to the voltage difference between those two points. That proportion is the resistance of the conductor.

How It Works

Rather than go to the usual plumbing analogy, I'm going to try to explain the physics. Metals like copper are made of a lattice of atoms that have a particularly useful property: each one has an electron that can be easily pulled out of its normal orbital and move along the lattice. The lattice is also quite dense, so there are a lot of electrons available, and they're close together. When you raise the voltage of one end of a conductor relative to the other (voltages are *always* relative) you nudge the electrons toward one end of the lattice. This is called electron drift.

The electrons don't move much, and they don't move fast, but because they are so close together, they don't have to. When the electron is pulled out of the outermost orbital of a copper atom, for example, the same force (voltage) has nudged another electron close enough to be captured by that orbital. An individual electron drifts slowly, but the cascade of electrons from one atom to the next happens at nearly the speed of light.

The size of the conductor matters. The greater number of atoms in the cross section, the larger the electron "cloud", and the more electrons can move at a given time. This is current. A thicker conductor can pass more current than a thinner one. Here's why.

The path of an electron moving from atom to atom in the lattice isn't smooth. If they run into an atom, they get bounced in a random direction, losing their energy in the process, usually as heat. The ratio between the current flow through a given conductor and the voltage differential between points on the conductor, is an amalgamation of the features of the conductor that prevent the passage of charges, and this is called resistance.

So Ohm's law, despite being published in 1827, before most of the physics concepts that underlie it were discovered, describes the motion of the electron cascade through a conductor, and the physical properties of the conductor that get in the way of those charges passing through in one neat little formula: I=V/R. That is, Current (in Amps) equals Voltage between the two points on the conductor, divided by Resistance in ohms.

Examples

If I connect a 330Ω resistor between a 5 volt power supply and ground what happens? Well, that resistor has a voltage difference between its terminals of 5 volts. Divide that by 330 ohms, and we're pulling a current of about 0.015 amps, or 15 milliamperes (mA). If, like the Cestino, you're powering the resistor with USB, you should know that your USB port is capable of delivering at least 100mA. By itself, this load won't even make USB break a sweat.

Consider this: the LEDs I used for my Cestinos want 2.4 volts and no more than 25mA of current through them. (I like big, bright LEDs.) The voltage coming out of the ATmega's pins is five or zero, depending on whether they're turned on or not. How can I protect the LED from getting voltages that are too high and currents that are too high? If I connect the LED directly, it will burn out in fairly short order.

I can use what's called a dropping resistor. Remember how I said that resistance is a measure of how small the electron cloud to transfer the charges and how much the atoms of the conductor's lattice get in the way, expressed as the ratio between the voltage and the current that flows through? Well, a resistor is a conductor that isn't as good as copper,

and we can exploit that to limit how much current and how much charge is propagated to our LED. Resistors, as the name suggest, add resistance to the circuit. But how big a resistor (or to put it another way, how much resistance) do I need?

We can use Ohm's law to figure it out.

First, do a little subtraction. I have five volts, I'm dropping 2.4 volts in the LED, that leaves 2.6 volts that I need to drop.

Next, do a little algebra on Ohm's law. In algebra, if I have an equation X = Y, that means whatever X is equals whatever Y is. If I divide both sides of the equation by Z, X/Z still equals Y/Z. As long as I change both sides the same way, their relationship doesn't change. The same is true for multiplication, addition, and subtraction. So if I=V/R and I multiply both sides of the equation by R, multiplying V/R by R negates the division by R, leaving V, but I have to put the multiplication by R on the other side, leaving me IR=V. Dividing both sides by I, negates the multiplication of R by I in RI, leaving R, but I have to divide V by I as well. I wind up with R=V/I.

Let's do that again in the unit names we're more familiar with. If Amps=Volts/Ohms, and I multiply both sides by Ohms, I get Amps * Ohms=Volts. If I then divide both sides of the equation by Amps, I get Ohms=Volts/Amps.

I know volts, that's 2.6. I know amps, that's 0.025, or 25mA. Plugging those figures in, I get 2.6/0.025 or 104 ohms.

There's another wrinkle. Each pin on the ATmega 1284P can source (give out) no more than 40mA, and I might want to use that pin for something else at the same time. Plus, that's a mighty bright LED. I can spare some brightness to cut my power consumption.

The resistor I chose for the prototype was a 220 ohm. Why 220? I was going for a 330Ω, the de facto standard for LEDs in a 5 volt world, and misread the resistor. The prototype works, and I know that with 220Ω I'm not overdriving the LED, but how far am I loading down that pin? Let's find out.

If Amps=Volts/Ohms, volts are 2.6 and Ohms are 220, I'm giving the LED 0.0118 amps, or 11.8mA: a little more than a quarter of the current available from that pin, and well within the limitations of the LED.

LED Circuit

Given these examples, it's probably no surprise that the next step in building the Cestino is to build the LED circuit. Go ahead and size the dropping resistor for your particular LED. If your LED is out of the junk box, and you don't have a datasheet on it, a 330Ω resistor is probably close enough. Bend the leads sharply down from the body of the resistor and connect it from pin 1 of the ATmega to tiepoint row 8 of your breadboard, assuming your layout is like mine.

LEDs, or light emitting diodes are, as the name suggests, diodes. This means that, like all diodes, charges can only flow one way through them. If your LED is new, it will have different length leads coming out of it. The short one is the negative, or cathode lead, or and the long one is the positive, or anode. If your LED came out of the junk box, you'll need your multimeter to sort out the polarity, or you can just hook it up and test it with a jumper once the board is powered. If it doesn't light, turn it around.

To find the polarity of the LED with your multimeter, set your meter on Ohms on the lowest setting (or Diode Testing, if it has that) and connect the LED to the leads. If you get no reading and nothing happens, reverse the polarity. On my cheap meter, I still get no reading, but the LED lights up. On my good meter, the diode test voltage drops from 3.1 to 1.75 volts, reflecting the forward drop of the LED. There's no reading in ohms, but once again, the LED lights up when the polarity is right for both the diode test setting and the Ohms setting. The red lead on most meters is positive, making that lead of your LED the anode, and the black is negative, making it the cathode.

■ **Note** Wikipedia has a good mnemonic for keeping anodes and cathodes straight: ACID. Anode Current Into Device. In vacuum tube (aka valve) it's easy to remember that the cathode is negative, because the plate or anode is connected to a high voltage positive circuit called B+. Electrons drift from cathode to plate, but the positive charges (lack of negative charges, really) go from plate to cathode.

I like to keep my component leads as short as possible. It keeps them from touching other components and causing shorts, and it looks better in photographs (Figure 2-5). As with hookup wire, four or five millimeters of wire to go down into the tiepoint gives a good, secure connection.

LEDs aren't quite as cheap as resistors and capacitors, though. If you want to use your LED for something else when you're done with your Cestino, you can leave the leads long. In either case, connect the LED from row 8, or wherever the free end of your dropping resistor wound up, to the - (ground) bus. If you decide to go with short leads, bend them out horizontally from the base of the LED.

■ **Note** The leads of an LED are very, very fragile where they come out of the LED package. If you bend them more than once, they're likely to snap off flush with the package, which makes the LED useless.

If you chose an LED with more than two leads, you have a multicolor LED, and you'll need to refer to your datasheet or do some probing with your multimeter to find out which leads do what. The LED circuit only needs one of the colors, so you can use the other, with another dropping resistor, for a power-on indicator or something else. If your multicolor LED has a common anode and two or more cathodes, you can use it either by using only one color or by using multiple colors wired to pin 1 (check your dropping resistor values, and be sure you don't draw more than 40mA from the ATmega pin).

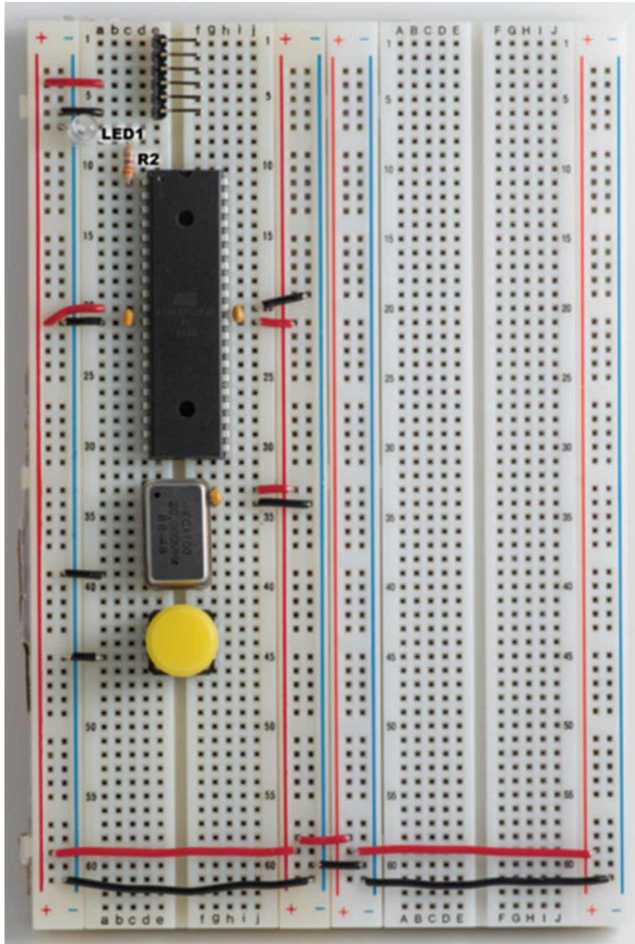

Figure 2-5. *The LED Circuit*

Clock Circuit

The clock circuit is the simplest of all. It's a piece of hookup wire, shown in Figure 2-6. Some care should be taken, though. Wire is not a perfect conductor, and we want the signal from the oscillator to arrive at the ATmega as pristine as possible. Even if your other hookup wires are luxuriant and gracefully braided, you really want to keep this wire short and neat. Run it from pin 8 of the oscillator to pin 13, XTAL 1, of the ATmega. We'll leave XTAL 2 unconnected.

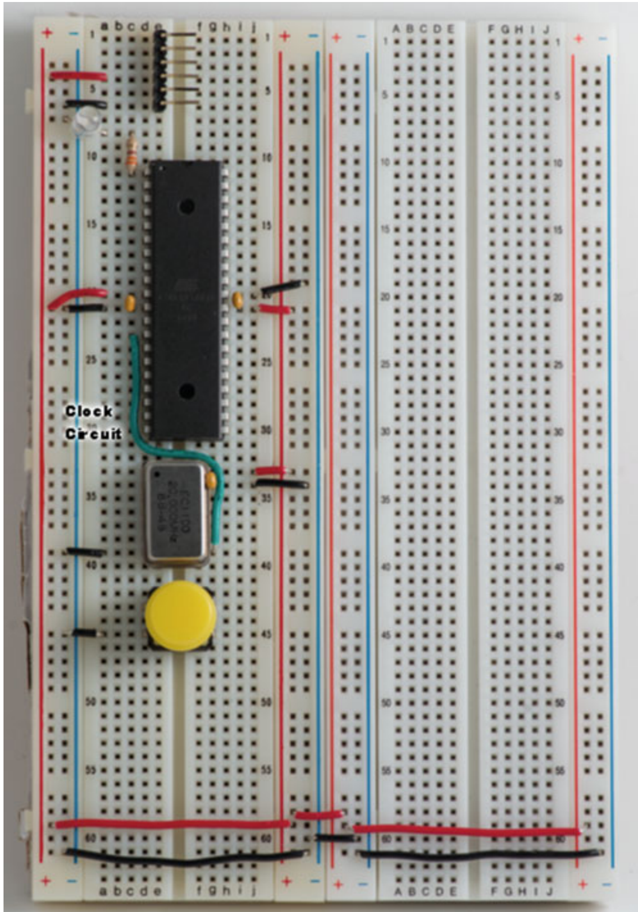

Figure 2-6. *The Clock Circuit*

Reset Circuit

On the ATmega pinout diagram (Figure 2-2), we can see the reset pin is pin 9, and that there's a line over the top of it. That line tells us something very important about that pin: it is active low. This means that as long as that pin has +5 volts on it, a reset *doesn't* happen. On page 327 of the datasheet, Atmel tells us that the voltage threshold for a reset is somewhere between 0.2 * Vcc and 0.9 * Vcc. Because our Vcc is 5 volts, that means that as the reset pin's voltage drops, somewhere between 4.5 volts and 1 volt, the ATmega will reset.

That pin has no voltage supplied to it by the ATmega itself. We have to make sure that pin stays at 5 volts or, more precisely, at more than 0.9*Vcc. So how can we hold the pin high until we want it to go low? It's easy to picture a relay or a transistor doing the job, or a light switch, even, but there's a much easier way.

The pin draws 10μA (microamperes), so it doesn't take much current to hold it at 5 volts. If we wanted to limit it to that current, we already know how to calculate a resistance to do the job with Ohm's law. If R=V/I (Ohms=Volts/Amps), and V is 0.5 (5 volts minus 4.5 volts, which is 0.9 * Vcc) and I is 0.000010 Amps (10μA), that puts our resistance at 50kΩ (50,000 ohms). Any value greater than that, and the current draw of the reset pin will pull the voltage down to where the ATmega might reset. We *don't* want the ATmega to reset until we tell it to, so any resistance between the + bus and the ATmega reset pin should be smaller than 50kΩ. If you're scratching your head, wondering why you'd want a resistor in the reset circuit at all, read on.

When the reset button is pushed, we want to pull the voltage on the reset line below the *minimum* reset threshold voltage of 1 volt, to be sure that, no matter what, the ATmega resets. There's an easy way to do that. Short the reset pin to ground. If you look at the schematic in Figure 2-1, that's exactly what we're doing. The resistor, while it needs to be a lower value than 50kΩ to keep the voltage high when the button is up, also needs to provide enough resistance to limit the current flowing between Vcc and Ground so that the voltage on the + bus doesn't drop too much. The ATmega won't function properly at 20 MHz if Vcc drops below 4.5 volts. The oscillator is even touchier.

By now, you've probably recalled from the parts list that I used a 10kΩ (ten-thousand ohm) resistor for the reset pullup resistor. With a 10μA load from five volts, this gives us a drop of 0.1 volts, which leaves the voltage at 4.9 volts on the reset pin. That's plenty to keep the ATmega from resetting. When we short the reset pin to ground, we change what the resistor is doing. It's now dropping 5 volts from one side, on the + bus, to the other, on the - bus. Using Ohm's law again, we discover that 5 volts/10,000 ohms is 0.5mA – half a milliampere, or about 1/1000th of what USB is delivering to the board. We can afford that. It's not going to draw Vcc so low that the ATmega stops working at all.

There's another consideration. Our resistor is turning that current into heat. There are strict limits to how much heat a resistor can deal with before it goes up in smoke. You may recall from Chapter 1 that resistors are rated in watts, and that I recommended using 1/4 watt resistors for all the projects in this book. Am I right? Is that safe? How many watts are we dissipating? Let's find out.

A watt of heat is equal to volts multiplied by amps. To plug in the symbols from Ohm's law, W=VI. We know amps: 0.0005 (0.5mA) and volts: 5. 5 times 0.0005=0.0025 watts. Our resistor can dissipate 1/4 watt, or 0.25 watts, so we're nowhere close to its limit.

Okay. So we want a 10kΩ resistor connected to the + bus to pull the voltage on the reset pin high enough to prevent a reset, but keep the current low enough when we short the reset pin to ground that we don't smoke the resistor or pull Vcc down so far that the ATmega stops working properly. We know the 10kΩ resistor can do all that. It's called a pull-up resistor, because it pulls the reset pin up to 5 volts.

We know from the pinout diagram that the reset pin is pin 9 on the ATmega. It's the one right before the Vcc connection. We already wired the tactile switch to ground when we did the rest of the power circuits, so we have all of the pieces we need. Let's wire the reset switch.

Plug the 10kΩ pullup resistor into the + bus, and the other side into the tiepoint row containing pin 9 of the ATmega. Then run a piece of hookup wire from pin 9 of the ATmega to the side of your tactile switch that isn't wired to the - bus. The full circuit is shown in Figure 2-7.

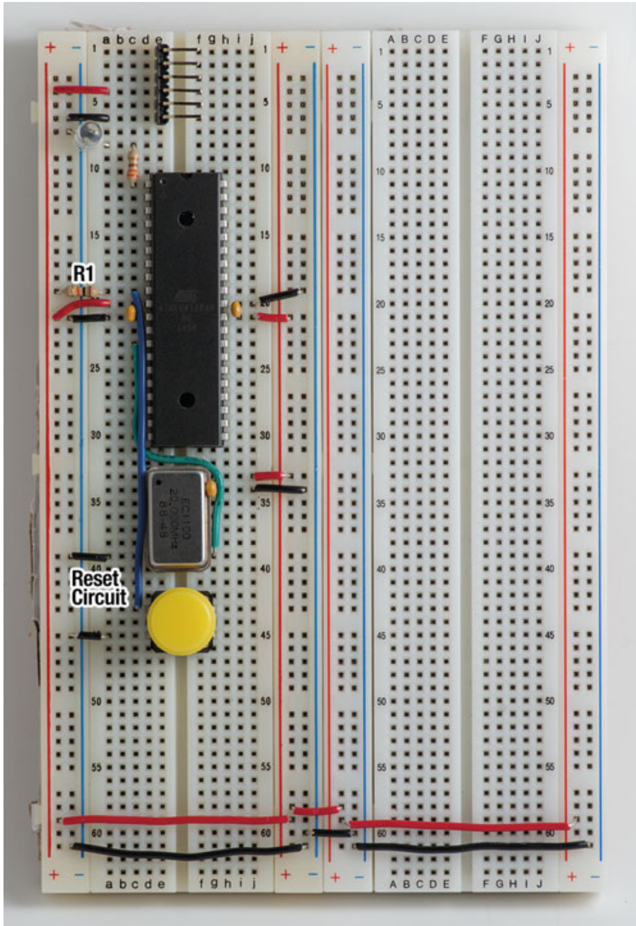

Figure 2-7. *The Reset Circuit*

TTL RS-232 Circuits

The last thing we need to connect is the TTL level RS-232 circuit. I talked about RS-232 and why Arduinos and the Cestino use it in Chapter 1, but because we're about to wire it up, we should take a moment to describe what it actually is. RS-232 is an ancient serial protocol, originally issued in 1969, to govern communications between teletypewriters. It transmits data single-file, one bit at a time, and to do this, it needs four wires: one for transmit (TX), one for receive (RX), and one for each end to signal that it's ready to communicate, RTS (Request to Send)from the transmitting side and CTS (Clear to Send) from the receiving side.

RTS-Reset Circuit

Arduinos and host computers are fast enough that they don't need the RTS/CTS mechanism to keep one end of the communication from overwhelming the other. Arduinos do, however, need to be reset so that the bootloader (covered in Chapter 3) will run and listen for serial communications for a few seconds before handing off control to the user sketch. Conveniently, RTS is active low, just like the ATmega's reset pin, so the Arduino designers repurposed the RTS line from the host computer (or, really, from the USB to TTL level RS-232 cable/board/whatever) to reset the ATmega prior to programming it. The RTS line is the first one in the connector, so we'll hook it up first.

There's a wrinkle. We're holding the reset pin at 5 volts with the pullup resistor. We don't know what resistance is present from the RS-232 RTS line to ground, so it'd be a good idea to keep the DC (steady) pullup voltage out of RS-232 and whatever steady RS-232 voltages are present out of the reset circuitry unless they're changing, which means the host computer has reset the Cestino. You may recall from earlier in this chapter and Chapter 1 that capacitors conduct only changing or alternating voltages. Just as we used them to short any pulsed or alternating voltages from our power supply lines to ground, we can use them to pass only pulsed currents from the RTS line into the reset circuitry. We'll use the remaining 0.1µF capacitor for that.

The capacitor needs to reach from tiepoint row 1 all the way to the other side of the connector (CONN1 on the Schematic in Figure 2-1) to one of the free rows of tiepoints there. I used row 10. If you're using a big ceramic disk capacitor as I did with the prototype, it will reach that far. If you're using the tantalum caps I recommended and used in the pictures, that's a problem. The leads of the capacitor can touch one of the other RS-232 connections or the dropping resistor for the LED. We need to insulate the leads of the capacitor with something. Tape would work, but if you have hookup wire, you have *lots* of insulation. There are probably little bits of it all over your work space right now. They'll work fine. Grab one of these pieces of spaghetti (that's actually the name for it—they used to sell it in rolls for vacuum tube electronics) and slip it on one of the leads of the capacitor, then bend and trim the leads of the capacitor as needed, and install it.

Now run a wire from tiepoint row 10, or wherever your capacitor wound up down to the reset pin—pin 9 on the ATmega. You can see mine in Figure 2-8.

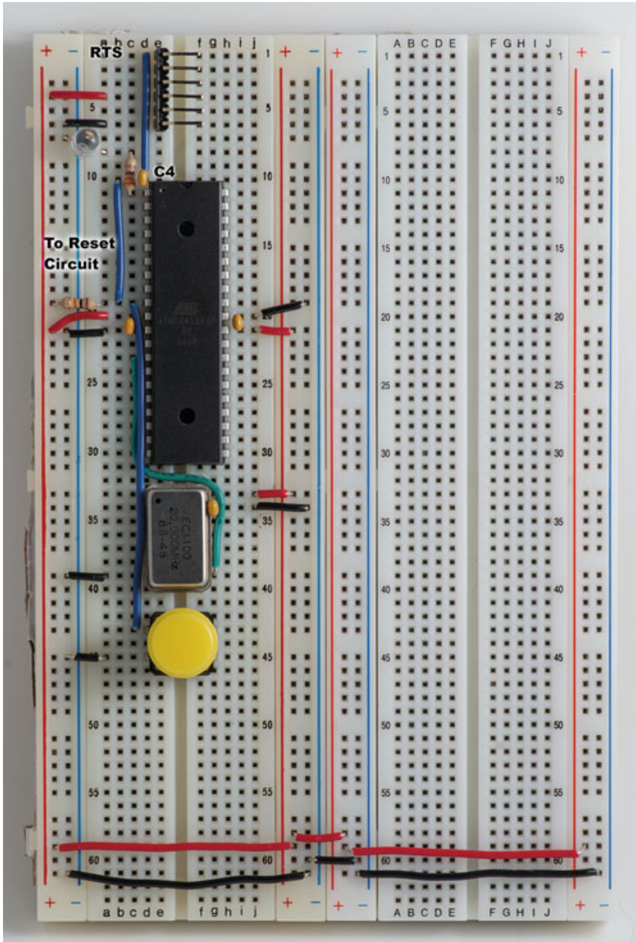

Figure 2-8. *The RTS-Reset Circuit*

TX, RX, and CTS Circuits

The rest of the RS-232 circuits are much simpler. Wire pin 2 of the connector, RX, to pin 15 of the ATmega. If you look at the pinout diagram in Figure 2-2, one of the functions of that pin is TXD0, or transmit data for USART0.

■ **Note** that a USART (Universal Synchronous/Asynchronous Receiver/Transmitter) is serial communications hardware built into the ATmega. We're using USART0 for RS-232 communication with the host computer. The ATmega1284P also has a USART1, but we won't be using it for anything.

In RS-232, transmit pins are always connected to receive pins, and vice versa. That being the case, wire pin 3 of the connector (TX) to pin 14 of the ATmega: RXD0 – Receive Data for USART0.

Pin 4, Vcc, is already tied to the + bus, and pin 6 is tied to the - bus. That leaves pin 5, CTS, which tells the host computer it's ok to transmit. It's active low, so we'll wire it to the - bus. It's always ok to transmit. You might notice in Figure 2-9 that I moved the LED to a different - bus tiepoint to make room for the CTS line's connection to ground.

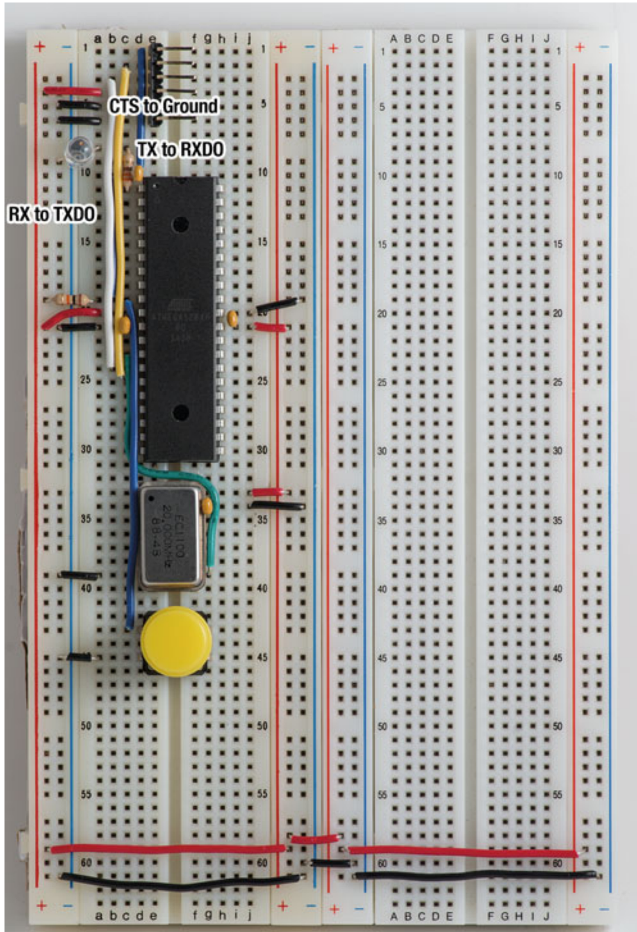

Figure 2-9. *TX, RX, and CTS Circuits*

Testing, Board Layout, and Static

There's not a lot of testing we can do to make sure the Cestino is wired correctly until we get the bootloader and core software loaded, and that happens next chapter.

For right now, when you connect the TTL level RS-232 connector up (RTS/the green wire on pin 1, Ground/the black wire on pin 6) you can rest your thumb on the ATmega. If it gets more than slightly warm to the touch, unplug and check your power connections. Also run your thumb over the oscillator for the same reason. They get screaming hot when their polarities aren't respected. If it gets that hot, you might want to break out the spare once you've fixed the problem.

If you've got a really bad short, especially in the power circuit connections, your host computer will complain that something is pulling *way* too much current. If that happens, disconnect and go over your power circuits again.

Nothing getting uncomfortably hot? No complaints from the host computer? Good. You can test the LED by plugging a jumper into the + bus, and connecting it to Pin 1 of the ATmega. The LED should light. If you wanted to test all the other + busses on your breadboard setup that way, it wouldn't be a bad thing.

Other than getting power polarity wrong, there's not much you can do that will damage the ATmega or the oscillator by connecting them to the + bus. The worst that will happen is nothing.

Likewise, you may hear dire warnings from people my age about static, and it's true that static electricity can destroy sensitive integrated circuits. The truth is, much of the panic about static came from first generation CMOS RAM, which was very, very sensitive to even a little static, completely unprotected against it, and *expensive*. We can be a little more cavalier with the ATmega and the associated components. They're cheap and robust. It's probably a good idea to discharge yourself to a metal file cabinet or something when you sit down to work on your Cestino and related projects, but that's as far as I'd bother. If you know you have a lot of static in your home or your workshop, or you've had really bad luck with it, you might want an antistatic wrist strap.

One other thing: From the photos, you'll probably conclude I'm fussy about board layout. If you watched me re-arrange the prototype over and over again, you might think I'm actually nuts on the subject. The truth is, I knew I was laying the board out to be photographed. It needs to show the circuits clearly during construction, and be reliable and durable, since I'm carrying the hero Cestino (the one in the photos) back and forth to my photography box. Your layout doesn't have to be so fussy. I will make one suggestion. Make sure, when all is said and done, that all the ATmega pins except the power pins have at least one empty tiepoint next to them, so you can connect those pins to the projects to come.

Credit Where Credit Is Due

For this project, I have stood on the shoulders of giants. Obviously, the Arduino LLC and community did the lion's share of the work, but Cestino would not have been possible without the work of the Sanguino and Arduino Mighty projects as well. It was they who first added the code necessary for the Arduino core and bootloader to run on the ATmega1284P. Those additions were subsequently picked up by the Optiboot bootloader

project, and the Arduino core project. All I had to do was use the most recent version of Optiboot, tie in to the standard Arduino core that comes with Arduino 1.6.5, and write a few configuration files to build a bootloader and core for a different, somewhat faster board. This is open source, when it works.

That the Cestino's reset circuit looks exactly like the Arduino Mighty's is likewise not a coincidence. The 20Mhz external oscillator configuration is, by contrast, my doing, after being exposed to TTL oscillators by building Sergey Malinov's Zeta computer. You'll hear more about that later.

Further

At the end of each project, there's a section called "Further." It's a place for ideas I've had but not tried, or variations on projects that were fun, or tinkering I'd like to do if I had unlimited time. For the Cestino hardware, you could download the Geda schematic for the Cestino, use Geda's board design tools to design a printed circuit board, and either have it made or make it yourself by various methods. You could then solder the Cestino up, giving you a board that's much more robust than the breadboard version.

■ **Hint** If you make a printed circuit board version of Cestino, use a socket for the ATmega itself, and perhaps for the oscillator. Being able to remove these can be helpful.

In truth, the "further" for this chapter is really the rest of this book, starting with Chapter 3, where we load the bootloader and the core software, and really bring the Cestino to life.

CHAPTER 3

Kick the Tires, Light the Fire

When we're done with this chapter, you'll have either a working Cestino or a pile of malfunctioning ICs and a bone to pick with me. We'll run a couple sketches on it for fun, to make sure it's working, and that we can program it, and to demonstrate that yes, it does function in a mostly Arduino fashion before we go our own direction. My hero Cestino, the one in the photos, is in exactly the same state, so I'm with you. Let's get started.

The Stuff You Need

New Parts

Your Assembled Cestino

A 120Ω resistor as a dropping resistor for the speaker. If your speaker is much higher or lower impedance than 8Ω, you'll need to calculate this value yourself.

New or Used Parts

The speaker

Your existing Arduino, for use as an ISP (In-System Programmer) to install the bootloader onto the Cestino.

Software

If you haven't already downloaded the Cestino package, now's the time. Make sure to get the latest version, if there's more than one by now.

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_3

Ditto the Arduino software itself. You will need Arduino 1.6.5 or later. Earlier versions will not work with the various add-ons in the Cestino package. Get it from `https://www.arduino.cc/en/Main/Software` directly. If you're feeling brave, you could even get the nightly build version on that page, but I haven't had good luck with it.

You'll need Mark Fickett's arduinomorse library. Get the most current version from github here: `https://github.com/markfickett/arduinomorse`. If you're already a subversion user, you can get it with that, obviously, or you can click on the *download zip* link on the same page.

Bootloader and Core

By now, I've used the words *bootloader* and *core* a lot. I'm going to use them a lot more, so we should probably be clear on what they are.

Bootloader

When you buy an Arduino, the microcontroller on it (usually an ATmega) comes preprogrammed with a bootloader. The bootloader is a small program that stays in program memory and lets you load sketches. The bootloader runs when the microcontroller is reset, and listens for a second (at least with the Optiboot bootloader we're using) for commands from AVRDUDE, which is inside the Arduino application. If it doesn't get anything from AVRDUDE, it transfers control to whatever sketch is loaded.

When you upload a sketch, the Arduino application calls AVRDUDE to do the work. AVRDUDE resets the Arduino using the RTS-Reset line (remember that from the last chapter?) and, within that second, establishes communications with the Arduino. Once communications are established, it uploads the sketch, and resets the Arduino again.

When the Arduino resets this second time, AVRDUDE doesn't have anything to say, so after one second elapses, it starts the sketch.

Core

The core software is a library of objects (in the Object Oriented Program sense) that sketches can call to talk to specific hardware on the microcontroller. When your sketch says digitalWrite (pin number), it's calling a method in one of these objects, rather than talking to the microcontroller itself.

This is called abstraction. Abstraction might seem like it reduces efficiency, and frankly, it does. In exchange, however, it lets the same sketch run on multiple models of Arduino, including the Due, which runs on a completely different microcontroller than the ATmega series. The abstraction of the core provides a standard interface between the sketch and the hardware.

A Little History of Software Abstraction

In the days of the 8 bit desktop computer, we used operating systems like DOS and CP/M. Both of these relied on a BIOS that was little different from the Arduino core, and both contained ways of loading user programs that are, if you squint enough, very similar to the bootloader. There were many programs where you put the floppy in the drive, hit the reset button or turned the computer on, and they booted themselves into the computer and ran.

Those old computers were slow. Every instruction mattered, so programmers were often tempted to bypass those interfaces and talk directly to hardware. This worked. They extracted far more performance from those old machines than anyone thought possible, but there was a cost.

If there were no interfaces defined in the BIOS or the operating system, or if you had software that was talking directly to hardware, you had to make sure those programs knew how to talk to your new hardware. Word processors, for example, had to know about your specific printer. On my CP/M machine today, it still makes me chuckle that Wordstar had to know how to print to various printers *itself*. Most of those printers are long extinct, but one of them, the Epson MX-80, introduced in 1980, spoke a proprietary printing language called ESC/P. Today, in 2015, 35 years later, Epson still lists impact printers on its website that speak ESC/P, and in theory, Wordstar for CP/M could print to them. Abstraction is a powerful thing.

Set Up the Arduino Application

Once you have the Arduino application version 1.6.5 or later running on your host system, go ahead and start it. Open the Preferences window.

Mac users, go to the menu bar at the top of your screen, and select Arduino ➤ Preferences.

Windows and Linux users, at the top of the Arduino window, click on File ➤ Preferences.

What we're looking for is the exact location of your sketchbook. There are default locations, but you can put it anywhere you like, and different platforms call it different things, so just look in the Sketchbook location: field of your Preferences window, open Finder or Explorer or Nautilus and navigate there. Linux and Mac users, you can cd there in a shell window if you prefer. (For all intents and purposes, a folder in the various GUIs and a Directory in a shell window are the same thing).

While you have the Preferences window open, I strongly recommend setting the Display line numbers option, as well as Show verbose output on both compilation and upload. This means there will be a lot more stuff going by in the lines at the bottom of the Arduino application window when you compile and upload a sketch, but it's information we may need for debugging.

■ **Tip** You really can put your sketchbook anywhere on your system. I actually have mine in my Dropbox folder so I can get at it from my Linux machine, my Mac, my Windows virtual machine ... you get the idea.

Once you're in the right place, you should be able to see all your existing sketches. If so, go ahead and shut Arduino down. It won't pick up the changes we're going to make while it's running.

The Arduino application is an extendable IDE (Integrated Development Environment), so we can add boards and utilities to it fairly easily if we put them where the application looks for them. We could edit the Arduino application's internal folders, but we don't have to. It scans the *hardware* and *tools* directories in your sketchbook, too, if they exist.

In the Cestino package you downloaded, there are three folders: cestino, Burn_Preflight, and Cestino_Sketches. Drop Cestino in the *hardware* folder, and drop Burn_Preflight in the *tools* folder. If either of these folders don't exist inside your sketchbook folder, go ahead and create them. Drop the Cestino_Sketches folder into your sketchbook folder itself.

Now when you start up the Arduino application, you should see a new line in the Tools menu item called Tools ➤ Bootloader Burn Preflight Check. When you select Tools ➤ Board, you should have a new option: Tools ➤ Board ➤ Cestino 1284p 20MHz using Optiboot.

ISP: In Circuit Programmer

Arduino application still running? Good. It's time to set your existing Arduino up as an In-System Programmer (ISP) so we can use it to burn the bootloader onto the ATmega1284P.

Wait, what?

Some ICs are programmable. The ATmega1284P certainly is, but other types such as EEPROMs (Electronically Erasable Read Only Memory), and flash ICs are, too. In the bad old days, as we'll see when we get into EPROMS in Chapter 8, you had to socket these ICs so you could take them out any time they needed programming, erase them, and burn a new program into them. An In-System Programmer, combined with either a proprietary system connector or a standard JTAG (Joint Test Action Group) port lets you program the IC without taking it out. Normally, ISPs are rather expensive, proprietary equipment. Atmel's programmers are reasonably affordable at about U.S. $60 each, and if you have one, you can skip this step and use it. Likewise, if you have the optional universal programmer, you can use that as well, but you'll have to extract the ATmega1284P from the breadboard to program it. Make sure you get pin 1 back where it belongs when you put it back.

■ **Note** If you have a proper ISP or a universal programmer that can program ATmega1284Ps, things are different for you. ISPs can usually be selected with the Tools ➤ Programmer menu item, and a great many are in there. Universal programmers will want a .hex file to upload, and the modern Arduino application can export these with the Sketch ➤ Export Binary menu item. With the verbose output options set in the Arduino application, it will even tell you where it put them.

Arduino as ISP

For those without a dedicated ISP, don't worry. There's an example sketch that comes with the Arduino application that lets most any old Arduino do the job of an ISP. I'm using an Adafruit DC Boarduino, but nearly any 5 volt Arduino with a separate USB chip should work. (At this time, ATmega32Ux boards like the Leonardo don't work with ArduinoISP). Your Cestino, once it's built, will also work.

How ArduinoISP Works

Your existing Arduino does three jobs during this process:

Provide power to the Cestino

Communicate with the ATmega1284P with the SPI protocol

Receive the Bootloader and Core software from the host system and forward it to the Cestino

The Arduino application will call a separate program built into it called AVRDUDE (AVR Downloader/UploaDEr). AVRDUDE handles all communications between the host computer and the AVR/ATmega, and generates most of the text that shows up in the debug window of the Arduino application. AVRDUDE will signal your Existing Arduino over the usual USB connection and ask it to open communications with the ATmega1284P. The ArduinoISP sketch lets your existing Arduino do exactly that.

AVRDUDE tells ArduinoISP to program the "fuses" of the ATMega1284P. These software-configurable switches, among other things, tell the ATmega1284P where to get its clock signal, what kind of signal to expect, and how to calibrate its internal delays so it can communicate properly with whatever clock it's been configured for. It's an absolutely critical step.

AVRDUDE then sends the bootloader and core, byte by byte from the files you installed in the hardware folder of your sketchbook folder.

If all goes well, the last thing that happens is that AVRDUDE resets the ATmega1284P, and when the ATmega finishes resetting, it will be a functioning Cestino.

Set Up the ArduinoISP Sketch

Connect your existing Arduino to your host computer with the usual USB connection. Is your Arduino so old it's using 9 pin RS-232 and has an external power supply? Awesome. That will work fine, too. Check the Board and Port settings and make sure both are correct for your existing Arduino. You've probably loaded hundreds of sketches on your Arduino already. That's all we're doing at this point.

In the Arduino application, select File ➤ Examples ➤ ArduinoISP, and upload that to your Arduino.

Set the Programmer and Board in the Arduino Application

Once you have successfully loaded your existing Arduino with the ArduinoISP sketch, it's time to configure the Arduino application so that it sends the right data to the right place. Things can get a little confusing doing this.

Remember, we're not uploading a sketch to your existing Arduino. We did that. The ArduinoISP sketch is running on your existing Arduino, waiting for you to send data to it over the usual serial port, so it can send that data to the Cestino's ATmega1284P. We need to tell the Arduino application what data to send that will ultimately wind up on the Cestino.

So hit the Tools ➤ Board menu item and go all the way down to the bottom and select Tools ➤ Board ➤ Cestino 1284P 20Mhz Using Optiboot item. This tells AVRDUDE what bootloader to send. Then select Tools ➤ Programmer ➤ Arduino as ISP.

Make sure you select Arduino as ISP and not ArduinoISP, despite the sketch being called that. The ArduinoISP programmer is a dedicated ISP with an unfortunate and confusing name, and it communicates differently. We want Arduino as ISP. If it sounds like I spent hours chasing nonexistent bugs from getting this wrong, you've got a good ear.

Leave the port setting the same.

Wire your ArduinoISP into the Breadboard

Now that we have the software all set up, it's time to wire your existing Arduino into the Cestino breadboard, as shown in Figure 3-1.

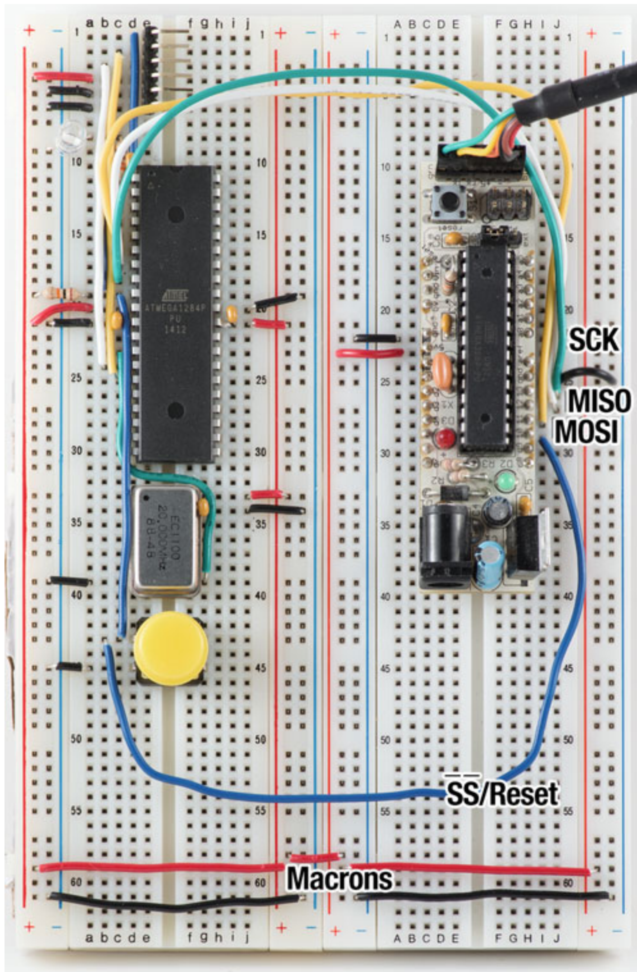

Figure 3-1. ArduinoISP wired to Cestino

■ **Note** SPI stands for Serial Peripheral Interface. It is a synchronous serial protocol, which means it transfers one bit at a time in each direction, synchronized to a clock signal. There are at least two devices on a given SPI bus, the master and at least one slave. The master device sends the clock via the SCLK (sometimes called SCK) line and signals which slave may communicate with it via the Slave Select, or SS line. The MOSI (Master Output Slave Input) line sends data from the master to whichever slave is selected, and MISO (Master Input Slave Output) line sends data from the slave that is selected to the master.

In order for SPI to work, obviously, there must be four connections between the devices: SCLK, MOSI, MISO, SS, as well as a ground connection. In order for your Arduino to burn the bootloader onto the Cestino, we're going to ask your Arduino to provide 5 volt power as well.

Table 3-1 shows what SPI signals are on which pins for several of the most common Arduino types, as well as on the Cestino. Notice that these signals are also found on the ICSP port on most Arduinos and Arduino compatibles, so in many cases you can use the ICSP port instead. You'll need female-to-male jumpers to access those pins, however.

Table 3-1. *SPI Pin Diagram*

Board	MOSI	MISO	SCK	$\overline{SS}$	Reset
Uno, Duemilanove, and most ATmega328P based clones	pin 11 ICSP 4	pin 12 ICSP 1	pin 13 ICSP 3	10	Reset ICSP 5
Mega 1280/2560	pin 51 ICSP 4	pin 50 ICSP 1	pin 52 ICSP 3	53	Reset ICSP 5
Cestino	pin 6	pin 7	pin 8	pin 5	pin 9

On the Cestino, the SPI port begins at pin 5, with the $\overline{SS}$ pin. MOSI is on pin 6, MISO is on pin 7, and SCK is on pin 8. Since the ATmega1284P is the slave in this arrangement, the $\overline{SS}$ line from your existing Arduino is wired to the Cestino's reset pin. (The Cestino's own $\overline{SS}$ line is only active when it is the master.) Make sure to connect it directly to the Cestino's reset line, and not on the ground side of the reset button or the RS232 side of the coupling capacitor.

Wire the Arduino's MOSI line to pin 6, the Cestino's MOSI line. Wire the Arduino's MISO line to the Cestino's MISO line on pin 7, and wire the Arduino's SCK to pin 8 on the Cestino, the SCK pin.

But wait. There's more.

Remember that any voltage is relative. Without a ground, SPI's voltages have no common reference between the Arduino and the Cestino. So wire a ground from one of the many GND pins on your Arduino to the - bus on the Cestino board, preferably close to the ATmega1284P. Because we're asking your Arduino to power the Cestino as well, wire one of the +5v lines on your Arduino to the + bus on the Cestino.

Now, hopefully, we're ready.

Preflight Check

I know it's tempting. I've done this hundreds of times now, and I'm still tempted to assume everything is right and hit the burn bootloader button. Don't do it. Not yet. There are a lot of things that can go wrong. If they do go wrong, as I discovered the hard way, you can make the ATmega1284P unable to communicate at all. You can brick the chip. The problem is the fuses.

Fuses, in ATmega parlance, are programmable switches used to configure the ATmega. Once upon a time, these would be actual fuses, which could be blown (once) to set configuration. They set critical parameters like, among other things, whether the

ATmega can be programmed over its SPI port, what kind of clock it's using, and what speed the clock is running at. If these parameters are set wrong, your ATmega may not be able to boot at all, or if it can, it may not be able to communicate, even over SPI. If SPI doesn't work, for whatever reason, your existing Arduino running ArduinoISP will not be able to program the ATmega. If the wiring between your existing Arduino and the ATmega1284P on the Cestino isn't right, or is noisy, AVRDUDE will try to set the fuses anyway. The results are usually wrong.

It's not all bad. If you have a universal programmer, it's fairly straightforward to load the ATmega into it and manually set the fuses back to their default values. I've done that a lot, but I had to assume that you, the reader, might not have a universal programmer, and that leaving you with a bricked ATmega1284P in the third chapter of this book wasn't going to go over well, so it would be better to avoid messing up the fuses in the first place.

Remember Burn_Preflight that you dropped into the *tools* folder of your sketchbook? That's what it's for. Got a power light on your existing Arduino? Everything hooked up? Good. Let's see if it works.

Go to Tools ➤ Bootloader Burn Preflight Check and select it. With the verbose output turned on for compile and upload as I suggested, you should see some output. Lots of output. It should look something like Listing 3-1.

Listing 3-1. Successful Burn Bootloader Preflight Check

```
Hello from Burn_Preflight
/Applications/Arduino.app/Contents/Java/hardware/tools/avr/bin/avrdude
-C/Applications/Arduino.app/Contents/Java/hardware/tools/avr/etc/
avrdude.conf -v -patmega1284p -cstk500v1 -P/dev/cu.usbserial-
AH02F147 -b19200

avrdude: Version 6.0.1, compiled on Apr 14 2015 at 16:30:25
         Copyright (c) 2000-2005 Brian Dean, http://www.bdmicro.com/
         Copyright (c) 2007-2009 Joerg Wunsch

         System wide configuration file is "/Applications/Arduino.app/
         Contents/Java/hardware/tools/avr/etc/avrdude.conf"
         User configuration file is "/Users/jim/.avrduderc"
         User configuration file does not exist or is not a regular file, skipping

         Using Port                    : /dev/cu.usbserial-AH02F147

[Lots of stuff snipped for brevity]

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.05s

avrdude: Device signature = 0x1e9705

avrdude done.  Thank you.
```

Got something like that? Is the device signature 0x1e9705? Good. That's a successful preflight check. You might want to read the error messages for future reference, but you don't have to. Got something else? Read on.

Programmer Not Responding

If your output looks more like this:

```
Hello from Burn_Preflight
[Lots of stuff snipped out here]
         Using Port                    : /dev/cu.usbserial-AH02F147
         Using Programmer              : stk500v1
         Overriding Baud Rate          : 19200
avrdude: stk500_recv(): programmer is not responding
avrdude: stk500_getsync() attempt 1 of 10: not in sync: resp=0x00
avrdude: stk500_recv(): programmer is not responding
[Lots more retries]
avrdude: stk500_getsync() attempt 9 of 10: not in sync: resp=0x00
avrdude: stk500_recv(): programmer is not responding
avrdude: stk500_getsync() attempt 10 of 10: not in sync: resp=0x00
avrdude done.  Thank you.
```

you need to check that the ArduinoISP sketch is running on your Arduino, and that the 5v and GND lines are connected to the + bus and the - bus of the Cestino correctly.

If your output says that the programmer is not responding but you're not getting multiple attempts, like this:

```
Hello from Burn_Preflight
[lots of stuff cut out]
         Programmer Type : STK500
         Description     : Atmel STK500 Version 1.x firmware
         Hardware Version: 2
         Firmware Version: 1.18
         Topcard         : Unknown
         Vtarget         : 0.0 V
         Varef           : 0.0 V
         Oscillator      : Off
         SCK period      : 0.1 us

avrdude: stk500_recv(): programmer is not responding
```

check the SS/Reset circuit and make sure it's properly connected. If you're sure it is, try pressing the reset button on both the Cestino and your Arduino.

Programmer not responding errors can also come from USB getting into the wrong state. If you're getting them, and you've tried everything else, try powering the host computer all the way down and turning off any USB hubs between it and your existing Arduino. Particularly with Linux hosts, there seem to be some issues in the Arduino Application's USB handling that leave the usb port in an unusable state.

Device Signature Error

Sometimes the preflight check will appear to run, but the results aren't right. You might get output like this:

```
Hello from Burn_Preflight
[Lots of other stuff cut]
avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.07s

avrdude: Device signature = 0x1e9705
avrdude: Expected signature for ATmega328P is 1E 95 0F
         Double check chip, or use -F to override this check.

avrdude done.  Thank you.
```

It looks like a successful read, but AVRDUDE is complaining. Notice where it says Expected signature for ATmega328P is 1E950F. We're not trying to program an ATmega328P. We're trying to program an ATmega1284P. Go back to Tools ➤ Board and select Tools ➤ Boards ➤ Cestino 1284p using Optiboot, all the way at the bottom of the menu.

If the expected signature is 0x1E9705, but the device signature you got is 0x1E9706, then despite my warnings you got the ATmega1284 instead of the ATmega1284P. You'll have to stop and order an ATmega1284P.

Device Signature Yikes Error

If your output looks like this:

```
Hello from Burn_Preflight
[Lots of stuff snipped out]

Reading | ################################################## | 100% 0.07s

avrdude: Device signature = 0x000000 (retrying)
```

```
Reading | ################################################## | 100% 0.06s

avrdude: Device signature = 0x000000
avrdude: Yikes!  Invalid device signature.
         Double check connections and try again, or use -F to override
         this check.
avrdude done.  Thank you.
```

Check your SCK, MISO, and MOSI lines. Remember that SPI, unlike RS232, expects the same line on both devices to be connected together, so MOSI goes to MOSI, MISO goes to MISO, and SCK goes to SCK.

Could Not Find USBtiny Device Error

This error is my favorite. This is what it looks like if, despite my warning, you selected ArduinoISP as the programmer instead of Arduino as ISP.

```
Hello from Burn_Preflight
[Lots of stuff snipped out]
         Using Port                    : usb
         Using Programmer              : arduinoisp
avrdude: Error: Could not find USBtiny device (0x2341/0x49)

avrdude done.  Thank you.
```

If you see this, click on the menu in Tools ➤ Programmer and scroll down to Tools ➤ Programmer ➤ Arduino as ISP and select that.

Burn the Bootloader

Everything ready? Good. Getting clean preflight checks? Good. Let's hook our little monster up to the lightning. Click on Tools ➤ Burn Bootloader.

Wait for it ...

Is the LED flashing, as in Figure 3-2? In groups of three flashes? That means the bootloader is restarting continuously. That's a good thing. It's a heartbeat.

Congratulations. Your Cestino is alive. It's on life support from your existing Arduino, but it's alive. Let's see if it can stand on its own. Disconnect your Arduino from the Cestino and plug the USB to TTL cable into the 6 pin header on the Cestino, making sure that the ground wire on the cable is on the pin connected to the – bus (black toward the ATmega1284P, green away from it, assuming your breadboard looks like mine).

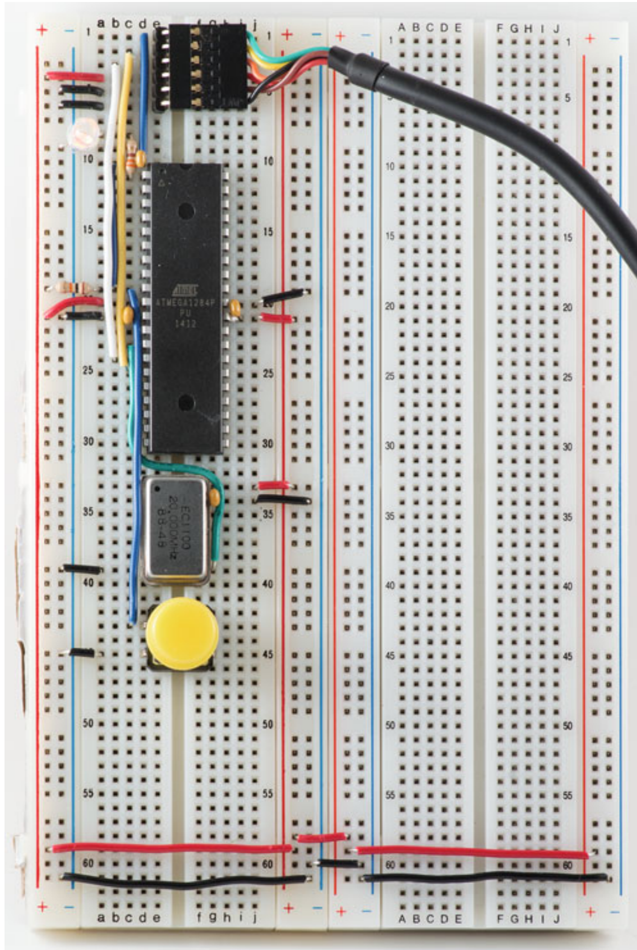

Figure 3-2. *Cestino Powered with LED Flashing*

Blink Pin 1: It's Alive!

The usual sketch for this moment is the Blink sketch from the Examples dropdown of the Arduino application. Unfortunately, it won't work for us. The standard Arduino has its LED on pin 13. Because we're using the ATmega1284P's natural pin layout, our pin 13 is where the clock signal goes in, so a sketch can't access it. Our LED is on pin 1. This is something we're going to have to get used to any time we use a traditional Arduino tool, so we might as well start now. Go to File ➤ Examples ➤ Basics ➤ Blink and open it.

Some versions of the Arduino app aren't as careful as they should be about not letting you write to the examples, so we'll just make a copy and work on the copy. Go to File ➤ Save As... and save it as blink1.

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  Most Arduinos have an on-board LED you can control. On the Uno and
  Leonardo, it is attached to digital pin 13. If you're unsure what
  pin the on-board LED is connected to on your Arduino model, check
  the documentation at http://www.arduino.cc

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald
 */

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

Notice how everything refers to pin 13? Change all those references to pin 1, so it looks like this:

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

[Comments snipped for brevity]

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald
 */
```

```
// Modified 20-June-2015 to use pin 1 (portb pin 0) of the Cestino board
//by Your Name Here.

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(1, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(1, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);             // wait for a second
  digitalWrite(1, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);             // wait for a second
}
```

Please, please, please make sure you annotate this version of Blink to tell the world what you did with it, and what those changes are for. In a year, when you accidentally grab Blink1 instead of Blink for a regular Arduino, you'll thank yourself when that comment pops up to tell you why it won't work.

You can also use my version of blink1 in the Cestino_Sketches folder.

Or Not So Much: Troubleshooting

If your Cestino passed the Bootloader Burn Preflight Check, and you get the continuous sequence of three flashes, a pause, then three more flashes after you bootload it, the Cestino is correctly programmed. The error messages you get should be largely the same as the ones from the preflight check, but recovering from some errors may be a lot harder.

Programmer Not Responding

If the Programmer Not Responding error rears its ugly head again during bootloader burning, you have a couple options. Re-run the preflight check and see if it shows up there too. If it does, it's possible your USB port has gotten into an unusable state. Current versions (1.6.5) of the Arduino application seem to have some intermittent USB problems, particularly on the Mac, so try shutting everything down. Make sure the pin 1 LED on the Cestino goes out and stays out, and then start the system up again (cold start it) and see if the preflight check works now. If it does, go ahead and try the bootloader burn.

Wrong Sketch?

If loading the bootloader seemed to go smoothly, you're getting the three flashes, pause, three flashes output, and the sketch uploaded, but nothing's happening, press the reset button on the Cestino. If you get the three flashes, that means the bootloader is starting, but for whatever reason, the sketch isn't lighting the LED.

Double check that your sketch is trying to light pin 1 and not pin 13, or that you didn't accidentally upload the empty demo sketch. Weird things happen late at night.

Power or Clock Problems, Perhaps?

Is your Cestino getting any power? Set your multimeter to DC volts (Direct current: current which does not oscillate. Like the current from a battery) and to a scale where 5 volts will show up well (on my cheap meter this is the 0-20VDC scale) and go over the pins on the ATmega1284P itself and make sure you get 5 volts between pins 10 and 11, (pin 10 is Vcc, 11 is Ground) and the same five volts between pin 30 (Ground) and 31 (AVcc) Then check the oscillator. Pin 7 is ground, pin 14 is Vcc.

Is your Cestino getting a clock signal? Disconnect power from the Cestino (pull the USB plug out of the host computer) and set your meter for Ohms (Ω) on whatever the lowest scale is. Check to make sure pin 8 of the Oscillator and pin 13 of the ATmega1284P are connected. Also make sure nothing is plugged into pin 12. if something is plugged in there, it can jam up the signal from the oscillator, and that will keep your Cestino from running.

Also, check to make sure the oscillator is a 20Mhz oscillator. 16Mhz may work for uploading the bootloader (SPI's timing is handled by its external SCK line) but it won't work for RS232, which is what we're sending the Cestino directly. If the delay between the three-flash sets seems much longer than a second, be suspicious. It's noticeably slower even with a 16Mhz oscillator.

You might wonder how the Cestino could pass the preflight check without having a proper clock signal. As they come from the factory, ATmega1284Ps use an internal 8Mhz clock. As long as the IC has power and ground, it will work. Once we set the fuses, however, the ATmega is looking for a 20Mhz external oscillator on pin 13, and nothing else will do. If you have an oscilloscope or a counter or a logic analyzer capable of reading 20Mhz signals (wish I had one of those) you can check to see if the clock signal looks good. If it doesn't, or you're suspicious of the oscillator in any way, replace it. They're cheap. You might consider getting a second ATmega1284P in the same order, as long as you're paying postage.

ATmega1284P Is Bad?

Sometimes you get a bad IC. Whether they were bad from the start, or something happened to them in storage, or shipping, or your parts bin, some days things just don't work. If your ATmega is still dead cold after a couple hours of being connected to power, or if for any other reason you think your ATmega is dead (or bricked), pull the USB cable from the host computer to disconnect power, and gently slip your screwdriver under the ATmega and lift it from the breadboard a bit. Then slip the screwdriver further under it, lift some more, and so on until the whole IC is free.

Before you toss the ATmega, take a quick look at its pins. Obviously, they should all stick down, and they should all be in two neat lines, one on each side of the package. Sometimes, when you push an IC into your breadboard (or a socket), a pin or two won't be aligned right and they'll get mashed into the bottom of the package instead of connected. Diabolically, from the top, you usually can't tell. With the IC in your hand, you can. If one of the bent pins is a power pin, the clock pin, one of the RS232 or SPI pins, that could very well be the problem. *Gently* bend the pin back to position. You can bend the pins on an IC once, sometimes twice and get away with it. Further bends will snap them off, and if one snaps off, it's time to replace the ATmega. If you have part of the pin left, and you want to try soldering a piece of wire to it, be my guest, but ATmegas aren't that expensive.

The Morse Code Practice Translator

Hopefully by the time you get here, your Cestino is sitting, merrily flashing its LED at you. We could do a lot of tests to verify its performance (I did) but frankly, that's dull. Let's have some fun with it instead. We'll do it the traditional Arduino way, before we head off into the woods. We're going to build the Morse Code Translator, shown in Figure 3-3.

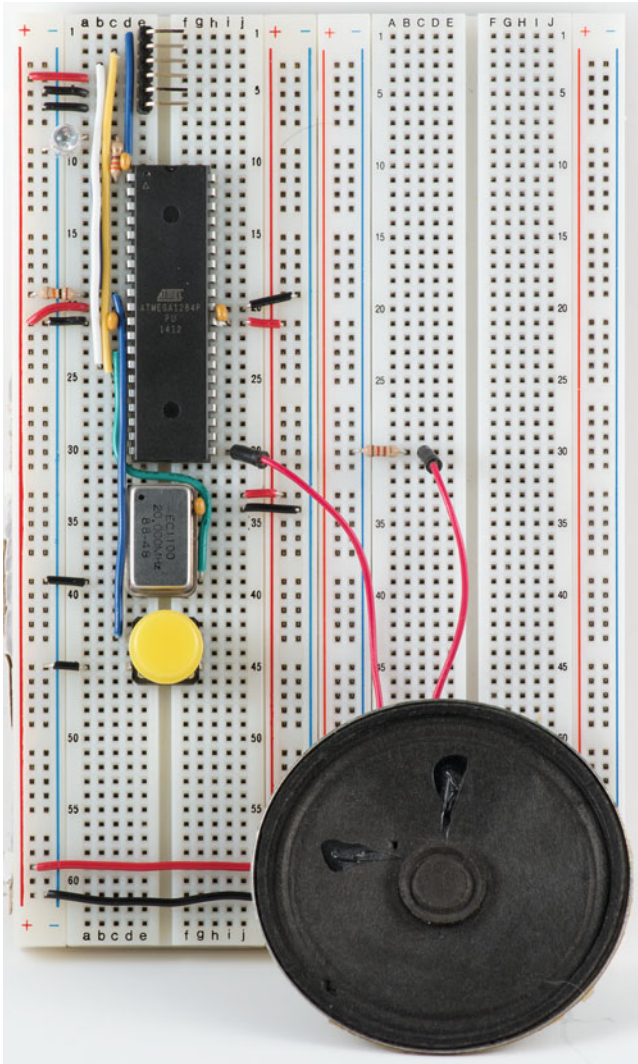

Figure 3-3. *Morse Code Practice Translator*

What the Morse Code Translator will do is take messages from the host computer and translate them into tones over a speaker. I only know a little Morse, but if you're learning, a computer can give you the kind of endless repetition it takes to commit this ancient code system to memory.

Download the Arduinomorse Library and Install It

We could write our own Morse code library, but this is a mini-project, and it's been done, so let's just download a library instead. It's the Arduino way. If you haven't already done so, go to `https://github.com/markfickett/arduinomorse` and either download the library with subversion or click on the *download zip* link, and uncompress the archive. Either one will work. Drag the *arduinomorse-master* folder into your sketchbook folder and drop it in the *libraries* folder. If the *libraries* folder doesn't exist, go ahead and create it. Terminal users in Linux or Mac, cd, cp, and mkdir work, as always.

The arduinomorse-master library contains several objects that we need to write the Morse Code Practice Translator.

A Quick Introduction to Object Oriented Programming

Nearly all Arduino sketches are written in a mishmash of C and C++. Mostly they're C, an old school functional language, except where they need to call parts of a library, or some parts of the Arduino core. The arduinomorse library is written in C++, and it's very object oriented, so now's a good time to talk about what object oriented programming is, and what you're looking at.

There are four basic concepts you have to understand to grasp object oriented programming. Different languages call them different things, but we'll use the C++ names.

Objects

An object is a data structure that can contain attributes and methods.

Attributes

An Attribute is what the rest of the programming universe calls a variable. Attributes can be public or private, which determines whether outside objects can touch them directly or not.

Methods

A method is what the rest of the programming universe calls a function. Like attributes, they can be public or private, which determines whether outside objects can touch them directly or not.

Classes

This will take some explanation. In C, there are declarations called `typedef`. Typedef allows me to assign a new name to a variable type. If I wanted to make sure a given numeric variable has only one byte, I might declare it like this:

```
typedef uint8_t mynumber;
```

A uint8_t is a single byte, unsigned, so we can use all eight bits of it. You may see this again when we start writing bytes to ports. This typedef doesn't declare a variable. It declares a type. If I want a variable of the mynumber type, I declare it like this:

```
mynumber george;
george=0;
```

Here I've declared the variable George, and set George to 0. Pretty straightforward stuff.

A class is like typedef for objects. It does not declare an object itself. It declares what an object will be like if one is declared (instantiated is the object-oriented equivalent).

In C++ (but not Java) I can do this:

```
// Include the libraries and classes we need for this program.
#include <stdint.h>
#include <iostream>

// Declare my class here
class myclass {
        // Declare our typedef
        typedef uint8_t mynumber;

        //Declare our private attribute of type mynumber.
        //No code outside the object can touch george.
        private:
                mynumber george;

        // Declare our public methods.
        // These can be called from outside the object.
        // We could have public attributes too.

        public:
                // Void means we're returning no values.
                // Note the cast, where we turn the integer input into a
                   mynumber.
                void set(int input) {
                        george = (mynumber)input;
                }

                // Here we're returning an integer.
                int read() {
                        // Note the cast - we return the value of george,
                        // a mynumber attribute as an integer.
                        return (int)george;
            }
};
```

```
// Here is where we create an object of the type myclass. We call it
// myobject. We're declaring globally for the entire program.
myclass myobject;

// Main must return an int.
int main(){
        // Call the set method
        myobject.set(255);
        // Call the read method over here vvv
        std::cout << std::to_string(myobject.read());
}
```

■ **Note** This example is written to compile and run in vanilla C++. Sketches don't contain a main( ) because the Arduino app puts one in behind the scenes that calls setup( ) and loop( ) for us.

Looking at the code, you can see that it declares a class at the top: myclass. That class contains a typedef: `mynumber`, a private attribute (aka variable) `George`, which is of the type `mynumber`. `George` is declared private, so code outside the object isn't allowed to change the value of `George`, or read `George` directly, or even know that `George` exists. This is the default behavior for anything inside a class, but it's declared explicitly here to make it obvious.

The class also contains two public methods, `read()` and `set()`. Because these are public, they can be called from other code.

The next interesting thing is where we instantiate an object of the type `myclass`. I called it `myobject`. It's exactly like declaring an integer or a char.

Once the object is instantiated, we can use it. Because it's declared globally (outside any other function or object), we could use it from any function or object in the program.

We'll use it from `main()`, because this is vanilla C++, but if you were using it in Arduino code, you could call it from `setup()` or `loop()`. The command myobject.set(255) calls the set method we defined in the class and instantiated when we declared `myobject`. The `set()` method can modify the value stored in George inside `myobject`, and it's a public method, so 255 gets stored in George. The function call `myobject.read()` embedded in the cout (C++'s "write this data to the terminal" command, rather like `Serial.print()`) reads the value of George inside myobject. Again, it's a public method, and it returns an integer, which we have to convert to a string, because unlike `Serial.print`, cout can only handle strings.

There's a lot more to object orientation, and it gets brutally complicated as you go deeper into it. Arduino libraries are usually set up as classes. Arduinomorse is like that. The basic rules of object oriented C++ are: we can't touch private attributes or methods; we can touch public attributes and methods; and in order to use a class, we must instantiate an object (declare a variable of that type) of it first. Knowing those rules will get you as far as we need to go in this book.

Hook Up the Hardware

For the Morse Code Practice Translator, as mentioned earlier, you need a 120Ω resistor and a speaker with wires you can connect to the breadboard. Wire the resistor from pin 21 on the ATmega1284P to the same row of tiepoints on the other breadboard. Make sure not to plug it into either of the + or - busses that are under it. It won't hurt anything, but it won't work. Wire one side of your speaker to that row of tiepoints, and the other lead to the - bus.

How the Hardware Works

It doesn't get much simpler than this. There's a method in the Arduino core called `tone()` which generates a tone of a given duration on a given pin. This does all the tone making for the arduinomorse library. One side of the speaker is connected to pin 21, because it's handy, through the 120Ω dropping resistor so we don't overload pin 21 (remember, outputs are limited to 40mA). Dividing 5 volts dropping over 128Ω (the resistor plus the speaker, as they are in series) we get about 0.039 amps, or 39mA. Just inside the limit. My speaker has a maximum rating of 0.5W (Watts), so if we multiply 0.039A times 5 volts, we get 0.195 Watts, less than half the limit for the speaker.

A Quick Word on Impedance

Speakers are inherently Alternating Current devices, and AC is… different. Here's why. Where simple resistors offer only resistance to AC or DC circuits, capacitors and inductors (coils, like the ones that drive our speaker) also have reactance, which, in the case of the speaker, is a measure of how long it takes the magnetic field created to collapse and reverse. The voltage drop that a reactance produces is delayed from the corresponding drop in current, too.

Our speaker has a DC resistance of about 8Ω. This is its minimum impedence. When faced with an AC signal, its reactance will be higher in most cases. It won't go lower. Our pin's output impedance is about 125Ω (divide 5 volts by .040 amps), assuming the 40mA limit factors all the reactance in. That's very close to the resistor-speaker value of 128Ω described above. You want the impedance of the speaker and the port to be about the same to transfer the most current. It's not super-important for this circuit, as we don't care about the sound quality of the audio, but it's something to bear in mind.

The Code

```
// Morse Practice Oscillator
// Copyright 2015,2016 James R. Strickland
//-------------------------------------------------------------
// This program is free software: you can redistribute it
// and/or modify it under the terms of the GNU General
// Public License as published by the Free Software
```

```
// Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will
// be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE.  See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public
// License along with this program.  If not, see
// <http://www.gnu.org/licenses/>.
// ---------------------------------------------------
// Morse Code Practice
// This program pulls in morse.h from Mark Fickett's
// "arduinomorse" library and impliments a simple serial
// text to morse code practice sounder.
// ----------------------------------------------------
#include <morse.h> //Pull in the morse.h library.
#define PIN_SPEAKER 21 //define the correct speaker pin.
#define PIN_LED 1  //Define what pin our LED is on.
/* ----------------------------------------------------
   Create the C++ object "receiver" of the
   SpeakerMorseSender type.
   Set the output pin to PIN_SPEAKER, set the tone frequency
   to 440hz, -1 carrier frequency (it just adds noise), and
   set the morse code words per minute to 20. If you're
   new to morse code and trying to learn it, 5.0 might
   be a better value.
   If you'd prefer to read the Morse as LED flashes instead,
   Comment out the SpeakerMorseSender instantiation and
   uncomment the LEDMorseSender instantiation instead.
   ---------------------------------------------------*/
SpeakerMorseSender receiver(PIN_SPEAKER, 440, -1, 20.0);
//LEDMorseSender receiver(PIN_LED, 10.0);

/* Create the String object "FromSerial" because traditional
 *  C style string handling is painfully bad. */
String FromSerial = "";

/* The setup() function is one of the standard boilerplate
 * arduino functions. It executes once, so we put all our
 * configuration code in it. */
void setup() {

  Serial.begin (115200); //Set up the serial console at 115200 Baud.
  receiver.setup(); //tell receiver to set itself up.
```

```
/* Set the message to be sent to CQ. CQ is traditional
 * Morse shorthand inviting operators who hear it to respond. */
  receiver.setMessage(String("cq "));

/* Now send the message. Blocking means the sketch can do
 * nothing else while we're sending. */
  receiver.sendBlocking();
}

/* the Loop() function is one of the standard boilerplate
 *  Arduino functions too. It is called over and over again,
 *  forever. */
void loop() {

 /* Serial.available() returns the number of bytes available,
  *  if any, from the serial port. If that number is not zero.
  *  If that number is equal to zero, it is equivilent to
  *  logical false, otherwise is it logical true. We don't
  *  care how many bytes are available, just whether they
  *  are or not. If bytes are available, use Serial.read
  *  to read them into the FromSerial String object until they
  *  are gone. We assume the user won't send us strings that
  *  are too long. */
  if (Serial.available()) {
    FromSerial += (char)Serial.read();
  } else
  if (FromSerial.length()) {
    receiver.setMessage(FromSerial);
    receiver.sendBlocking();
    FromSerial = "";
  }
//Remember, in the loop() function, we repeat forever.
}
```

How to Use the Sketch

Open the Serial Console and type the words you want to here in Morse code into the line next to the Send button, at the top. Then press the Send button or hit return. You should hear the familiar long and short beeps of Morse code from the speaker.

How the Code Works

Most of this project happens in the code, and there are two parts of it: the main sketch, and the library.

The main sketch includes `morse.h` from the Arduinomorse package. You can add packages manually, or you can click on Sketch ➤ Include Library and select the library you want from all the libraries the Arduino app knows about.

Next, the sketch defines `PIN_SPEAKER` (and `PIN_LED`). Note the lack of semicolons after these lines, and the hashtag in front. These are preprocessor macros.

The preprocessor is part of the C/C++ compilation system. It allows you to automatically make changes to the code when it's compiled. It's often used to make code more portable, so you can build it on multiple systems. In this case, the #define macros are simple search and replace macros. Every time the preprocessor finds the word `PIN_SPEAKER` in the code, it will replace it with `21`.

In Arduino programming it's very common to see a lot of macros, particularly in the guts of the bootloader and the core. Unlike a variable or a pointer, a macro takes *no* memory from the ATmega. None. Zip. The macro is processed at compile time.

Next, the sketch instantiates `receiver` as an object of the `SpeakerMorseSender` class. Remember, `receiver` is the object, `SpeakerMorseSender` is the class.

Next, the sketch instantiates the object `FromSerial` of the class `String`. The Arduino `String` object has a lot of useful methods and is easy to use. I called the object `FromSerial`, because it will be getting data from the serial console. More on that further down.

In the `setup()` function, we tell the sketch it will be communicating with the serial console with `Serial.begin(57600)`.

■ **Note** Serial.begin(115200) sets the RS232 speed to 115200 baud. Make sure your serial console is set to the same speed, or the serial console and the Cestino won't understand each other. I use 115200 baud for the remainder of this book.

Once serial is set up, we call receiver's method `setup()`. Setup is a method of the SpeakerMorseSender class that tells the object to set the speaker pin as an output. I know this, because I poked through the library code and saw it.

Next, we call receiver's method `setMessage()`. We set the message to "cq", which is Morse code shorthand for "Anyone who can hear this, please respond." Notice that we cast "cq" which is a character array by default, into a String object, which is what the arduinomorse library wants.

In the next command, we call receiver's method `sendBlocking()`, which tells receiver to go ahead and beep out the message we sent.

Why are we sending a message in setup? Because it's handy to know that the speaker is connected properly and everything is working when the sketch starts.

■ **Note** It's been said that the telegraph was the 19th century's Internet. Morse code was invented in the 1830s (not coincidentally around the same time Ohm's Law was developed) and used heavily for over a century. Just like Twitter has its own slang—lol, rotfl, ttyl, and so on—Morse code has traditional slang, too, and for the same reason. It's quicker and easier to send.

Blocking just means that the sketch can't do anything else while it's beeping out Morse code. The library can send nonblocking, but we don't need it in this case.

Once we enter the `loop()` function of the sketch, one of two things happen over and over again. If the `Serial.available()` method returns anything but zero (logical false) we read one character and add it to the FromSerial String object. Otherwise, if the `FromSerial.length()` method returns anything but zero, we set the message, send it blocking, and clear the FromSerial String.

If there is no data waiting in the Serial object (`Serial.available()` returns 0 or logical false) or if `FromSerial.length()` returns 0 or logical false because the FromSerial String object is empty, we do nothing, and loop repeats forever, until we load a new sketch or turn off the power.

■ **Note** Boolean values are stored in 8 bits of data, even though they only need one. The way they're evaluated is that if every bit is 0, then the Boolean is false. Otherwise it's true. This is why I can throw integers, like FromSerial.length( )'s return value, at boolean logic and have them evaluate correctly. The canonically correct way would be to see if FromSerial.length( )==0, but this works.

Summary

In this chapter, we set up an external Arduino as an ISP and loaded a bootloader onto the Cestino. After that, we tried two fairly simple sketches on it to prove that it works.

Credit Where Credit's Due

Once again, the Arduino LLC did most of the heavy lifting in this project. I added the Burn Bootloader Preflight Check tool. Obviously, Mark Fickett's arduinomorse library saved me a ton of coding.

Further

Like Chapter 2, the further part of this project is really the rest of this book, in which you use the Cestino as a tool to do the remaining the projects. The Morse Code Practice Translator, by contrast, could go much further.

Out there on the Internet, there exist several other Morse code libraries. Unlike arduinomorse, some apparently have the ability to time and translate incoming Morse code. Given that and an extra button (or a proper Morse code key) wired to a separate pin, one could have two-way conversations with the Arduino in Morse code. A web search will turn up a number of resources.

The ability to send or receive and translate Morse code might have huge possibilities in the ham radio world. Although code is not required for some licenses anymore, it is still *used*.

CHAPTER 4

8 Bit Ports

The Arduino LLC set out to accomplish some very specific goals with the Arduino, and one of these was to make programming Arduinos as simple as possible for beginners. Most hardware interactions involve digitalRead() and digitalWrite() or some variation on those two functions. These are useful functions, but they're an abstraction—there's that word again—from how the ATmega really communicates with the outside world.

So far as the ATmega1284P (or any other ATmega used in Arduinos that I'm aware of) knows, there is no such thing as pin 1 (in the Cestino's case.) It knows that pin only as Port B Bit 0. The digitalWrite() function essentially tells the ATmega to turn on specific bit of a port, and subsequent digitalWrite commands to other bits in that port will switch them off and on, one at a time.

There's another way. The ATmega can send whole bytes to ports with one command. This is important. It's much faster and more efficient to control the port that way if you actually need all eight bits at once.

We'll need all eight bits at once. Most of the ancient integrated circuits you'll find in your junk box expect their communication in 8 bit parallel. In this chapter, we'll get to know the ATmega1284's 8 bit ports, and how to use them in sketches. We'll do it with a simple, classic project: the Larson (memorial) Scanner.

The Stuff You Need

This chapter is mostly about theory and code, but there are two miniprojects, both of which need the same parts.

New Parts

8 330Ω resistors. These are current limiting resistors to keep the LEDs from getting fried, and the ATmega1284P from having its outputs overloaded. 1/4 watt resistors are fine, as always.

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_4

New or Used Parts

8 LEDs, preferably the same size, shape, and color or 1 LED bar graph (any orientation) with at least 8 LEDs in it. Common cathode or common anode bargraph LEDs will work, but the wiring will be different. Most common LEDs will work fine with 330Ω resistors at five volts, and won't draw so much current that the ATmega can't source it. If yours are unusually large or bright, you may need to adjust the resistor values.

A Little Binary With That

The Arduino foundation hid the 8 bit ports for one good reason: in order to use them, you really need to have a good understanding of binary. So that we're all on the same page with binary, here's a quick refresher.

Counting in Binary

Computers are a human artifact. We made them to do things we do in ways we understand. Like us, computers store numbers in dual-state systems. They have logic gates that can be either on or off. We have fingers that can be either up or down. Seems pretty obvious, right?

Well, yes and no. If you count on your fingers, zero through nine, carry the one for ten, you're counting in base ten numbers. Like computers, the base 10 number system is a human artifact. It is how it is because we have ten fingers. Multiplying and dividing are easy because you just move the decimal point one space to the left or the right. It's all very convenient with our ten fingered hands.

It's not very efficient, though. Fingers are all used individually, as symbols. There are huge numbers of permutations of fingers that simply aren't used. A thumb and finger is two, but a thumb, finger, and pinkie have no meaning. And so on.

Logic gates are expensive. Okay, they're beyond cheap today, but in 1937 when Dr. George Stibitz (`https://en.wikipedia.org/wiki/George_Stibitz`) was building an adding machine, *his* logic gates (relays) were expensive, and he knew that using the base 2 (binary) number system would use *all* the permutations of his relays. It also meant the thing would fit in his kitchen, where he was building it. By the way, he's also the one who coined the term "digital" to describe electronic computing.

So. A single gate has two possible states, on or off, binary 1 or zero. If a computer wants to count beyond one, it has to add another Binary digIT. Now the computer can count from zero to three, representing a total of four values. Need to go higher? Add another bit. Now we can go from zero to seven, for a total of eight values. Continue adding bits until you can compute the size numbers you have in mind.

Remember how I said that the ATmega1284P is an 8 bit computer? That means it is designed to handle up to eight bits at a time, values from 0 to 255. More on that shortly.

Here's a party trick you can do to set it in your mind. Bet your friends you can count to 31 on one hand, assuming that hand has all five fingers. Your thumb is the rightmost bit. (This is easier with your left hand, for those so equipped. To do it with your right hand, rotate your hand so your thumb is pointed to the right.) Your thumb is the ones bit, the least significant bit. Zero is no fingers. Unless I tell you to raise a finger, keep it curled in the off position. Raise your thumb. That's one. Add one, carry to the left, which would be your index finger, and put your thumb back down. That's two. Three is thumb and index finger raised. Four is an obscene gesture: second finger raised, thumb and index finger lowered. Five is middle finger and thumb raised. Six is either a peace sign or another obscene gesture, depending on where you are: second finger and index finger raised. Seven is middle finger, index finger and thumb. Eight is hard to do. You have to raise your ring finger without raising any others. (Did you know that the middle of your middle finger and your ring and little fingers are connected to a different nerve from the thumb, index finger, and the other half of your middle finger? Now you do.)

Eight through fifteen are the same as zero through seven plus the eight finger. Your pinky is 16, the most significant bit. Seventeen is hang loose or call me maybe. Thirty-one is all five fingers raised. Go ahead and collect your bets.

If you've the patience, coordination, and flexibility to use both hands and all ten fingers, you can count to 1,023, but it hurts a lot, and your friends will probably wander off long before you're finished. If it were possible to use your toes this way, for a total of twenty bits, you could count to 1,048,576. IPv6, the new emerging standard for Internet addresses, has 128 bits of address space. Theoretically, this is enough to address a bit less than half of the atoms in the observable universe.

Bytes and Words

Okay. That's bits, and how to use them. Let's talk about bytes. I've thrown the word around, and you can't get away from it in modern computing. Most people know that there are eight bits in a byte.

Figure 4-1. *Byte Diagram*

It wasn't always so.

Back in the stone age of computing, 1956, to be exact, the term *byte* was coined because a particular IBM computer could use arbitrary numbers of bits to represent a given number, and the number of bits was called the byte length. It was a play on bite, but one that wouldn't easily be typoed into /bit/ by manual writers. Later, minicomputers shipped with eight bit bytes, and twelve bit bytes, and sixteen bit bytes.

AT&T standardized on eight bit bytes for data transmission, and the standard was cast in stone with the rise of eight bit microcomputers. Today a byte is eight bits, 0 through 7, as in Figure 4-1. No more, no less.

I know, I know. What about 64 bit computers? What do you call a number like that?

If there are more than eight bits in a single ordered set of bits, it's called a word, in modern parlance. Although this may seem like historical trivia, we *are* dealing with some very, very old components, and in old datasheets you sometimes see references to 16 bit bytes, or 8 bit words. Now you know how all these go together. Oh, and half a byte? Four bits? Is a *nybble*.

Bytes are usually represented with the least significant bit on the right, and the most significant bit on the left, just like base 10 mathematics. Ports on a given IC are under no obligation to follow this tradition. Remember that port A on the ATmega1284P has the least significant bit (the ones bit) on pin 40 and the most significant bit (the 128 bit) on pin 33. When we get to the Z80, you'll see that some bits for a given port aren't even on adjacent pins.

Bit Twiddling

Understanding binary and binary math gives us the same kind of control over each pin of the ATmega1284P as digitalRead() and digitalWrite() do. Even though we throw whole bytes at a port, we can twiddle the individual bits until the pins are how we want them.

A bit's value is either zero or one, so it's possible to do bitwise logic (boolean logic) on binary numbers. This means along with adding, subtracting, multiplying, and dividing, you can AND two numbers together, OR them together, or XOR (exclusive-or) them together. You can NOT bits. Finally, you can shift bits left and right in the word.

Wait, what?

AND

Let's start with AND.

If you've done any Arduino programming (or any programming at all) you're already familiar with AND. If condition a and condition b are true, then the AND is true. Otherwise it's false. Bitwise ANDing is based on the same general concept.

■ **Note** Don't confuse the logical operators: &&, || and ! (AND, OR, and NOT) with bitwise operators &, |, ^, and ~. (Bitwise AND, OR, XOR, and NOT, respectively.) It's easy to do because in logical terms, they do the same thing. The key difference is that bitwise operators act on binary data types, whereas logical operators evaluate conditions that may involve multiple variables. We're dealing exclusively with bitwise operators here.

00000001 & 00000011 = 00000001 If we AND 1 and 3, I get 1. Only the digits where both bits in both operands are set to one are evaluated as one. Any others are evaluated as zero. The & is the bitwise AND operator, just as + is an addition operator and * is a multiplication operator.

If you want to see the Cestino actually do this, the code looks like this:

```
void setup() {
  //All code is in setup because we want to run it only once.
  Serial.begin (115200); //Set up serial com at 115200 Baud.
  byte mybyte; //Declare the variable mybyte of the type byte.

  mybyte = 0x00000001 & 0x00000011;
  //Assign mybyte to the bitwise AND of 0x00000001 and B00000011, which are
  //1 and 3, respectively.
  Serial.print(mybyte, BIN);
  //Serial.print the result, formatted as binary. Note that leading 0s
  //will not be displayed.
  Serial.println(); //Print a linefeed.
}
void loop() {
  if (1 == 1) {}; //Do nothing, over and over, really fast.
}
```

The Cestino will return 1. Note that this is binary 1, equivalent to 0x00000001, but all the leading zeros are chopped off.

OR

OR is the opposite of AND. OR returns a 1 for any bit that is set to 1 in either byte. If the same bit is 1 in both bytes, OR will return a 1 for that bit as well. The OR operator is |, or the pipe character (not an exclamation point.) Unix and DOS users will be well acquainted with this character. Others may not be. On my U.S. Mac keyboard, it's a shifted backslash, all the way on the right above the return key, but there's no hard and fast standard where it will be on any given keyboard. Here's an example.

```
void setup() {  //All code is in setup so it runs only once.
  Serial.begin (115200); //Set up serial com at 115200 Baud.
  byte mybyte; //Declare the variable mybyte of the type byte.

  mybyte = 1; //Set mybyte to 1. Note that 1 is equal to 0b00000001.
  mybyte = mybyte | 0b00000011;
  //Set mybyte to the OR of itself and 3.
  Serial.print(mybyte, BIN);
  //Serial print mybyte in binary format.
  Serial.println(); //Print a linefeed.
}
void loop() {
  if (1 == 1) {}; //Do nothing, over and over, really fast.
}
```

The Cestino will return 11 (3 in binary.)

XOR

XOR (the ^ operator) does the same job as OR except that if /both/ bits are 1 at a given position, the return value gets a zero. Only bits that are different between the two operands get a 1 in the result.

```
void setup() {
  //All this code is in setup so it runs only once.
  Serial.begin (115200); //Set up serial com at 115200 Baud.
  byte mybyte; //Declare the variable mybyte of the type byte.

  mybyte = 0b00000001; //Set mybyte to 1.
  mybyte = mybyte ^ 0b00000011;
  //Set mybyte to the XOR of itself and 3.
  Serial.print(mybyte, BIN);
  //Serial print mybyte in binary format.
  Serial.println(); //Print a linefeed.
}
void loop() {
  if (1 == 1) {}; //Do nothing, over and over, really fast.
}
```

The Cestino will return 10. Only bit 2 will be set.

NOT

NOT is different. It takes only one operand, and inverts all the bits. Any bit that is zero is returned as one, and any bit that is one is returned as zero. So 0b00000001 becomes 0b11111110 when NOT is applied to it.

The NOT operator is a tilde. This: ~. On my Mac keyboard, it's on a key all the way at the top left above the tab key with a lot of other obscure, seldom used punctuation. It's the shifted version of that. Here's some demo code:

```
void setup() { //All code is in setup so it runs only once.
  Serial.begin (115200); //Set up serial com at 115200 Baud.
  byte mybyte; //Declare the variable mybyte of the type byte.

  mybyte = 0b00000001; //Set mybyte to 1.
  mybyte = ~mybyte; //Set mybyte to the NOT of itself.
  Serial.print(mybyte, BIN);
  //Serial print mybyte in binary format.
  Serial.println(); //Print a linefeed.
}

void loop() {
  if (1 == 1) {}; //Do nothing, over and over, really fast.
}
```

The Cestino will return 11111110, the inverse or negation of 00000001. This is 254 in decimal. Each bit of the original value 00000001 has been reversed.

Bit Shift Operators

The weirdest operators, conceptually, are the bitshift left and right. This is done with the less-than character, twice. Like this: <<.

A byte, you will recall, has eight bits, or binary digits. In Arduino code, we'd picture it like this: 0b00000001.

If we reassign it with mybyte=mybyte << 2; we shift the entire byte two steps to the left, which gives us a value of 0b00000100, which is 4. Bit shifting to the left is effectively multiplying by 2. Let's look at some demo code.

```
void setup() { //All code is in setup so it runs only once.
  Serial.begin (115200); //Set up serial com at 115200 Baud.
  byte mybyte; //Declare the variable mybyte of the type byte.

  mybyte = 0b00000001; //Set mybyte to 1.
  mybyte = mybyte << 1; //Bitshift mybyte one step to the left.
  Serial.print(mybyte, BIN);
  //Serial print mybyte in binary format.
  Serial.println(); //Print a linefeed.
```

The result of this sketch fragment (note that it's missing the entire loop() statement and is continued below) will be that the Arduino prints 10, which is two in decimal. Why? Because 00000001 (decimal one) shifted left one digit is 00000010 (decimal two.)

If we shift six more steps to the left, we set the most significant bit in our single byte to 1. Looking at the byte once again, 00000010 shifted six more steps to the left gives us 10000000. Here's more code. It would go in right under the code we've just looked at.

```
  mybyte = mybyte << 6;
  //Bitshift mybyte six more steps to the left.
  Serial.print(mybyte, BIN);
  //Serial print mybyte in binary format. We should get 10000000,
  //equal to decimal 128.
  Serial.println(); //Print a linefeed.
```

Right shifts work the same way, in the other direction. If you guessed that the greater-than sign twice is how you get a right-shift, you guessed right. It looks like this: >>, and is a shifted period on my Mac keyboard. Right shifting is the equivalent of dividing by two. If we take our existing byte, 10000000 (decimal 128) and shift it a step to the right, we get

01000000 (decimal 64.) Here's some more code. Again, it's not a complete sketch without the code above and below.

```
  mybyte = mybyte >> 1;
  //Bitshift mybyte one step to the right.
  Serial.print(mybyte, BIN);
  //Serial print mybyte in binary format. We should get 1000000,
  equal to decimal 64.
  Serial.println(); //Print a linefeed.
```

You might wonder what happens to the bits that get shifted off the edge of the byte in either direction. The answer is they cease to exist. They aren't stored anywhere. Shifting the other direction will not retrieve them. Once they're gone, they're gone. But let's see. These lines do exactly that.

```
  mybyte = mybyte >> 7;
  //Bitshift mybyte 7 more steps to the right, off the edge of the byte.
  Serial.print(mybyte, BIN);
  //Serial print mybyte in binary format. We should get 0. We shifted the
  //set bit all the way out of the byte.
  Serial.println(); //Print a linefeed.

  mybyte=mybyte <<1;
  //Shift mbyte to the left once to see if that bit is really gone.
  Serial.print(mybyte,BIN);
  //Serial print the result. We should still get 0.
  Serial.println(); //Print a linefeed.
}
```

Yup. Shifting seven digits to the right gives us a zero. The 1 goes off the edge of the byte, never to return. Shifting it left again shows that. The result is still zero.

And, because this is the end of the demo sketch, we'll put the usual boilerplate at the end.

```
void loop() {
  if (1 == 1) {}; //Do nothing, over and over, really fast.
}
```

Gotchas

There are some gotchas for bit twiddling in C++, which is what we're doing. Some of them are unique to the Arduino environment, some are just functions of binary math and how large numbers are stored.

Multibyte Data Types

Most Arduino data types aren't single bytes. The vast majority are multibyte words. Shifting them may not give you the value you expect, particularly if you assume as we did in the sample code earlier, that shifting a bit 8 steps to the left shifts it out of the byte altogether. On a 16 bit unsigned int, shifting 1 eight steps to the left gives you 256, or 0000000100000000.

```
void setup() { //All code is in setup so it runs only once.
  Serial.begin (115200); //Set up serial com at 115200 Baud.
  unsigned int myint;
  //Declare an unsigned integer - two bytes.

  myint = 1; //set it to 1.
  Serial.print(myint << 8, DEC);
  //Print it shifted 8 digits to the left.
  Serial.println(); //Print a linefeed.
}
void loop() {
  if (1 == 1) {}; //Do nothing, over and over, really fast.
}
```

If you run this, you'll get decimal 256. In a single byte, this would be a value you could never see. It takes 9 bits to represent it. Because Arduino ints are two bytes long, there it is.

Signed Data Types

Many variable types in Arduino and C++ are *signed*. This means that the most significant bit of the variable is used to determine whether the value is positive or negative. These are trickier than they look. An int, in Arduino, is a 16 bit signed variable. So `int c = 1;` stores a variable in memory that looks like this: 0000000000000001

So far so good. 255 is encoded as 0000000011111111, 256 is encoded 0000000100000000, and so on right up until you get to 32767, which looks like this: 0111111111111111. Add one to that, and a strange thing happens. The last bit gets set. Because this is a signed integer, the default type of integer, the last bit has a special meaning. If it is set, it means that the number is negative. But if you add 1 to 32767 in a two-byte signed integer, it will set that bit anyway. A number suddenly going negative means you've overflowed the datatype.

So 1111111111111111 is -32768. Add one to that, and you get 1111111111111110, or -32767. This is a function of twos-compliment math, which is a lengthy topic worth reading up on. Fortunately we don't need to bit twiddle the data we get from a device, so it doesn't come up often.

Demo code? Sure.

```
void setup() {
  //All code is in setup so it runs only once.
  Serial.begin (115200); //Set up serial com at 115200 Baud.
  int mysignedint; //declare a signed integer.

  mysignedint = 32767; //set to max value for a signed int.
  mysignedint = mysignedint + 1; //add one.
  Serial.print(mysignedint, DEC); //It's now negative.
  Serial.println();
  mysignedint = mysignedint + 1; //add one more.
  Serial.print(mysignedint, DEC); //it's now a lower negative.
  Serial.println();
}
void loop() {
  if (1 == 1) {}; //Do nothing, over and over, really fast.
}
```

Endian-ness

There is one more gotcha, and unlike the others, it can bite us even if we haven't touched the bytes coming out of a device.

If we have a 16 bit word, especially if we're passing it through an 8 bit computer, which byte is the higher one? If you picture the bytes all in a line, it makes sense that the least significant bit is the furthest right, and the most significant bit is the furthest left, but when we move them around in the ATmega1284P, they're stacked on top of each other in 8 bit columns. How do you know which one is bits 16 through 8 and which one is bits 7 through 0? Welcome to the concept of endian-ness. The word itself comes from Jonathan Swift in *Gulliver's Travels,* where there is a hundred years' war over between the little endians—people who break their soft-boiled eggs open on the small end—and the big endians—people who break them open on the big end. So it is in computer science.

It comes down to this: when two or more bytes of a given number (word) are stored in 8 bit memory at sequential addresses, if the lowest address is the most significant byte, your machine is big-endian. If the lowest address is the least significant byte, your machine is little endian. The Intel x86 architecture has always been little endian, and so are both the ATmega1284P and the Arduino GCC compiler. Trivia? You might think so, but I guarantee you it will come up again. There's no easy way to demonstrate this without more hardware, but I'll point it out when it comes up in the projects to come.

Serial.print() Weirdness

If you find yourself messing with negative integers and other signed, multibyte data types, be aware that Serial.print(int,BIN) (or many other formats) will give you results that are simply wrong. When you call Serial.print, it type casts your variable (changes the type of your variable) into a 4 byte long variable type. For positive values, this is transparent, but negative values coming from signed variables shorter than 4 bytes the sign bit gets absorbed where it shouldn't be, and the value is corrupted.

In order to get the values you expect, you have to cast the variable you're putting into Serial.print() into the unsigned version of the data type. Fortunately, casts in C and C++ are easy. Just put the data type you want to cast your existing variable into in parenthesis in front of the variable.

For an int, you'd do it like this: `Serial.print((unsigned int) mysignedint,BIN);`

Binary Notation in Arduino

The Arduino core lets you use nonstandard notation for binary values, like this: B00000001. It's very handy. Unfortunately, it's limited to single byte values, and will produce values that are simply wrong (without warning) if you try to use it for longer values.

The somewhat more standard notation is 0b00000001. It's a GCC extension, and the Arduino app uses GCC for its compiler. Rumor has it that this notation will be part of the next C++ standard, real soon now. So that touching multibyte values with binary values doesn't bite us really hard, I'll use the GCC extension to C++ notation: 0b00000001.

The Cestino's Ports and How to Use Them

The ATmega1284P at the heart of the Cestino has four 8 bit ports: A, B, C, and D. See the modified pinout diagram in Figure 4-2.

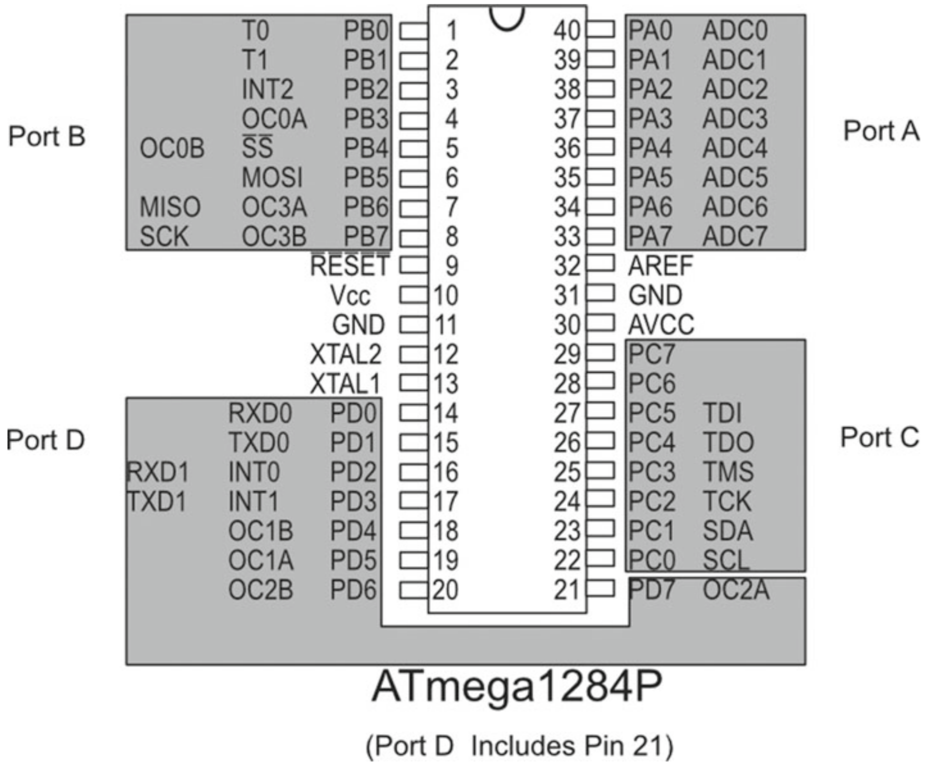

Figure 4-2. *Cestino Ports*

Don’t Step on the Ports

Each port on the ATmega1284P, in addition to providing General Purpose IO (GPIO), has special functions, which you can see on Figure 4-2.

Port A is where the analog pins live.

■ **Note** Port A is *backwards* from how you’d expect. Its bit 0 is on pin 40 of the IC and its bit 7 is on pin 33 of the IC. The analog pins are numbered the same direction, with Analog 1 on pin 40 and Analog 8 on pin 33. On the Cestino, only digital pins as used in digitalRead() and digitalWrite() follow the package pin numbers.

Port B is where the SPI connections live. If you’re using SPI in a project, these pins have to be protected when they’re going to be used.

Port C has the I2C connections. We're not using them, so we can use this port with impunity.

Port D has two different USARTs, one of which we're using for RS232 communication. This matters if we're trying to use a port that has another job. It can be done, but we have to watch out for the bits (pins) that are in use.

Port Registers and Commands

There are three special registers in the ATmega1284P for each port.

That's great. What's a register?

A register is, from the sketch's point of view, just another variable, although you don't have to declare it. The fundamental difference is that it's a variable that the ATmega1284P will automatically look at or deposit things in. Registers really go deeper than this. They're not handled with the same mechanisms as normal memory. They're little windows into the microcontroller, wherein you can change what is being done, or read what is being done. Registers can be read-only, write-only, or both, and they are fundamental to the architecture of the microcontroller or microprocessor.

The port registers are exposed by the Arduino core as DDRx, PORTx and PINx, where x is one of the Cestino's ports.

■ **Note** Your Arduino has ports also, but they may be different and have different names from the Cestino's. A common port environment is one of the big reasons to build the Cestino for the projects in this book.

We'll use port C (DDRC, PORTC, and PINC) for all of our discussions here, but DDRA, PORTA, PINA DDRB, PORTB, PINB, DDRD, PORTD, and PIND do the same things for the other ports. The ports are controlled, read, and written to by setting and reading these registers. You know how we spent all those pages in this chapter on binary math? This is why.

DDRx Register

The DDRx register—DDRC, for example—is the Data Direction Register. It does exactly the same thing as pinMode(pin number). It determines whether the pins in the port are inputs or outputs. You set these to a binary value, like this:

```
DDRC=0b11100111;
```

This command would set the first three bits and the last three bits of port C as outputs, leaving the middle two pins as inputs. You could do the same thing this way:

```
DDRC=231;
```

PINx Register

The PINx register (PINC, for our examples) is the input register for a given port, that returns what values are being read by the inputs. PINx is read-only. When you read the PINx register, any pins on the physical port which are held high will result in that bit of the register being a 1. Pins held low will return bits set to 0.

If, as I did when I was testing the Cestino firmware, you have a switch set up to connect the pins of Port C to the + bus or the - bus depending on position, here's what you'd do to read it.

First, set up the port.

```
DDRC=0b00000000; //Set all pins of port C to read mode.
```

Then read it and send the output to the serial console.

```
  Serial.print(PINC,BIN);
//print the value of PINC in binary format to the console.
```

Again, we treat the PINC register just like an 8 bit variable type in C++, because that's how it's exposed by the Arduino core. The difference is that we don't declare it, and that we can't change the value. It's set by the ATmega1284P and the Arduino core software.

PORTx Register

The PORTx register is the output register for a given port. To use it, set it equal to a given 8 bit value, such as a byte variable type, a decimal number less than 256, or a binary of eight bits, no more, no less. When the PORTx register is set, the corresponding bits on the physical port will go high for 1s and low for 0s, and they'll do it all at once, rather than one at a time as they would using digitalWrite().

Like this:

```
DDRC=B11111111; //Set all pins of port C to output (write) mode.
PORTC=B10101010; //Turn on alternating pins on port C.
```

Pretty straightforward, right? Well, yes, as long as you want to control all the pins of that port at once. With ports A, B, and C on the Cestino, we can do that. Port D is another matter.

In our case, we're using pins 0 and 1 of port D for RS232 communications between the Cestino and the Arduino application. If we don't care whether the Cestino can communicate while our sketch is running, we can ignore the RS232 pins and use them as we please, with the caveat that the RS232 electronics are still connected to them and will affect their values electronically. If we do want the Cestino to be able to communicate with the Arduino application while the sketch is running, we need to protect those two pins from getting clobbered. This is where all that binary we talked about comes in.

Let's assume we want to use all the pins except 0 and 1 on port D. We have to set it up correctly first.

■ **Note** Know your port pin order! Binary is always numbered right to left, both on paper and in binary formatted numbers in Arduino sketches. Ports A, B and D, by contrast, have their lowest significant bit on the left with pins 1 and 20 facing you. In this orientation, only port C will line up with the binary notation and bit shift directions. This fact can produce confusing results if you (or your author) lose sight of it.

```
DDRD=DDRD | 0b11111100;
//Change all the pins except 0 and 1 to output mode.
```

This sets all the pins except 0 and 1 of port D as outputs. If you OR 0 with the input/output value of pins 0 and 1, you'll recall from our discussion of binary, it leaves those pins with whatever value they had before, so RS232 should go on working.

Likewise, when we set PORTC to some value, we need to OR that value with the current output settings of port C.

```
PORTD=PORTD | B11111100;
//Set all the pins high except 0 and 1.
Serial.println("Zortwootle");
//Prove that console communications still work.
```

Remember: OR returns a 0 only if neither byte being evaluated has a 1 in that position, and a 1 for a given position otherwise, so ORing PORTD with a byte that has bits 0 and 1 set to 0 will return whatever value was already in PORTD.

■ **Note** It might look like you could read the input state of the port with the PORTx register. You can't. That's what PINx is for. PORTx will only tell you what output pins have been turned on.

Pullup and Pulldown Resistors

There's one more wrinkle with the PORTx register.

You may recall from Chapter 2 where we put a pull-up resistor on the reset circuit of the Cestino to ensure that, unless the reset switch is pressed or a reset pulse comes in from the RS232 port, the Cestino does NOT reset.

Pull-up resistors are very, very common in digital electronics. So common, in fact, that the ATmega1284P has them internally on all the ports. By default, they're turned off, so we haven't dealt with them until now.

Pull-up resistors are only useful on input pins, so the way they are controlled is to write to pins in a port that are set as inputs. Here's some sample code:

```
DDRC=0b00000000; //Set all pins of port C to read mode.
PORTC=0b11111111; //Turn on all pull-up resistors in port C.
PORTC=0b00000000; //Turn off all pull-up resistors in port C.
```

Just as we have to be careful not to change the input/output settings of pins that are doing other jobs for us (like pin 0 and 1 of port D) it's important not to write to these pins at all, lest we change the pull-up resistor states for those pins.

For reference, the ATmega1284P's internal pull-up resistors are between 20kΩ and 50kΩ (20,000 and 50,000 ohms.) You don't have to conduct much current at all to pull an input pin low with its pull-up resistors set. There are no corresponding internal pull-down resistors. If you need pull-downs in your circuit, you'll have to add them yourself.

Build the Larson (memorial) Scanner

The Larson (memorial) scanner is a project that comes straight from television. In the late 1970s and early 1980s (my youth) TV executive Glen A. Larson produced two science fiction TV series: *Battlestar Galactica* and *Knight Rider*. Although they were wildly different, both of them featured sentient robots with a single red light for an eye that scanned back and forth. (In the day, we used to joke that this was why Cylons couldn't hit anything when shooting—no depth perception and their eye kept moving back and forth.) The Larson Scanner, which I'm calling the Larson (memorial) scanner, is homage to those shows, and to the man himself, now that he's gone. It's also a classic Arduino project. We're going to bend it here, slightly, to get some familiarity with port manipulation.

The Circuit

The Larson (memorial) Scanner is a fairly simple animal.

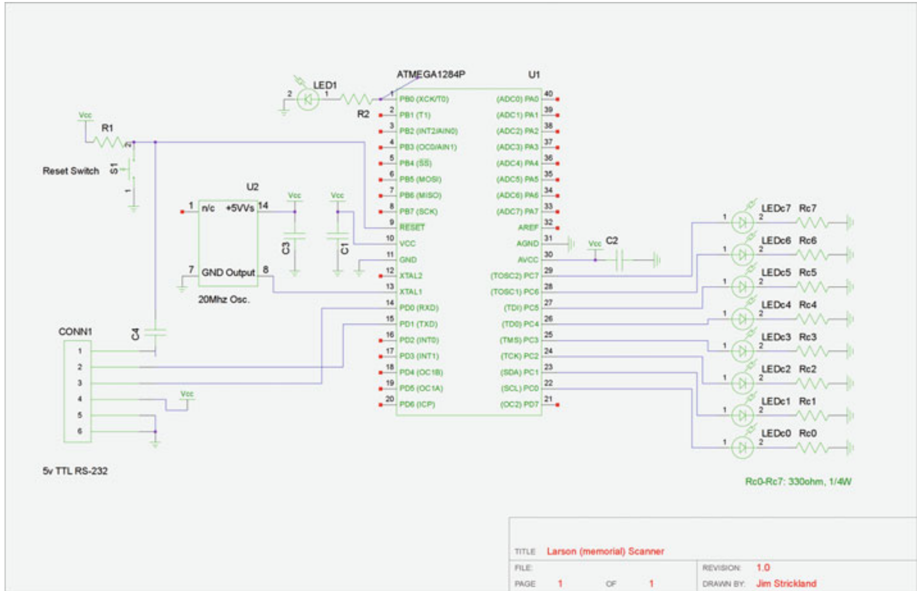

Figure 4-3. *The Larson (memorial) Scanner Schematic*

If the schematic in Figure 4-3 looks suspiciously like I've added the pin 1 LED circuit over and over again on port C with a couple variations, you've got a good eye. It's exactly that. Although I'm using an LED bar-graph display, internally it is ten separate LEDs, each with its own anode and cathode, of which I'm wiring up eight. Each LED gets its own dropping resistor between the ground (-) bus and the cathode, and each anode connects to a pin on port C. You know how to calculate dropping resistors and current already. Do make sure that the Cestino can drive all 8 LEDs at once without overloading.

All wired up? Does it look something like Figure 4-4? Good.

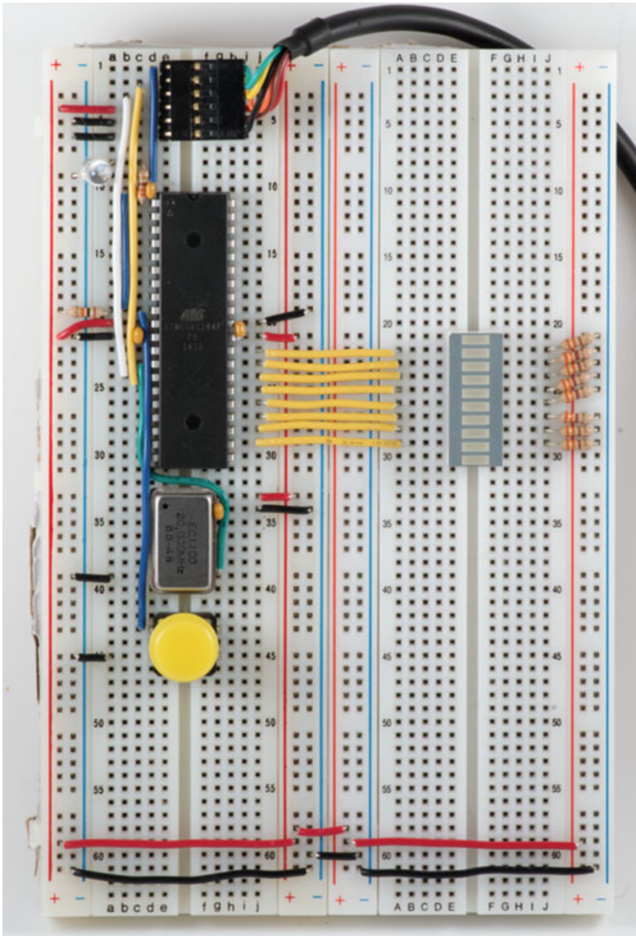

Figure 4-4. *Larson (memorial) Scanner*

If you have a bar graph larger than 10 segments, and it just doesn't seem to have enough pins, it's probably set up for multiplexing. Instead of using one data line for each segment, as we have in Figure 1–3, they have (usually) four data lines and half a dozen or more common cathodes or anodes (usually cathodes). You switch a cathode to ground, display your four pins, then raise that cathode and switch the next one to ground, display those four pins, and so on. If the driving electronics (Your Cestino and a sketch, perhaps) hustle, all the segments can appear lit at the same time, even though only a maximum of four really are. This trades speed (which the LEDs and our eyes can't use anyway) for current and data lines (port connections), and we'll do some of it when we get to chapter 10. For now, I recommend putting that bar graph back in the junk box and using a smaller one, or separate LEDs.

The Code

Here's where we get to use that port manipulation and binary math stuff we've been talking about. The traditional way to do the Larson Scanner with Arduino is to count from 1 to 8, use digitalWrite() to set a pin on, and another digitalWrite to turn the previous pin off, then change direction and go the other way. We can do it a whole lot easier.

```
//Larson Memorial Scanner (Port C)
// --------------------------------------------------------
//This sketch turns on a sequence of LEDs, one at a time,
//from the least significant bit to the most significant bit,
//then back again using bit shifting.
// ---------------------------------------------------------
//Hardware:
//LEDs wired with their anodes to port C's pins, and dropping
//resistors connecting their cathodes to the ground (-) bus.
// ----------------------------------------------------------
// James R. Strickland
// ----------------------------------------------------------

//precompiler definitions.
#define DELAYTIME 50 //How many mS (milliseconds) between LEDs?

//declaration of variables.
bool reverse; //Declare our reverse-direction flag

//setup() function - runs only once.
void setup() {
  DDRC = 0b11111111; //Set DDRC to all outputs.
  PORTC = 0b00000001; //Set PORTC to 1. This will turn on pin 22.
  //PORTC can be initalized to any bit you want.

  reverse=(PORTC == 1); //Sanity checking for reverse.
  //If PORTC==1, reverse is set true. Otherwise the
  //boundary logic below barfs.
}

//loop() function - runs forever.
void loop() {
  //Here's that boundary logic. Whatever reverse is, flip it
  //if PORTC is 128 or 1.
  if  ((PORTC == 128) || (PORTC == 1)) reverse = !reverse;

  delay(DELAYTIME); //The Cestino can change the port value
//thousands of times a second. Our eyes and the LEDs, not so
//much. 50mS seems to give a nice, quick scan back and forth.
```

```
//Here's where we actually set the PORTC values.
  if (reverse) { //shift right
    PORTC = PORTC >> 1;
  }
  else { //shift left
    PORTC = PORTC << 1;
  }
}
```

How It Works

This sketch works by shifting a single "on" or true (1) bit left or right. When the value in PORTC reaches either 1 or 128, the least or most significant bit in the port, we change the shifting direction until we hit one of those two boundaries again. Repeat forever.

The direction change logic is the most complicated. In a single if statement, I check to see if bits 128 or 1 are set, and if so, I invert the value of reverse, whatever it is. If it starts out true, we were shifting right, and we've just hit 1, so we need to shift left. If it's false, we were already shifting left and we need to shift right.

The remaining pair of if statements carry the shift out, depending on whether the reverse flag is set, and there's a delay to make it smooth.

So that's the Larson (memorial) Scanner, done with bit shifting. You could do it with loops and setting the port value directly, but if we're going to throw values at the port anyway, let's do something interesting.

Binary Numbers on Display

Have you ever seen a binary clock? I kid you not, they're a clock that has bit fields for hours, minutes, and seconds, and the bits rotate once a second. (Thankfully they don't usually display Unix epoch time, a 4 byte, unsigned integer of seconds since midnight, January 1, 1970.) We don't have enough bits in the Larson (memorial) scanner to display the full time, let alone epoch time, but we can count seconds for a while.

There aren't any changes to the hardware. We have a port wired to LEDs, and the Cestino can drive them all simultaneously. (You did calculate your dropping resistors so it could, right?) That's all we need. All that changes is the sketch.

```
//Binary Numbers On Display
//-----------------------------------------------------------
//This sketch counts seconds in binary from 0 to 255, then
//resets. It's a really, really short sketch.
// ----------------------------------------------------------
//Hardware:
//LEDs wired with their anodes to port C's pins, and dropping
//resistors connecting their cathodes to the ground (-) bus.
// -----------------------------------------------------------
// James R. Strickland
// -----------------------------------------------------------
```

```
//precompiler definitions.
#define DELAYTIME 10 //How many mS (milliseconds) per update?

//setup() function - runs only once.
void setup() {
  DDRC = 0b11111111; //Set DDRC to all outputs.
  PORTC=0;
}

//loop() function - runs forever.
void loop() {
  PORTC++; //Take whatever is in PORTC and add 1.
  delay(DELAYTIME);  //Wait DELAYTIME milliseconds.
}
```

How it works: As before, setup() configures DDRC as all outputs. Then it sets PORTC to 0. The loop() function increments whatever value is in PORTC by 1, waits DELAYTIME milliseconds (You can set it shorter than 1000 if you're not patient), and repeats forever.

Wait. Where's the reset logic? How does the Cestino know when to zero the register and start counting over again?

PORTC is a single byte value. When the Cestino tries to increment PORTC from 255 to 256, it clears all the rest of the bits and sets the 256 bit. There is no 256 bit in the 8 bit PORTC register, so the register gets set to 0.

Some times, when you know your binary, it really is that simple.

Further

There exist bar graph LEDs with 50 or more LEDs, all multiplexed together so no more than four of them are on at any given time. Connecting these to the Cestino seems easy. You connect the anodes to one port, and the common cathodes to one or more ports, do a little port manipulation, make sure the delays are short, and you're good. This would work because when the ATmega1284Ps port pins are *zero* or low, they are effectively connected to the ground (-) bus.

There's a catch. Those connections still go through the integrated circuit, and the limits for how much current the ATmega1284P can sink (that's what we're talking about) are almost as small as the amounts of current it can source (what we've been doing in this chapter.) Look in the datasheet and do the calculations to make sure you don't overload the Cestino, or it may reward you by spontaneously restarting, or other maladaptive behavior. If you want to drive large bar graph displays like this, the best answer is to get a Darlington transistor array, like the ULN2803. They're cheap, they work very well with ATmegas, and they can switch 500 milliamps per channel. We'll use one later.

A Cestino port combined with a ULN2803 or similar Darlington array can be used to control robotics, sprinkler systems, or whatever you can imagine. Eight bits at a time.

CHAPTER 5

Collector, Base, and Emitter

Transistors are as ubiquitous as dirt. In the CPU of the computer I'm typing this on, there are 1.4 *billion* of them, mostly acting as switches, plus over 128 billion more in the RAM. The number of these devices in the world is truly mind-boggling. In this chapter, we're going to deal with discrete transistors: the kind packaged one at a time with three leads out the bottom. This isn't just an academic exercise. If you need to shift more current or voltage than the pins on your Cestino can handle, you'll probably reach for a transistor, particularly a Darlington transistor. (More on these in a bit.) Whereas the output pins on the Cestino can source or sink no more than 40mA at five volts, a TIP120 Darlington with a proper heat sink can sink 5 amps (5000mA) at up to 60 volts, and can be easily driven by one of the Cestino's output pins.

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_5

If you're going to go much beyond this book, you'll need to have a working understanding of transistors and how to use them. Hopefully when we're done here, you'll have that. If you're going to scavenge them out of old equipment, you'll often find that the manufacturers have stripped the identifying marks from all the transistors (boo), and that's where this project comes in. When we're done here, you'll have the analyzer shown in Figure 5-1. You'll be able to place an unknown transistor in it and get the transistor's type (NPN or PNP) and which pin its base is on. These facts go a long way toward making a scrap transistor useful.

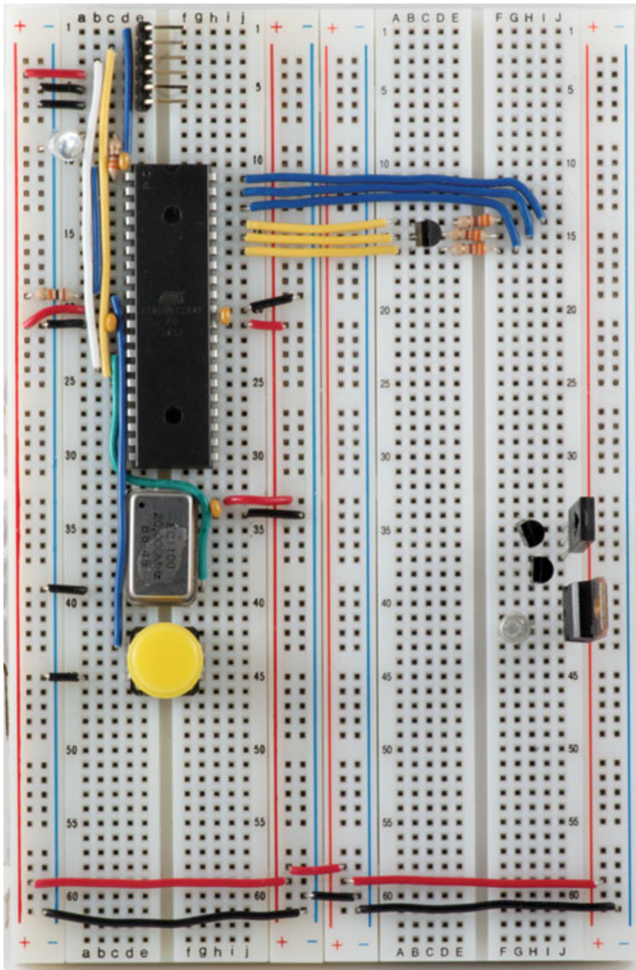

Figure 5-1. *Transistor Analyser and Various Transistors*

This chapter goes about as deep into the underlying technology as we're going to go, so if you start to feel your eyes glazing over at all the theory (and some math), hang in there. We'll go back to dealing with hundreds or thousands of transistors in abstract in the next chapter.

The Stuff You Need

There are a few things you'll need.

New Parts

3 330Ω resistors. You can reuse the ones from Chapter 4 if you like.

New or Used Parts

A few bipolar junction transistors (BJTs) or power transistors. I tested with an assortment from Microcenter and used BC547s, 557s, BD140s, and a couple TIP120 Darlington transistors I had left over from an old robotics project. I also tested an unknown I had, part of a grab bag, and discovered it to be an NPN transistor with its base on its center pin.

Software

A datasheet on your transistor. 2n2222As can be found here: `http://www.onsemi.com/pub_link/Collateral/2N2222A-D.PDF` (This is much nicer than the older Fairchild semiconductor datasheet, which lacks the gain charts.)

Ultimately the device we'll build can begin the process of identifying bare transistors, but until we have it working, it's best to use transistors you know.

A Little Semiconductor Theory

Transistors are semiconductors. To understand how they work, we have to dig a little deeper into the underlying physics. Books have been written on the subject. Since this isn't one of those books, I'm hand-waiving an awful lot of of the subtleties of physics and quantum theory in order to keep this section to a reasonable size. If you're interested, it's a big, big field, and there's lots written about it. I recommend this article: `https://en.wikipedia.org/wiki/Electrical_resistivity_and_conductivity`.

We start with atoms, but we'll be scratching the complicated surface of quantum theory and band theory before we're done.

Electron Orbitals, Bands, and a Dab of Quantum Theory

You may have heard, somewhere along the line, that atoms, far from being indivisible as Dalton claimed, are made up of protons, neutrons, and electrons, and that protons and neutrons form the nucleus, and electrons orbit it much as the planets of our solar system orbit the sun, and all the stuff in our galaxy orbits the black hole at the center. That's more or less Bohr's model, and it's at the heart of most of the explanations of transistor physics I've seen. Is it right?

Not quite. Thanks to Heisenberg's uncertainty principle (and the physical realities it describes) we know it is impossible to know both the position and momentum of an electron at the same time. Measuring one distorts the other. He proved it mathematically, too. Instead, modern quantum theory, of which Heisenberg's theory is part, describes electrons in terms of probability waves. This probability wave is why, among other things, electrons behave as though they are waves, even though they are particles with mass.

So what?

Electrons go faster when their energy is higher. Higher energy electrons will be further from the nucleus. If you squint your eyes, it almost looks like classic Newtonian mechanics, even though it's not. (As it happens, if you apply Shrödinger's equation to planitary orbits, it works just fine, but the other possible orbits are fractions of a milimeter apart. On the scale of planets, that's continuous.)

An orbital's probability function is an expression of the three quantum numbers that define it. They have a variety of shapes, and these shapes, combined with electron spin, define the number of electrons that can reside in each one. There are combinations of quantum numbers that are not allowed, and Pauli's exclusion principle prevents any two electrons from occupying the same quantum state.

There are combinations of quantum numbers which cannot hold electrons, for a variety of reasons, and it is this that defines the boundaries of the various orbitals' probability. These boundaries are critically important in semiconductors.

If an orbital at lower energy it isn't full, an electron will preferentially give up the extra energy (slow down) and drop down into the orbital below, attracted by the protons in the atom's nucleus. If there's no room, the electron will stay in the orbital it's in.

If you've had any chemistry, this probably sounds familiar. The outer orbital of electrons in an atom define its properties and the chemical bonds it's capable of making, but what we're doing doesn't involve those electrons in quite that way, so we'll sidestep chemistry and go to band theory instead.

Band theory, essentially, is what happens when you have not one atom, but billions. The probabilities mingle to a range of probable values. Billions upon billions of throws of the dice of probability and you wind up with a band of energy levels that correspond loosely to the shells of a given atom.

Electrons move within this band with very little coaxing from outside forces. Once the outer orbitals are full, resulting in full energy bands, the electrons are bound into place and don't move without much higher applications of energy. Metals share lots of electrons, as their outer electron shells are nearly empty. This makes it easy for electrons to drift from one atom to another and transfer charges to each other.

Does that sound familiar? It should.

Copper, Revisited

You may recall from Chapter 2, a Dab of Ohm's Law, I said, "Metals like copper are made of a lattice of atoms that have a particularly useful property: each one has an electron that can be easily pulled out of its normal orbit and move along the lattice." I went on to say that that copper is a good conductor because it is dense, which puts lots of those mobile electrons close to each other. You may further remember that I said the electrons themselves don't move (drift) very far under the influence of a voltage field, but they drift close enough together for electrons to hop into nearby atoms at the speed of light. Copper has one electron in its outermost orbital. When you get copper atoms en masse, the band these outer orbitals form has lots of room for more energy and electrons can drift readily from atom to atom and the charge is transferred to the next electron. The electrons that go into a wire are not the individual electrons that come out, but the energy is transferred just the same. (In quantum theory the idea of an individually identifiable electron is fuzzy indeed!)

Silicon

Okay. Armed with this theory, let's look at silicon.

When pure silicon crystalizes, it forms a stable outer orbital. Each of its four electrons is bound up in a covalent bond with another silicon atom. Stable outer orbitals have full energy bands, no free electrons without the application of grotesque amounts of energy, and so they don't conduct. The energy level is said to form a band gap.

Silicon, by itself, is an insulator.

In order to make semiconductors out of silicon, you dope it. Modern dopants are usually phosphorus or boron, and a wide variety of methods are used to get them into the silicon crystal matrix, from applying a coating and letting it bake right in to ion implantation, which is a high energy physics process.

Phosphorus' outer electron orbital contains five electrons. When you replace a silicon atom in the crystal lattice with a phosphorus atom, phosphorus forms the same four stable covalent bonds with the silicon atoms around it, but it also has a free electron, which broadens its energy band toward higher energy. The result is called an N type (for negative) semiconductor—one that can donate electrons. Now that it has a free electron, N type doped silicon conducts electricity.

Boron's outer electron orbital contains only three electrons. It will happily form three of the usual four covalent bonds with nearby silicon atoms, but it leaves a charge deficit. This extends the energy band toward lower energy levels, and the result is called a P type (for positive) semiconductor, which can receive electrons. Like N type silicon, P type silicon conducts electricity.

Diodes

You can probably see where this is going. When you put an N type semiconductor and a P type semiconductor together, interesting things happen. Some of the excess charges in the N type and the missing charges in the P type migrate toward each other, until they cancel each other out. This creates a depletion region on either side of the junction

where there are neither free charges on the N side nor free places for them to go on the P side. The result is that your semiconductors stop conducting. In fact, if you connect a battery in series, with its negative to the P type and its positive to the N type, electron charges released by the battery's chemistry push the negative electron charges away from the battery connection toward the P type, and the missing charges toward the N type. (Electron charges are always negative. A positively charged electron is called a positron and is antimatter. Seriously.) The positive end of the battery, by contrast, readily accepts the negative charges it attracts from the semiconductor. The result is that the depletion zone grows very large.

Flip the battery around, however, and the result is entirely different. The voltage potential of the battery pushes electrons away from the negative end, which transfers charges through the N type semiconductor toward the P type semiconductor. The inverse is also true. The positive end of the battery greedily slurps up any charges that come its way making more charge deficits. The depletion zone shrinks, and charges flow through the N type, through the junction, through the P type, and into the battery's positive terminal. The flow of charges is a current.

If this sounds like a diode, it is. If you wonder how the earliest crystal sets, using a phosphor-bronze needle scratching a chunk of galena (lead sulphide) as a detector worked, and why that detector can be replaced with a modern diode satisfactorily, here's the missing fact: galena is a naturally occurring semiconductor. What they made were diodes, just like the one we described. (Later radios used vacuum tube detectors which can, among other things, act as diodes, but the physics are entirely different. Electrons really do move around in vacuum tubes, but that's a topic for a different book.)

Transistors, At Last

So great, we've got a diode. We were talking about transistors, weren't we?

To make a diode into a transistor, you add another layer of semiconductor. If you're making an NPN transistor, the most common type, you put another layer of N type semiconductor on the other side of the P type, just like the name suggests. You'd probably expect that another depletion zone forms at this new junction.

It does.

If it sounds like we've stacked two diodes together back to back, well, we have, more or less. (Alas, wiring two diodes together in the real world does not produce a transistor. The thick combined base prevents control of the collector and emitter layers.)

The middle layer of a transistor is quite thin, allowing migration of charges to go through it, and also reducing the voltage (and thus, the current) needed to control its depletion zones with the other two layers. This layer is called the base, and it controls the resistance of the device (and thus, the flow of charges, aka current.)

The other two layers are called, respectively, the collector and the emitter. When a voltage potential is raised between the base and the emitter, just as in the diode, the base-emitter depletion zone is thinned, extra electrons and missing electrons become available, and the semiconductor layers conduct, resulting in a current. Because the base layer is thin, and there are suddenly a very large number of negative charges in the base layer this overwhelms the depletion zone between the collector and the base, allowing charges to flow from the emitter to the collector. The transistor becomes conductive

from collector to emitter, and a small voltage to change the state of the thin base layer can control the larger collector and emitter layers' resistance, and in so doing control the charges that can flow through the transistor, aka the current.

This is called a bipolar junction transistor.

Bipolar transistors come in two flavors: NPN and PNP. As the names suggest, PNP transistors have positive layers on the outside and the base is connected to the negative layer. They are the compliment to the NPNs we've been discussing. Instead of lowering their resistance (and therefore current) as the voltage potential between base and emitter goes up from zero as NPNs do, PNPs lower their resistance as the voltage potential goes down.

Remember that all voltages are relative. If you have an AC signal, some percentage of your waveform's voltage is below the zero volt line, so having a transistor that can amplify in that mode is very useful. A pair of complimentary transistors: a PNP and an NPN with the same response curves can build a simple, low-cost, efficient amplifier.

Those who've had electronics before are, right now, yelling "In BJTs, a small *current* controls a larger *current*. Not a voltage."

Mathematically, and according to Ohm's law, they're right. A small current (flow of charges) from the base to the emitter occurs, and controls the larger current that occurs between collector and emitter proportionally to this current. But it is not the current that causes the changes in the depletion zone. It is the change of potential (voltage) which does it. The base-emitter current is a biproduct, albeit one that is convenient to measure.

To whit: There is another type of transistor called a Field Effect Transistor. These are the transistors most common in integrated circuits. In FETs, a metal plate called the gate is electrically insulated from the semiconductor materials. This gate is used to manipulate the depletion zones of the transistor by raising a voltage potential on it. No (or almost no) current flows between the gate (analogous to the base) to the source or the drain, (analogous to the collector and emitter). It is the voltage potential alone that causes the migration of excess or missing charges to the right places within the FET to allow (or block) conductivity.

Sadly, the project in this chapter (I promise, there's a project coming, bear with me.) won't produce meaningful information when analyzing FETs.

The result of all this is a device in which a small current (BJTs) or a voltage potential (FETs) controls the resistance of the device.

A transistor is an electrically controlled variable resistor. It does not power anything itself. That's the power supply's job. Instead, it varies the output of the power supply in controllable ways to produce signals, whether DC pulses as we deal with exclusively in this book or AC as used in radio, switching power supplies, and the like. They're inexpensive singly and beyond cheap when sold in bulk in integrated circuits. Transistors are, as I said in the introduction, as common as dirt. They're nearly as cheap as dirt, too. Because dirt is partly ground rock and most of the rocks on Earth are silicates—mixtures of silicon and other elements—it can be said that in fact, transistors *are* dirt.

Kirchhoff's Laws and Voltage Dividers

So great, we have transistors. Now we can get to work, right?

Not exactly. Back in the land of normal physics, where we can talk about current as a thing rather than the movement of charges, transistors have strict limits on how much current they can handle. Think about it in terms of what we already know. A transistor has to be connected to a power supply and a ground to function, but they function by changing resistance. By lowering it, specifically. But how are you going to see that if the transistor is already connected to power and ground? Also, the transistor, like light emitting diodes, has a limited current it can handle before it goes up in smoke. If you're thinking about dropping resistors now, you're absolutely right, and you already know enough Ohm's law to know what dropping resistors do and how to figure the values of them.

Kirchhoff's Laws

What I glossed over in the discussion about dropping resistors is why they work. In order to explain that, we have to have a bit more theory, this time from a 19th-century German physicist named Gustav Kirchhoff. Specifically, we need to understand his circuit laws.

■ **Kirchhoff's First Law** The sum of the currents flowing into a node of a circuit (where several conductors are connected together) is equal to the sum of the currents flowing out of the node.

Currents don't wait around at nodes. The charges go someplace. Do something. Nothing goes to waste. So your circuit had better be able to handle the current you throw at it, either by using it or by limiting how much is available to the circuit. Also, if you put two resistances in parallel, the sum of the currents supplied to each one will equal the total current. This tells you that because you are "shorting" your resistance with another resistance, the effective resistance is lowered. The current is divided between the two resistances.

The second law is like unto the first.

■ **Kirchhoff's Second Law** The sum of the voltages in any closed circuit is zero.

Voltage is always relative. If you connect a light bulb (the old fashioned, incandescent kind) across a battery, it seems obvious that the lead connected to the + side of the battery and the lead connected to the - side of the battery will be the same as the voltage the battery puts out. (Assuming a battery with no internal resistance. These don't actually exist, but they make the math easier.) The filament of the bulb is dropping the entire voltage of the battery, soaking up the current, and releasing the energy as heat. (It gets white, glowing hot and releases photons. Light. The vacuum in the bulb is so the filament doesn't catch fire.) If you connect two bulbs in series, and the resistance of the first bulb is equal to the resistance of the second bulb, Kirchhoff's second law means that the voltage at the point where the first bulb connects to the second bulb will be half the voltage of the total circuit. The current, courtesy of Kirchhoff's first law, remains the same. Why are the two bulbs in series only half as bright? Because incandescent bulbs turn current into watts of heat. Watts = volts*amps. With half the voltage, we get half the wattage, thus half the light.

That's great, but I said that dropping resistors limit the current for the LED. How does that work?

LEDs are not resistive loads. They have a fixed forward voltage drop. If their input voltage is exceeded, their resistance drops off dramatically, current goes up, and they fry. Our dropping resistor limits the voltage to the LED, and since the voltage drop within the LED is fixed, they limit the current the LED receives.

So where does that put us with transistors?

Transistors, at least the BJTs we're studying, are current controlled variable resistors from a circuit standpoint. Since we know that the sum of voltages across series resistances is zero, and that a fixed resistor's voltage drop at a given current is also fixed (by the resistance and the current—Ohm's law) if we put a resistor in series with the emitter to base connection of a transistor, what happens?

We get a voltage divider.

Voltage Dividers

Two resistors, connected in series, with the measuring point between them. There are other types, particularly when dealing with AC, but this is the kind we'll be dealing with in this chapter.

Figure 5-2 shows a resistive voltage divider. Resistances in series get added together, just like with the incandescent light bulbs described earlier. Kirchoff again. If I have two resistors wired together and wire a battery from one resistor's bottom pin to the other resistor's top pin, the sum of the voltage potential from top to bottom will equal the voltage potential of the battery. One resistor drops part of the voltage and eats part of the current. The other resistor gets the rest.

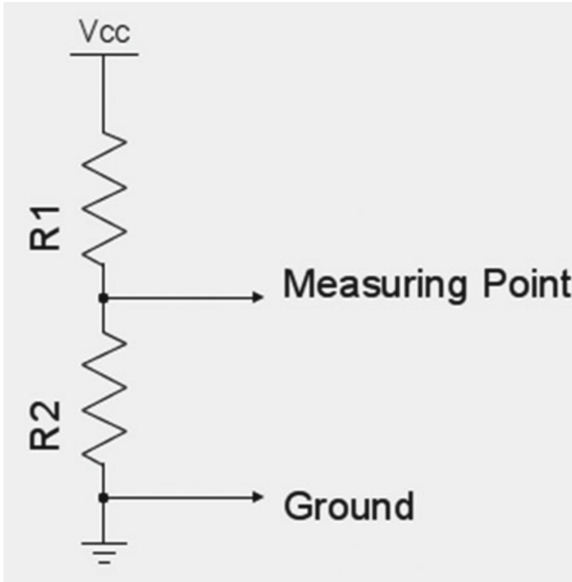

Figure 5-2. *A Resistive Voltage Divider*

Looking at the diagram, we know that these two 330Ω resistors equal one 660Ω resistor. We can crunch this with Ohm's law and say that to drop five volts in 660Ω, divide 5 volts by 660 you get about .0076 amps, or 7.6mA. If we calculate the watts, we know that our two resistors together are turning 5 times 0.0076 amps into heat, or 0.038 Watts, or 38mW. So because there are two resistors, will our 1/4 Watt resistors be beefy enough?

Let's do the math.

Kirchhoff tells us that we're dropping 2.5 volts per resistor. 2.5/330Ω is 0.0076 (or so) amps, or 7.6mA. 0.0076 times 2.5 is 0.019 watts. Our 0.25 watt resistors are more than up to the task.

It's easy to see that the voltage in the middle of this voltage divider will be 2.5 volts, because our resistors are equal. What if they're not?

You probably guessed there's a formula for that. It looks uglier than it is.

Vout=(R2/(R1+R2))*Vin.

Voltage output between our resistors equals resistance 2 divided by the sum of resistance 1 and two, all times Vin.

In our example divider, we can see how it collapses into dividing by two. R1 and R2 are equal, so Vout=R/2R * Vin. R/2R is the same as 1 over 2, or 1/2.

Remember that our transistor is a current controlled variable resistance. If we want to calculate whether it's going to stay within its operating parameters (and not fry) we'd figure its minimum resistance in this circuit and figure the current it would be carrying. Unfortunately that minimum resistance is a variable depending on how you set the transistor up. However, the manufacturers do thoughtfully include the collector current limit, so for component safety values, we can use that.

If our collector resistor (called the load resistor in a transistor circuit) is 330Ω, and the supply is five volts, even if our transistor goes to zero ohms (which it can't) our current is .015A, or 15mA. 5V/330Ω=0.015. A 2n2222A transistor, one of the most common bipolar types, is rated at 800mA (continuous) of collector current. A 2n2222A can handle the current we're throwing at it, and a whole lot more.

How much will it swing the voltage? And how much current is required from base to collector to do it? Good question. To calculate that easily, we'd have to know the resistance of the transistor, but since that resistance is a function of the circuit the transistor is in—collector voltage, emitter voltage, base-emitter current, we have to do more calculation.

Transistors in Voltage Dividers

The current gain of a transistor, how much bigger the collector-emitter current is than the base-emitter current, is a measure of the transistor's amplification. Mathematically, it is equal to collector current divided by base current (for BJTs and similar transistors. FETs are different). Like Ohm's law, this formula is subject to algebraic manipulation.

To return to the previous example, with a collector current of 0.015A (15mA) at five volts on a 2n2222A, if I want the 2n2222A to actually try to dissipate all 15mA, and drop the voltage as close as possible to zero, how much base current do I need?

Well, if I multiply both sides of the equation by Base_current, I get gain*Base_current=Collector_current. Divide both sides by gain, and I get Base_current=Collector_current/gain.

Collector_current is 15mA. The On Semiconductor datasheet for the 2n2222A, and the gain chart therein, tells us that the gain on a 2n2222A with 15mA of current on the collector should have a minimum gain of 110. Divide 0.015A by a gain of 110, and you get about 0.00014A or 0.14mA. Engineering practice seems to use a base current of about 0.1 times collector current for saturated switching, so this is about right. The 2n2222A saturates with a base voltage of between 0.6V and 0.8V with 15 mA of collector current. So our base parameters are clear: 0.7v at 0.14mA will saturate a 2n2222A for a 15mA load. The voltage resulting from the voltage divider will approach zero.

Build the Transistor Analyzer

Some years ago, I purchased a grab bag of transistors from a well known electronic supply house. They were *very* cheap, fractions of a penny each. I soon found out why. The manufacturer whose leftovers they were had, in a misguided attempt to prevent his or her competitors from reverse-engineering the circuit, washed all the markings off the transistors. All of them are bare metal cases with three leads coming out the bottom. That's all.

Because I would like to use these transistors in projects where the exact specifications aren't critical (digital applications where they will be saturated or completely cut off) it would be helpful to know what type of transistor they are (NPN or PNP) and what the pinouts are.

The transistor analyzer we're going to build (Figure 5-3) does part of that job. It turns the transistor and two other resistors into a voltage divider network and uses that network to explore the anatomy of the transistor and determine what type it is, and where the base pin is.

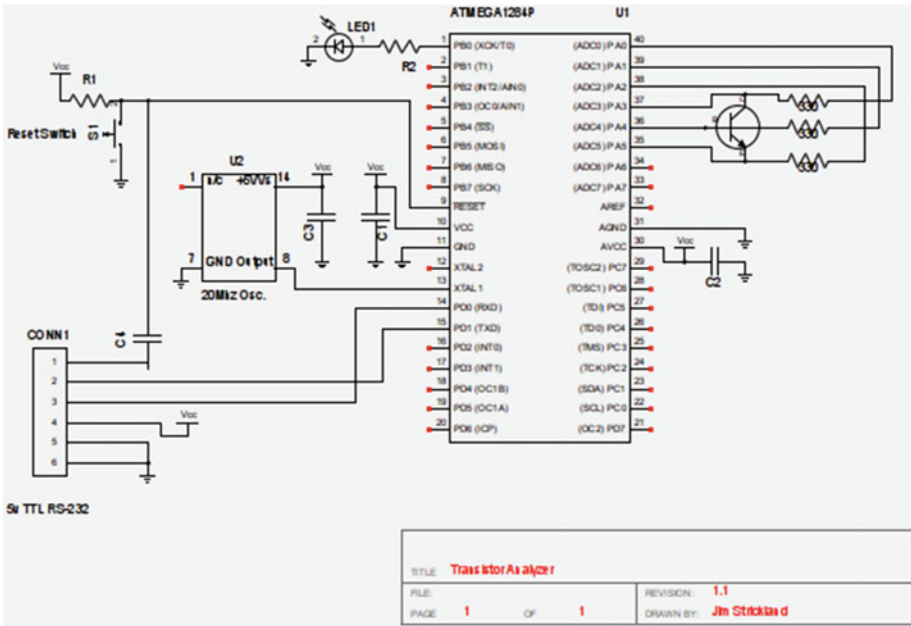

Figure 5-3. *The Transistor Analyzer*

Given those pieces of information, determining the other two pins is at least simpler.

■ **Note** The device we're building here may damage or destroy really small signal transistors and FETs, but most normal BJTs and power transistors should be fine. We're only hitting them with these voltages and currents for fractions of a second.

Construction

Our analyzer is a fairly simple animal.

As you can see from the schematic in Figure 5-3, our analyzer uses port A. Pins A0-A2 (40,38, and 38) are the pins that set up the circuit for the transistor, we'll use A3-A5 (37, 36, and 35) as analog inputs to measure the voltages our circuit produces.

Refer to Figure 5-1. and install the three 330Ω resistors across the slot in your second breadboard even with ATmega 1284P pins 37,36, and 35. Wire pin 40 of the ATmega to the "far" end of the top resistor, pin 39 to the far end of the middle resistor, and pin 38 to the far end of the bottom resistor. Then wire pin 37 to the top resistor's other end, pin 36 to the middle resistor, and pin 35 to the bottom resistor. Make sure to leave several empty tie points on your breadboard between the end of the resistor and pins 35-37, so you have a place to plug in the transistor.

That's all there is to the construction. It's a really simple circuit. The devil is in the details, or the software in this case.

There's one other thing. Having built the circuit, let's talk about the pins in their real order from now on. Remember. Pin 40 on the ATmega1284P is the 0 pin of port A *and* Analog 0. 39 is PA1 and Analog 1, and so on. Left to right, if you're looking at the Cestino from the pin 1 side. For convenience (mostly mine) let's call them A0, A1, A2, and so on through A7. The A can mean either Analog or Port A.

Wait. Didn't I say there were voltage dividers in this circuit?

There are.

To form a circuit, a voltage potential must exist. There must be, to back away from the theory a little, a power and a ground. We know that, like Port C, the pins of Port A can source current, up to 40mA per pin. They can also sink 40mA per pin as well. A Cestino pin, like any Arduino pin, can be a power or a ground. We know from the chapter on theory that a transistor has two circuits in it, the base-emitter circuit, and the collector-emitter circuit, and I'll tell you that we're going to have them share a ground on the emitter. So that pin on the Cestino will be set to 0, low, or ground. (Same thing, in this case.)

Imagine a transistor connected like the one in the schematic in figure 5-3, with its collector on A3, its base on A4, and its emitter on A5. (This is actually the most common layout for BJTs, but it's not a standard.)

If we want to turn on the base of the transistor, we'd set A1 to 1. The voltage goes from pin A1, passes through a 330Ω resistor to the base leg of the transistor. Meanwhile, if we set pin A2 to zero, it becomes a ground, which is also connected to a 330Ω resistor, and from there to the emitter leg of the transistor. There is now a voltage potential between the base leg of the transistor and the emitter leg, and both of those legs are connected to the Cestino by a 330Ω resistor.

We're using pins A3, A4, and A5 as analog pins to measure the voltage present on those pins in the middle of our voltage divider.

In Figure 5-4, below, I've rearranged the circuit layout to make the voltage divider circuit we're building clearer. If we set A0 high, and A2 low, we wind up with a three component voltage divider, with the transistor in the middle. The resistance of the transistor is controlled by A1, which is wired (via a resistor not shown in the diagram) to the base.

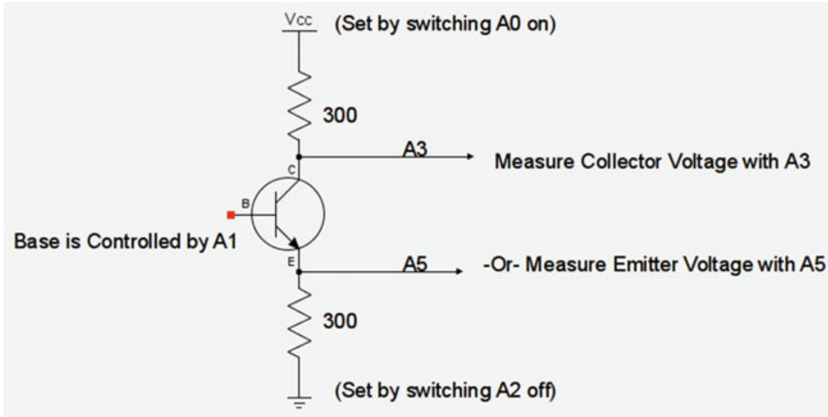

Figure 5-4. *Voltage Divider Detail*

Wait. How do we solve a three-component voltage divider where one of the resistances is unknown?

We don't have to. Recall that Kirchhoff lets us add resistances in series, so we add our unknown: the resistance of the transistor base-to-emitter circuit, to a known: the resistance of the two 330Ω resistors and solve that.

Cutting and pasting the voltage divider formula from the theory part of this chapter, we know that Vout=(R2/(R1+R2))*Vin. We know that Vin is 5V. We know that R2 is 330Ω. We know that R1 is really R1a+R1b, where R1a is our other 330Ω resistor, and R1b is the unknown resistance of the base-emitter circuit of the transistor, but we'll ignore that for now and solve for R1.

First, let's get Vin out of the way. Vin*(R2/(R1+R2)) is equal to Vin/1*R2/(R1+R2)

Multiplying fractions is straightforward: you multiply the numerators (top numbers) together and multiply the denominators together. Simplifying the resulting mess is what made it painful in school, but we're not doing much of that here. Multiplying gives us Vout=(Vin*R2)/(R1+R2). That breaks it open for us. Vout*(R1+R2)=Vin*R2. (Vin*R2)/Vout=R1+R2, and R1=((Vin*R2)/Vout)-R2.

Now we can break out R1a and R1b. R1a+R1b=((Vin*R2)/Vout)-R2.

Subtract R1a from both sides and we get (finally) R1b=((Vin*R2)/Vout)-(R2 + R1a).

Looks ugly. Fortunately, we know that R1a and R2 are the same (330Ω, most likely, but we might want to change it some time, so let's leave it a variable.) We'll call it RESISTOR. That gives us R1b=((Vin*RESISTOR)/Vout)-(RESISTOR*2). If we fill in the constants we're not changing, that Vin is 5 volts, it gets even simpler. R1b=((5*RESISTOR)/Vout)-(RESISTOR*2).

So to get the resistance of the base-emitter circuit of the transistor in our analyzer, all we have to do is multiply 5V times RESISTOR, divide it by the voltage on that leg of the transistor, and subtract twice the value of RESISTOR.

The Code

Let's write some code.

We're going to be using some higher math functions in this sketch (in case you hadn't guessed), so before the sketch does anything, let's go ahead and include the math.h library.

```
#include <math.h>
```

We define RESISTOR here, as a preprocessor macro.

```
#define RESISTOR 330
```

If you used resistors other than 330Ω, make sure to put that value in for RESISTOR instead of 330.

Next, we need a function to read the voltage. There is, of course, a catch, and if you've been doing Arduino for any length of time, you already know it: the analog-digital converter returns, via analogRead(), an arbitrary value between 0 and 1023 (it's a 10 bit AtoD) where 1023 represents the reference voltage.

We can fix that. We know our reference voltage is 5V. 5/1024 turns out to be 0.0048828125. Yeah, ew. I'm cutting and pasting that one every time. Given that value, all we have to do is grab the value of the analog pin, multiply by our nasty little floating point number, and we're done.

```
float ReadVoltage(int pin) {
  return (analogRead(pin) * 0.0048828125);
}
```

Now we can calculate the resistance by solving the resistance dividing network, just like we talked about earlier. Because RESISTOR is a preprocessor macro, we can set it to whatever we want up there, and it will be filled in down here.

```
float ReadResistance(int pin) {
  float temp = ((RESISTOR * 5) / ReadVoltage(pin) - (RESISTOR * 2));
  return (temp);
}
```

The next routine determines whether the transistor is an NPN or a PNP. From the discussion on theory, you will recall that an NPN transistor is essentially two diodes with a common cathode (the base). This means that if a transistor is NPN, there should be a low resistance between base and collector, and a low resistance between base and emitter, but a high resistance between emitter and collector, since any current attempting to flow that way will be expanding the depletion zone of either the base-emitter junction or the base-collector junction, depending on the polarity. (Current really flows from negative to positive, as electrons turned out to be negatively charged, something Benjamin Franklin simply could not have known when he coined the terms. The calculations work either way for most things at this level of abstraction, as long as one is consistent.)

The three transistor analyzer circuits give us a convenient way to measure resistance from any leg of the transistor to any other leg of the transistor by changing voltages and by which leg we measure. This is how we'll determine the type of transistor we are looking at. If we find a transistor has one and only one pin with low resistance to both other legs of the transistor, we know the transistor is an NPN.

PNP transistors, by contrast, conduct from collector to emitter when the base is low, and both collector and emitter will have low resistance to the base as well. Testing explicitly for a PNP transistor would be a lot more complicated. Fortunately, since we have a simple test for an NPN transistor, anything else must be PNP (or bad.)

This analysis is exactly what the next piece of code does. It raises one pin and measures the resistance with the other two voltage dividers that result. If it finds one and only one pin that produces low resistances to the other two pins, the transistor is NPN, otherwise it's PNP.

Here's the code that does it. It's another function.

First, we initialize all our variables. I'm old fashioned. I like my variables declared and initialized at the top of the module. You don't have to do it that way, but I find it easier to debug.

```
bool is_npn() {
  bool npn = false;
  bool found_one_set = false;
  bool inf_or_negative = false;
  float temp = 0.0;
```

Next, we need to switch on pins A0, A1, and A2 in order. We could do it with a counter and some logic, but we know from chapter 4 that bit shifting a 1 to the left does that automatically, so we can build a for loop that does it that way. We also reinitialize our inf_or_negative flag, that tells us if a series of measurements has been disqualified, to false every time we loop.

```
  for (PORTA = 1; PORTA <= 4; PORTA = PORTA << 1) {
    inf_or_negative = false;
```

Now we take the measurements, from A3 to A5, inclusive, and set the inf_or_negative flag if any of them are infinite or negative resistances. (When we hit the pin we've turned on, we'll get a 0 resistance.)

```
    for (int c = 3; c <= 5; c++) {
      temp = ReadResistance(c);
      //Any infinite or negative resistances will disqualify this PORTA value.
      inf_or_negative = inf_or_negative || ((isinf(temp)) || (temp < 0));
    }
```

If we get three measurements with no infinite or negative resistances, we might have an NPN transistor, but only if this line is the only one. We check the found_one_set flag to see if it's been set before. If it has, we can go ahead and have the function return false, as the transistor cannot be a good NPN transistor. If it hasn't, we set found_one_set to true.

```
if (!inf_or_negative) {
      if (found_one_set) {
        return (false); //If we found more than one valid set, the transistor is
                        //not NPN.
      } else {
        found_one_set = true;
      }
    }
  }
```

If we reach the end of the PORTA loop with found_one_set, then we have an NPN transistor. If we reach the end of the PORTA loop without the found_one_set flag set, then we have a PNP or a bad transistor. Either way, we can just return the value of found_one_set once we've reached the end of the PORTA loop.

```
  return (found_one_set);
}
```

Let's review. The function's name is is_npn. It returns a boolean, and takes no arguments. It loops using the PORTA register as the loop index, and increments the loop by left-shifting the value of PORTA.

Inside the PORTA loop is a second loop, indexed on c, which counts from 3 to 5, which correspond to A3-A5, the analog pins. For each of the analog pins, we take a resistance measurement, and logically determine whether that resistance is infinite or negative with inf_or_negative. This means that if inf_or_negative is set true once, it will remain true regardless.

When the c loop exits, we evaluate inf_or_negative. If it's false, we check found_one_set. If found_one_set is also false, we set it. If it's true, we return logical false, since the transistor can't be NPN and must be PNP or bad.

When the PORTA loop exits, we return found_one_set. If it's true, and we reach the end of the PORTA loop, we must have found only one set, and the transistor is NPN, otherwise it's PNP. Either way, we return the value of found_one_set. Here's the whole function in one piece, with comments.

```
// ----------------------------------------------------------
// An NPN transistor will have one and only one pin with low
// resistance to the other two pins. Search for that pin. If we
// find more than one or we don't find any, it's a PNP
// transistor.
// ----------------------------------------------------------
bool is_npn() {
  bool npn = false;
  bool found_one_set = false;
  bool inf_or_negative = false;
  float temp = 0.0;
```

```
  for (PORTA = 1; PORTA <= 4; PORTA = PORTA << 1) {
    inf_or_negative = false;
    for (int c = 3; c <= 5; c++) {
      temp = ReadResistance(c);
//Any infinite or negative resistances will disqualify this
//PORTA value.
      inf_or_negative = inf_or_negative || ((isinf(temp)) || (temp < 0));
    }
// if we didn't have any infinite or negative resistances,
// for this PORTA value is a good one, but we need to make sure
// there's only one.
    if (!inf_or_negative) {
      if (found_one_set) {
        return (false);
//If we found more than one valid set, the transistor is not
//NPN.
      } else {
        found_one_set = true;
      }
    }
  }
//if we found one and only one valid set (PORTA value) then the
//transistor is NPN.if we found none, it's not.
  return (found_one_set);
}
```

One more function to go.

You might have noticed that the one valid set of measurements which produces no infinite or negative resistances is taken between the base and the other two pins with the base high. In effect, we've already found the base. This method works great for NPNs but not at all for PNPs. So the code for determining the base of the transistor needs to know whether the transistor is an NPN or a PNP, and obviously it needs to return a pin number. That's how we declare it. We also declare a boolean flag, reject, and initialize it, and declare a float, temp, and initialize it.

```
byte base_pin(bool is_npn) {
  bool reject = false;
  float temp = 0.0;
```

If the npn flag is set, we basically repeat the test used by the is_npn() function, only this time we keep track of the pin number and return it when we find a series of resistances that have neither infinite nor negative values. Note that the flag npn is passed into this function when it's called.

```
  if (npn) {
    for (PORTA = 1; PORTA <= 4; PORTA = PORTA << 1) {
      reject = false;
      for (int c = 3; c <= 5; c++) {
```

```
        temp = ReadResistance(c);
        //if the value is infinite, this is not the base.
        reject = reject || ((isinf(temp)) || (temp < 0));
      }
      if (!reject) return (PORTA);
    }
```

If the npn flag is not set, we know we're dealing with a PNP transistor, and we need to find which pin has negative resistances to both the other pins, because it is presently conducting. This will only occur with one pin off and two pins on, and the values where that occurs in 3 bits are 3, 5, and 6. So we set up an array with those values, and iterate through the array with the test_pattern_index loop, and initialize reject as false. We're reusing the reject variable, but it doesn't mean the same thing in this context.

```
  } else {
    byte pnp_test_patterns[] = {3, 5, 6};
    for (int test_pattern_index = 0; test_pattern_index <= 2;
    test_pattern_index++) {
      PORTA = (byte)pnp_test_patterns[test_pattern_index];
      reject = false;
```

For every cycle of the test_pattern_index loop we take the usual three measurements on pins A3-A5 inclusive. If we *don't* get a negative or zero resistance, then we don't have the base low.

```
      for (int c = 3; c <= 5; c++) {
        temp = ReadResistance(c);
        //we should have negative resistance for any pin
        //conducting to the base. If the value we get isn't
        //this is not the base.
        reject = reject || ((isinf(temp)) || (temp > 0));
      }
```

Once we exit from the c loop, if reject is false, we have found the base pin, but the first three pins of PORTA are set to its inverse. With our binary skills from Chapter 4, we know we can take the inverse of PORTA, but that's going to turn on a lot of 1s that we haven't used at all. We're really only interested in the first three bits of PORTA, so we invert PORTA and bitwise AND it with 0b00000111, and return the result.

```
      if (!reject) return ((~PORTA) & 0b00000111);
    }
  }
}
```

These four functions do all the work. There's another function, along with the contents of setup() and loop(). These are about making our results human-readable and sending them to the serial console of the Arduino application.

The first function is called log2. It takes a value and returns a byte. Here's the code, in its entirety.

```
byte log2(double val) {
  return ((byte)round(log(val) / log(2)));
}
```

What on Earth is this for? Well, when we want to go from a binary value n say, 0-3 to a value that will switch on the corresponding pin; A0, A1, or A2, for example; we can take 2^n and get that value. (Because we're going from floats to bytes, we need to use the round function.) The code would look like this: round(pow(2,n)).

Going the other way, from pin number to binary value, is a lot more complicated. Most C/C++ math libraries include a log2 function, which does exactly that, but Arduino's math.h library does not. Fortunately, if you take the natural log of a number and divide it by the natural log of 2 (all floating point division) you get the equivilant of log2, and that's exactly what this function does. return the value, as a byte, of the round of log(val) divided by log(2).

Floating point operations are not especially fast on the Cestino. The ATmega line doesn't have a built in floating point unit. But we're doing it in the display section, where the speed is limited by our serial communication and how fast we can read. Not a problem. Speed is relative.

In setup(), we set the serial console to 115200, as usual. Then we initialize DDRA so that only pins A0, A1, and A2 are outputs. We take the extra step here of setting PORTA to 0 to make sure all the pull-up resistors are turned off, as they will badly distort our analog readings if they're on.

```
void setup() {
  Serial.begin(115200);
  DDRA = 0b00000111;
  PORTA = 0b00000000;
}
```

At the start of loop() we initalize transistor_is_npn to false, and pretty-print the header for the output, along with the description of what data we're about to throw at the user. Then we set transistor_is_npn to the output of is_npn(), and pretty-print that result.

```
void loop() {
  bool transistor_is_npn = false;
  Serial.println("Transistor Analyzer");
  Serial.println("-------------------------------------------------------");
  Serial.print("Transistor Type:");
```

```
  transistor_is_npn = is_npn();
  if (transistor_is_npn) {
    Serial.println("NPN");
  } else {
    Serial.println("PNP");
  }
```

Next, we call base_pin() with transistor_is_npn as its parameter. Then take the log2 of the result, add three because with our layout, it's easier to see which transistor leg is connected to the analog read pins than it is to see which of our outputs it's connected to. Finally, we print the result with a header that tells what it is, including an A to concatenate with the pin number to make it clear. Then pretty-print the bottom line of the output, and we're done. More or less.

```
  Serial.println("Base: A" + (String) (3 + (log2(base_pin(transistor_is_npn)))));
  Serial.println("-------------------------------------------------------");
```

Remember that loop() is called over and over again. For some transistors, the voltages and currents we're feeding them are at or near their limits. For others, the values may be somewhat higher than the transistor is really rated at. It's a good idea, once we've gotten the information we were after, to turn everything off and leave it off, rather than risk heating the transistor up to the point where it's damaged. We can go ahead and do that over and over again forever.

```
  while (0 == 0) { //As long as we're doing nothing,
    PORTA = 0;   //Turn off all pins so small transistors don't get hot.
  }
}
```

Credit Where Credit's Due

Where do I even start? The procedure for testing a transistor with a multimeter to determine its type (NPN or PNP) is documented in numerous sites online.

I'm not a physicist by trade or education, so the physics presented here are the result of much reading on my part, starting with Dr. Leon Lederman's *The God Particle* (1993), and much other reading including (inevitably) Wikipedia. Most explanations of transistor physics content themselves with Bohr's electron shell/orbital model, but having read Lederman, I thought the underlying quantum theory was fascinating. I set out to describe that. It proved to be a much larger undertaking than I imagined.

Fortunately, a friend of mine, Geoff Alleger, is a physicist, knows his quantum theory, and had the time to look over Electron Orbitals, Bands, and a Dab of Quantum Theory, and to critique it. If I subsequently went off the rails, at least I didn't start out there.

Further

In this project, we dove into transistors at some depth. In the process of explaining what a transistor *is* we dove into a startling amount of physics. In the process of explaining what a transistor *does*, we dove into Kirchhoff's laws and voltage dividers. Then we built an analyzer that peeks at the transistor's anatomy via resistance measurement (using voltage dividers) and determines what type the connected transistor is and where the base pin is.

The project itself has a clear, well established further direction. It is possible, through the addition of more resistors and much more code to determine the transistor's full pinout, its amplification factor, and to test whether it's good or not. Furthermore, that same physical setup can be used to test a wide variety of other components. I know it's possible, because there is a project that does exactly that: `https://github.com/lowvoltage/Transistor-Tester`. Various iterations of this project are produced in Chinese factories and are sold as finished devices or kits on the internet, but this website offers schematics and code for free. Be advised that the code is not Arduino code. It's written for Atmel's own development tools, and there's something of a cultural divide between the AVR hackers and Arduino folks that you should be aware of should you need ask for help.

As for transistors, look around you. The entire world is full of transistors and things made from transistors. If you want to make a full adder? you can do it with a pile of transistors. There are a huge variety of designs and schematics to be found on the web.

Want to make a few bytes of DRAM? You can do that too. I'd suggest looking up the Wikipedia article on DRAM, the one on DRAM memory cells, and probably Dennard's original 1968 patent on single-transistor single-capacitor DRAM cells, #3387286. remember. 128 *billion* or so of those in the memory of the computer I'm typing this on.

Want to build a transistor radio or an amplifier for your guitar? They're out there too. Our entire civilization is built on transistors, so there are lots of projects you can do with an understanding of them.

Not bad for dirt, eh?

CHAPTER 6

TTL: The Missing Link

There are billions and billions of FETs in any modern desktop computer. But what do they *do* exactly? How do you get from unimaginable billions of transistors to computation?

You already know some bitwise logic. Transistors can do that.

Last chapter, we talked about NPN transistors with a load resistor on the collector. In this configuration, as the current is raised on the base, the voltage available at the node where the load resistor and the collector are connected *drops*. If that sounds like a logical NOT, as usual, you've got a good ear. Let's work with that.

Imagine for a moment that you had a circuit like that in Figure 6-1. On the left are two NPN transistors, Q1 and Q2, with their collectors and bases tied together. The bases are wired to the positive bus via a large resistor. Call it 4kΩ.

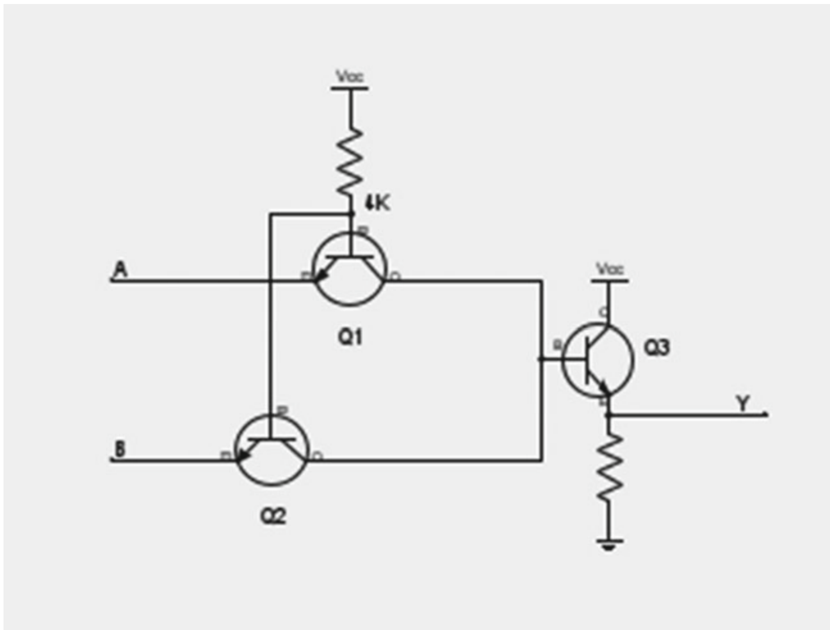

Figure 6-1. *NAND Gate*

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_6

If we ground both emitters, Q1 and Q2 both conduct, raising the base of Q3 so that it conducts, which raises point Y high.

If we raise the emitter of Q1 or Q2 close to the positive bus, that transistor ceases to conduct, as current no longer flows from its base to its emitter. The other continues, however. Q3's base remains high, and Q3 goes on conducting, and point Y remains high.

If, however, we raise both emitters, it turns off both Q1 and Q2. Q3's base goes low. It will stop conducting, and point Y gets pulled low. Let's turn that into a truth table.

Table 6-1 shows the truth table of the circuit we've just created. If point A, the emitter of Q1, is high and point B, the emitter of Q2, is low, point Y, which is on the emitter of Q3, is pulled high by the transistor. The same thing happens if A is low and B is high. If both A and B are low, Q3 goes on conducting just fine, and Y remains high. But if A and B are both high, Y is pulled low by the emitter resistor to ground on Q3.

Table 6-1. *Truth Table for the NAND Gate in Figure 6-1*

Q1 Emitter	Q2 Emitter	Point Y
Low	Low	High
Low	High	High
High	Low	High
High	High	Low

In terms of logic, this is an AND of the inputs A and B, with a NOT applied to it. The logical name of this is a NAND, for NOT-AND.

If you got the Texas Instruments datasheet for the 74xx00 (available here: `http://www.ti.com.cn/cn/lit/ds/symlink/sn74ls00.pdf`) flip to page 3. You can see the rest of the schematic. It looks a little different. Q1 and Q2 are combined into a special tetrode transistor with two emitters. It does the same thing. The diodes protect the tetrode transistor by preventing the inputs from going lower than the circuit's ground. The Q3 transistor, in turn, is tied to what's called a totem pole output, which is an amplifier circuit. If you set out to build the gate the way it's shown in that schematic, it should work.

Now imagine you need a lot of NAND gates. Consider that for each time we use AND or NAND in the Cestino, we're processing eight bits at a time. Imagine etching all the components in the NAND circuit into a tiny piece of silicon, many times over, into an integrated circuit.

The datasheet gives it away. The 7400 is exactly that. It was released in 1964 by Texas Instruments. The rest is history.

The Stuff You Need

This project takes remarkably few parts, and all of them can come from the junk box.

New parts

None!

New or Used Parts

You will need some 74xx00 series logic ICs. Nearly anything in a 5v family will do, and most of the types you can plug into your breadboard are 5v. Some types are more interesting than others. If you're buying new ones or borrowing from friends, try to get a 74xx00 and a 74xx92 or 74xx93, where XX is LS, or AHCI or F, or even nothing at all (the plain 7400, and 7492 will work fine.)

What TTL Is

TTL. There are those initials again. Didn't it mean 0 to 5 volts, as opposed to RS232 levels which are + or - 12v? Yes, it did. Didn't it have something to do with our 20Mhz full can TTL oscillator? Yes, it did. How does all of this tie together? What is TTL, anyway?

TTL stands for Transistor-Transistor Logic. TTL chips like the 7400 are designed to do two things. They implement one (or more) logic functions and they also have inputs designed to handle exactly the kind of signals their outputs produce, so you can interconnect them. For 7400 series logic, these signals run between 5V and 0V. The inputs require a certain amount of current to trigger them, and the outputs are capable of generating many times that much current, so multiple inputs can be connected to a single output pin.

There are other kinds of electronic and electromechanical logic. You can do logic with diodes, but the voltage and current drops limit how much you can chain together. You can do it with relays, but it's not very fast, not power efficient, and it's enormous. You can do it with vacuum tubes (valves, if you're in the United Kingdom), and in fact, that's how it was done in the earliest days of computers, but that's a topic for a whole different book.

TTL is more than a type of IC, however. It's the way those ICs communicate, the +5 volt to 0 volt communication. The timing. The lack of analog circuitry. I said that the 20Mhz full can oscillator that drives the Cestino is a TTL device, and it is. By nature, crystal oscillators produce a sine wave: a series of voltages that form a curve over the wavelength, with peaks and valleys of the specified voltage. That's not what a TTL device wants to see, so a TTL oscillator adds output circuitry that turns on when the oscillator's output goes above a certain voltage, and off when it goes below. The output circuitry further ensures that the positive and zero pulses are of equal length.

Not all oscillators are used to produce symmetric on and off pulses. In the datasheet for the full can TTL oscillator we're using for the Cestino, this is expressed in terms of output symmetry. It should be 50 percent under typical conditions. Some datasheets may also refer to this as the duty cycle, which is the percentage of time in a full wave (one off and one on pulse) the output of the oscillator is on. We'll revisit duty cycles, among other things, in Chapter 11.

The Extended 7400 Series Family

The family of logic we're going to deal with most, because it's the most common and I'm most familiar with it, is the 7400 series. This series takes its name from the 7400, which we just examined, but it's enormous. The Wikipedia article at `https://en.wikipedia.org/wiki/List_of_7400_series_integrated_circuits` lists hundreds. All of them are input/output compatible. They use the same voltages, the same signaling conventions, (although whether they're active high or low is a matter for the datasheet), and have similar speeds. They're designed to be compatible with each other.

There is, of course, a catch. Quite a few catches, in fact. There is more than one 7400 TTL standard. Usually the whole family is referred to as the 74xx family for this reason. I mentioned in Chapter 1 that you needed a 5V Arduino to get through chapters 2 and 3. This is because the newest Arduinos are 3.3V TTL, and can only be connected safely to TTL operating at 3.3V. If you hook them to 5V TTL, they will fry. There are whole branches of the 74xx family (the 74LV and 74LVC types) designed to operate at lower voltages with such ICs, although you usually encounter them only in surface mount (SMD) packaging. They're often 5V input compatible, but they want a Vcc that's lower.

Also, different families of 7400 have different levels of current available at their outputs, and their trigger voltage thresholds are different, so a 74HC00, a high-speed CMOS version of the 7400, would have real trouble producing the current and voltage needed to drive its 7400 or 74LS00 ancestors. When you build with 7400 series logic, make sure the families you use are compatible with each other.

Types of TTL

7400 series TTLs come in a huge variety of types, each one a little piece of a larger piece of logic. The 7400 is a quad-two-input-NAND gate IC. It has four gates, each with two inputs and one output, and each gate has NAND functionality—the output goes to 0 (low) when both inputs are 1 (high).

There, many other types. Some have specific logic functions like the 7400 does, but many others implement some other small computing need. There are also variants with different input or output specifications, for interfacing with other ICs.

Some examples:

- The 7402 is a quad 2-input NOR gate. Each output is 1 (high) only on when both inputs are low (0).
- The 7403 is another quad 2-input NAND device like the 7400, but it has open collector outputs, meaning the output is tied to the collector of an internal NPN transistor. This requires the use of an external pullup resistor, but it allows the output pin to sink 25mA.
- The 7404 is a hex inverter. Six logical NOTs. These are useful when you have an active low input that you'd like to trigger with an active high output. Or six of them, in this case.

There are also hex-inverting Schmidt triggers, which are mostly useful for following analog voltages and switching on at one voltage, but off at a different one. Like I said, very specific jobs.

The 7492 is a divider IC. Actually three of them in one package, although you can only access two. If you feed one of its inputs, it will divide your pulse count by two, turning on the output for every other pulse. If you tie its dividers together, it will divide by 12. You can use these ICs as counters as well, although it's important to keep in mind what each stage is doing so the results aren't unexpected. We'll talk about this one at more length in the TTL Tester project section later in this chapter.

The 7493 is nearly identical to the 7492, except that it is a binary counter, will count from 0 to 15, and will produce correct binary values on its outputs. It can also be used as a divide by 16, divide by 8, or divide by 2, depending on how it's wired.

There are buffers and line drivers, which simply repeat the signal that's sent to them, but allow more current to be drawn or isolate the logic circuitry from whatever's on the other end, and so on. There are shift registers, which allow one or more bits to be put in, and then bit-shifted up or down through the outputs. These are useful, for example, when you want to hold a specific binary value on the outputs, but you're not in a hurry and don't want to spend a lot of your controlling device's pins on the input.

Latches also take a binary value in, but they take it in parallel and hold it until instructed not to.

There are even specialists like the 74141, which takes binary input and drives an ancient display device called a Nixie tube, a neon light–based device used before the advent of LEDs. Nixies have been obsolete a long time, so their dedicated driver devices are hard to find (look on eBay) and pricey. The 74143 and 74144 are similar devices for driving 7 segment LEDs. Given the limited current capabilities of some logic, it's easy to understand why these specialists were necessary, but when it comes time to drive our own 7 segment LEDs in Chapter 10, we'll do it with a Cestino port.

There are flip-flops, which allow you to set an output bit to a given state, reset it, or toggle it from one state to the other.

There's even the 7483, a 4 bit full adder, meaning you can put a 4 bit binary in on each of its two inputs, as well as a carry input, and it will add the two numbers and give you the result. If you stack two of these together with the carry-output of one connected to the carry-input of the other, you can add eight bit numbers.

Whew. This is almost microprocessor stuff.

Actually, it's part of a CPU. An adder of some bit-width, usually 16 bits for the earliest 8 bit processors, is a necessary part of any CPU, where it is used as the program counter: how the CPU keeps track of where it is in memory.

If you're asking yourself if our nerd ancestors might have wired up whole CPUs out of TTL as a hobby, you're asking the right question. Not only did they do it as a hobby, they did it in industry too. And when they did, the minicomputer was born.

Why TTL

74xx TTL was, and is, a big deal. In the 1960s and 1970s, when the 74xx family was first introduced, it allowed much more rapid prototyping, simply by allowing boards to be wire-wrapped together, or later using solderless breadboards like ours. Instead of taking days just building and testing a given logic gate, you pulled one out of the box and hooked it up.

What could you build with it? Well, with enough logic, you could build a whole CPU. Consider that for a moment. A CPU is more or less a group of logic functions bundled together that run in synchronous (usually) to an external clock pulse. Your program calls these logic functions in a particular sequence. A CPU, in the end, *is* logic. TTL made it possible to build CPUs smaller and faster than ever before. As I said earlier, thus was the minicomputer born.

Minicomputer didn't mean what it means today. A minicomputer wasn't a tiny board like the Raspberry Pi, or (arguably) the Arduino itself. These were decades away. Minicomputers were computers whose size was reduced to merely a few file cabinets from the room sized mainframes that preceded them, by the extensive use of TTL. They exploded onto the market, prying it away from the likes of IBM and other mainframe makers, who built their logic up from single transistors. TTL made that possible. If this seems like ancient history of no special relevance today, consider this:

Prior to the age of minicomputers, most programmers and analysts interacted with their machines with stacks of punch cards, or at best magnetic tapes. You coded up your project, handed the stack of punch cards or the tape to your friendly operator. When your time slot came up on the computer, they loaded your cards or tape, ran your job, and collected the output, either on another tape or on printouts. You might get it back tomorrow. You certainly got a bill. Mainframes and their human infrastructure were expensive.

The rise of minicomputers, by contrast, began the era that we in the 21st century would recognize. You talked to minicomputers with print and later video terminals. You ran your own programs, and you probably created them on the computer. In the late 1970s, your minicomputer might well have run Unix, ancestor to Linux, IOS, Android, and most modern, non-Windows operating systems. In a very real sense, our modern microcomputers (technically, computers built around microprocessors) are superpowerful, superfast minicomputers with graphical front ends.

By way of comparison, the minicomputer that served my entire undergraduate school was a VAX 11/750. It ran at 3.125Mhz, and I don't know how much memory it had, but certainly less than 20megabytes. Not gigabytes. Megabytes. Even its disk storage didn't number in gigabytes. Next to the Mac I'm writing this on is a Raspberry Pi. Its CPU runs at 900Mhz, and it has a full gigabyte of RAM, and the SD card it uses for storage is an 8 gibibyte device. Nevertheless, it runs Linux, and can execute exactly the same kinds

of programs that old VAX could. So can my Mac, orders of magnitude more powerful than the Pi, and so can my desktop Linux box. When microcomputers became powerful enough to run minicomputer software, the computer age we live in today was born, and all of that, all that we use today, came about because TTL ICs like the 7400 let you build logic circuits faster, easier, and cheaper than you could build them by hand. They're still used today, still made today, in fact. they're the "glue logic" that ties more complex ICs together.

So let's take a look at some TTL. If you don't have a 7400, or a 74LS00, or 74AHC00 or whatever, look the 74xx chips you do have up online and get their datasheets. You can use the lessons in this chapter to play with those chips, but you'll have to work out what the chip does and how to drive it with the Cestino for yourself.

This brings us to the topic of datasheets.

How to Read a Datasheet

We've already touched on datasheets in the other chapters, starting at the beginning with the 372-page monstrosity that is the ATmega1284P datasheet. Even with simple LED projects, the datasheet has important information, but as we move forward, they become critical. In today's marketplace, datasheets can be hard to come by. Some companies even require non-disclosure agreements to get them. Fortunately, the kinds of ICs we're dealing with are either hobbyist-friendly, with available datasheets, or old enough to be part of the era when a datasheet was considered part of the marketing effort for a given IC. The Texas Instruments 74xx00 datasheet is one of the latter.

The first page of the datasheet tells us what the Texas Instruments name for this IC is: the SN7400. It also tells us that it applies to the SN5400, which is the military spec version of the same IC, as well as the SN54L00 and SN54S00 variants. In the 7400 version, this datasheet covers the SN7400, the SN74S00, and the 74LS00.

Next, it tells us what packages the SN54x00 and SN74x00 is available in: mostly DIPs like the one we'll be looking at in this chapter, but also in the FK ceramic chip carrier, and the small outline PS package that only has two gates in an 8 pin DIP that I've never actually seen. Note especially that the pinouts differ between the SN5400 in the W package, lower left, and the package above it which includes the 7400 and 74LS00 version. All these things are important. It's not likely you'll fry the 7400 unless you get the Vcc and GND mixed up, but it won't work unless it's wired right.

Page 2 gives you ordering information and temperature tolerance information. If you're designing satellites or other extreme environment equipment, this matters to you. For us, tinkering as we are at room temperature, any old 74x00 should be fine. Tape, tube, and reel packaging refer to how they're shipped in bulk from the manufacturer. Useful if you're running an assembly plant. Not so much if you're tinkering with reused parts as we are.

Page 2 also shows the function, or truth table of the 74x00, and tells us the critical information that the 7400 is a *positive* logic chip. This means that for a pin to be on, or a logical 1, the voltage must be high. Negative logic chips exist. The distinction is important.

We've already seen page 3. If you're wondering why they tell us how to build the gates in their IC from scratch, they don't. Not really. What this is, is an electrically equivalent circuit, so that when we're designing other circuits to connect to the inputs and outputs of the 74x00, we can plan for their current, voltage, and resistance needs.

Page 4 contains the data we've been skimming for in our LED datasheets: the absolute maximum ratings. These are the highest and lowest values a given pin can have and still operate the IC within its rated capacity. They don't guarantee that the IC will fry if you exceed them, but they do promise that it shouldn't fry if you don't.

Under this are the recommended operating conditions. This section lists the supply voltage (Vcc) as between 4.75V and 5.25V for the 74xx00. You already know what that means. Note the high and low level input voltages. These refer to the voltages required to put the pin in a given logical state. For example, ((VsubIH)), calls out the logical high input level as two volts. Any signal above two volts will be considered logical high. Any signal below it may not be. Likewise, the ((VsubIL)) voltage is the voltage below which a signal will be considered logical low. It must be at or below 0.8v.

Below that are the current limits on the output pins. You've already seen this on the ATmega1284P's pins, It's a little confusing that the high level output current is -0.4mA. What this means is that the pin can source 0.4mA, or 400μA. It can sink 16mA to ground. By contrast, this is a lot lower than the 40mA the ATmega1284P can source or sink on each pin.

Below that and continuing onto page 4 are various operating parameters over temperature. The short version of all this is that the IC changes characteristics depending on its temperature. Also on page 5 is the switching speed. With a 400Ω, 15pF (Pico-Farad) load, the 74x00 described here will switch high in between 11 and 22 nS (nanoseconds) and low between 7 and 15 nS. How fast is that? Well, a 1MHz wave repeats every microsecond, and a nanosecond is 1/1000 of a microsecond, so if we divide 1000 by the switching speed in nanoseconds, we get MHz. 12 nanoseconds is about the middle of the value we can expect for switching both high and low, so 1000/12=about 83.3MHz. If we assume the worst case, 22nS, 1000/22=about 45.5MHz. Nothing we'll be doing in this chapter will approach those speeds, but I would not expect a 74x00 to go any faster than 45.5MHz.

When you start chaining logic together, it's tempting to think that things happen instantly. They don't. It takes (worst case) 22nS for each gate in this IC to switch states. That's not the only factor. Any time you have resistance and capacitance together, you have a time delay. A bad connection can go slower than you expect because the resistance goes up and by the way, a solderless breadboard can have significant capacitance too. Hopefully it's clear now why I said to keep the clock line as short as possible and as close as possible to both the TTL oscillator and the ATmega1284P. We're pushing the speed limits of breadboards a little.

Back on the datasheet, Note 4 is especially important. It says that all unused inputs must be held either to Vcc or GND to ensure proper operation. Using only one gate in the 74x00? Tie both inputs high or low, otherwise the outputs may go high or low, or may oscillate. Worst case, it can damage the IC.

The rest of the information on page 5 and on to page 6 repeats the same data we've had, but in different test scenarios—different operating temperatures, different loads, different versions of the IC, and so on.

Page 7 shows the schematics by which the test data was achieved, and timing diagrams. The important fact to note here is that while we think of TTL as being either on or off, there's time in between, where the voltage is rising or falling, and if you're designing a timing-critical circuit, we need to know that.

Don't worry. We're not. The circuits we'll be building in this chapter don't get out of the kilohertz range. For us, 22nS is instant enough.

The rest of the datasheet is ordering information, a list of all the varieties the IC comes in, (There's a space-qualified version, the SN54LS00-SP. I wonder if you can get those on EBay.) bulk packaging information, exact mechanical drawings of the IC in various packages, thermal information, so you don't fry ICs by overheating them while wave-soldering them to boards, and a big, fat disclaimer that says not to use these ICs in life support equipment without a special agreement with TI, and so forth and so on.

Datasheets can be dense. They're written by electronic engineers for electronic engineers, and as they promise specific performance of a device in specific circumstances, they're quasi-legal documents as well. Some are better than others. TI's, in addition to having friendly licensing terms, are also well written. Some, particularly for low-cost components originating in non-English speaking countries, can be truly challenging to decipher. Fortunately, the 74xx series TTL family has been around for decades, and versions are made by a number of corporations around the world. If you find the Fairchild Semiconductor datasheet incomprehensible, you can get the datasheet for the TI version. The performance of a given IC (say, a 74LS00) will be similar enough across versions to keep you out of trouble. By contrast, if you find the ATmega1284P's datasheet a tough read (it's not, it's just big) you're stuck with it, as only Atmel makes the ATmega line of microcontrollers.

How to Read Your IC

There's a wealth of information in the markings on an IC. If you're over a certain age, you may need a magnifying glass or very good light (or both) to read it, but an IC should always have its number, such as 74LS00. There may be letters and numbers before or after the main IC number, which tell you things like what kind of package the IC is in, whether it's got special ratings or capabilities, and so on. These extra numbers and letters vary by manufacturer, so if you need extra capabilities, it's important to find the manufacturer's trademark or name on the IC and get their specific datasheet for it. There's an excellent website for identifying these often cryptic, often outdated marks here:

`http://how-to.wikia.com/wiki/How_to_identify_integrated_circuit_(chip)_manufacturers_by_their_logos`

Often, but not always, there is a date code, and this usually takes the form of YYWW, the last two digits of the year, followed by the two-digit week number of that year. So 1412, the date stamp on one of my ATmega1284Ps, would mean it was made in the twelfth week—the middle of March—2014. Not all ICs have date codes. Some may have date codes relative to an arbitrary event, such as a ruler's coronation.

Finally, there's a notch at one end, just like there was with the ATmega1284P. That tells you which end pin 1 is on. If there's not a notch, pin 1 will have a spot in the top of the package over it, or perhaps a paint mark, or some other clear indication that pin 1 is *here*. This is important. TTLs really don't like having their power backward.

Build the TTL Explorer

This project is going to be a little different from most of the others. I don't know what 7400 series logic you have in your junk box. We'll be working out of mine, and I'll talk about specific strategies for poking at each IC, with respect to what it can do, and showing output. If your logic ICs are different, you'll have to work up your own sketches, but hopefully you'll find my examples helpful.

74xx00

Let's start with the 74xx00 quad two-input NAND, from which the whole family takes its name. As you might be able to see in Figure 6-2 below, mine's a 74LS00, made in the 48th week of 1981 by Fairchild in Singapore. At that time, Fairchild Camera and Instrument (now known as Fairchild Semiconductor) was owned by Schlumberger Limited, an oilfield management company. Go figure. We'll go ahead and use the Texas Instruments datasheet. If we were pushing toward the specification limits of the 74LS00 in any way, we'd want to work from the Fairchild sheet, but we're not, and the TI datasheet is nicer.

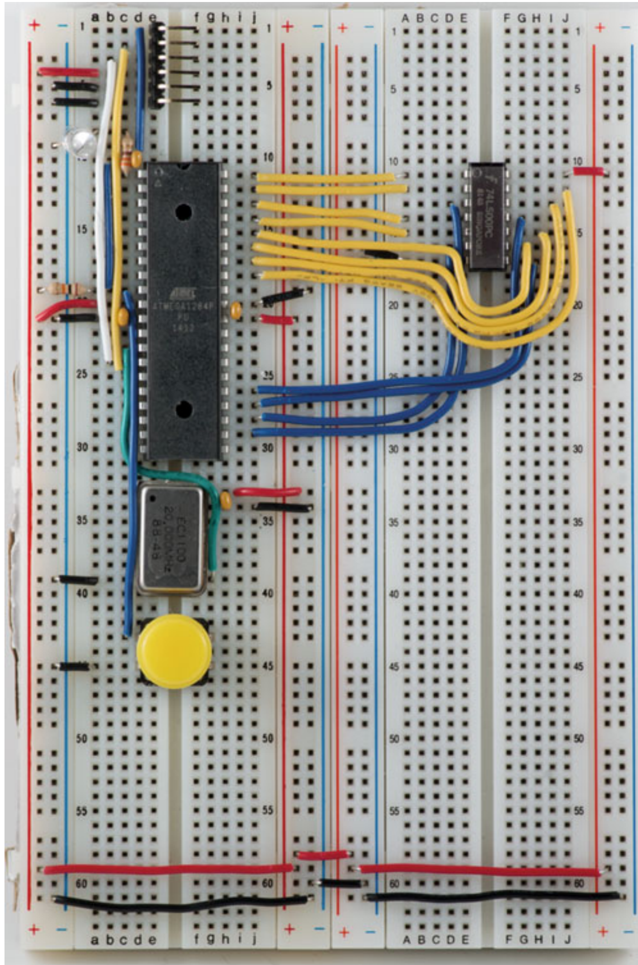

Figure 6-2. *74xx00 TTL Explorer*

Suppose that you wanted to wire the ATmega1284P to allow hardware resets only when the software is ready for one? Instead of tying the reset button to the - bus, and to the reset pin on the ATmega, you could tie it to the + bus, and connect it to one input of a NAND gate in a 74LS00. You could then tie the other input of the NAND gate to pin 1, aka pin B0, where the LED is connected, and tie the output to the reset line on the ATmega. The result would be that when pin B0 is off, pushing the reset button does nothing. The output stays high. Turning pin B0 on also does nothing. The output stays high. Only if you press the button AND pin B0 is high does the output go low and reset the ATmega. This is the kind of job you do with a 7400. Simple jobs. Little jobs. It does, after all, take two of them to output a complete byte.

Nevertheless, let's hook this one up and throw bit patterns at it, and watch it actually do what the datasheet says it should do.

Figure 6-3 shows how to wire up your 74xx00, hereafter referred to as the LS00. Note that the pins in the schematic are out of order so the logic gates can be in order. When you're wiring your 74xx00 up, make sure you work from the right pinout diagram in the datasheet. It's the one on the top left of the first page of the datasheet. Make sure the pin 1 notch or spot is at the end of the LS00 closest to the edge of the breadboard, the same as the ATmega1284P's is.

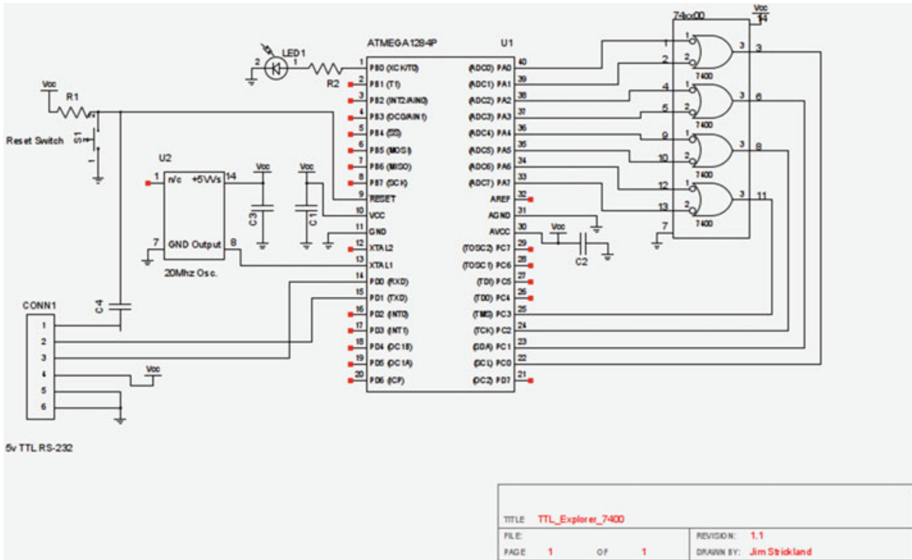

Figure 6-3. *74xx00 TTL Explorer Schematic*

Wire pin A0 (pin 40 on the Atmega1284P) to pin 1 of the LS00. A1 goes to pin 2 of the LS00. These two outputs are connected to the inputs of the first NAND gate on the LS00. Gate 1's output is on pin 3 of the LS00, so wire that to pin C0 (pin 22) of the ATmega. The next gate is the same way. Wire A2 and A3 to pins 4 and 5 of the LS00, and pin C1 to pin 6.

For reasons known only to the original designers, the other side of the LS00 is essentially backward from this one. Wire pin 8 to C2 on the Cestino. It's the output of Gate 3. Then wire A4 to pin 9 and A5 to pin 10 to control the gate. Likewise, pin 11 of the LS00 is the output of Gate 4. Wire it to pin C3 on the ATmega, and wire pins 12 and 13 to A6 and A7, respectively.

Don't forget power and ground. Power (Vcc) is on pin 14 of the LS00. Tie this to the + bus. Ground (GND) is on pin 7. Wire this to the - bus.

That's all there is to it.

The code for this project is about as simple as it comes. If it looks like I derived it from the Binary Numbers on Display code in chapter 4, again, you've got a good eye. Cycling through all the values between 0 and 256 is a good way to get al*l* the possible permutations of inputs for a device.

```
#define DELAYTIME 10
```

The first thing we do, once we wade through all my comments, is set DELAYTIME to 10ms. Skimming through the datasheet, you may recall that this IC is specced to respond in less than 22ns (nanoseconds). Why a delay of more than 10,000 times that? It makes it easier for humans to read.

Next comes a function. It's not strictly necessary for this sketch, and if you don't want to use it, you don't have to, but I find it annoying to read binary output where the length of the variable changes constantly. Particularly when you're comparing bit fields, it's very hard (at least for me) to keep straight what bit I'm looking at, when I may have 10 on one line and 10000001 on the next. This function returns a String object which contains the full representation of all the bits passed to it in the byte, and prepends a 0b, which by now we're all getting used to as the way the GCC compiler represents binary numbers. Let's dig into how it works.

```
String zerobee(byte input) {
  String temp = "";
```

The first thing we do after declaring the function is to initialize the string we're building to an empty string. This is very, very important, because we never explicitly set the string. We only add characters to it. If it contains garbage to begin with, nothing in this code will change that. Initializing it makes sure that it doesn't.

```
for (int c = 0; c <= 7; c++) {
    if (input % 2) {
      temp = "1" + temp;
    } else {
      temp = "0" + temp;
    }
    input = input / 2;
  }
```

This code is where the work gets done. It iterates from 0 to 7. Each time, it checks to see if the input byte divided by 2 has a modulus, or a remainder, using the modulus operator. By now, my habit of commingling integer values of zero as false and not zero as true is probably very familiar. I'm doing it here, too. If there is a modulus, we prepend the character "1" to temp. If there isn't, we prepend a "0". Then and only then do we actually divide the input by 2, and the loop iterates again.

```
  temp = "0b" + temp;
  return (temp);
}
```

Once we exit from the loop, we prepend "0b" onto temp, and return the String to the calling function.

Serious microcontroller programmers are screaming right now. Do I know how many cycles that much division chews up? Do I know how *inefficient* that is?

Yes. I do know. Division is very expensive, and we're doing sixteen of them every time we call the function. But the truth is, this is not a high speed, memory constrained environment. The huge capacity of the ATmega1284P (compared to the ATmega328 in most 5v Arduinos) gives us the luxury of wasting a few cycles and some flash space in order to make things easier to read. So that's what we're doing.

Onward.

```
void setup() {
  Serial.begin(115200);
  DDRA = 0b11111111;
  DDRC = 0b11110000;
  PORTA = 0;
  PORTC = 0b00001111;
  Serial.println("Starting...");
}
```

This code is, by now, pretty familiar. It sets the console to 115200 baud, sets port A up to write with all pins, and sets port C up to read with its first four pins. Why not use all 8 pins? We're displaying the full width of PINC, and if we leave those pins unconnected, their values are unknown. We'll mask the high order bits out when we print them later.

We initialize PORTA to 0, and set the internal pull-up resistors on the lowest four pins of PORTC.

Why pull-up resistors? It's always nice to have your inputs in a known state, in this case on. Pull-up resistors are unnecessary on a totem pole output like the ones on this 74LS00, but they don't hurt anything. (A totem-pole output is a push-pull amplifier circuit with a PNP and an NPN transistor forming a voltage divider, and the output between them.)

The loop function is where the action is, and there isn't much to it.

```
void loop() {
  Serial.print("Port A:"+zerobee(PORTA));
  Serial.println("\tPORTC:"+zerobee(PINC));
  PORTA++;
  delay(DELAYTIME);

  if (PORTA == 0) {
    while (true) {}
  }
}
```

Remember that the loop() function, shown here, is called over and over by the Arduino core, so just like we did with the binary numbers on display sketch, we increment the port's value until it hits zero. Yes, it is initialized at zero. Look where the PORTA++ is. It will never, ever reach the test where we check to see if PORTA is 0 with the initialized

value. The only way it will ever pass that test is when the iteration begins with PORTA at 255 and it increments it. As a single byte value, PORTA rolls over to zero, passes the test, and the while(true){} loop runs forever, doing nothing.

So what does the output of this sketch look like? Well, if you have your own 74xx00, it's more fun to run the sketch. If you don't, here's the trimmed version of my output.

```
Starting...
Port A:0b00000001        PORTC:0b00001111
Port A:0b00000010        PORTC:0b00001111
Port A:0b00000011        PORTC:0b00001110
```

Notice what just happened. Pins A0 and A1 are both 1, so gate 1 of the LS00 goes low, outputting zero. Our input value is 3. This pattern repeats further down.

```
Port A:0b00001011        PORTC:0b00001110
Port A:0b00001100        PORTC:0b00001101
Port A:0b00001101        PORTC:0b00001101
```

You can see gate 2 going low when PORTA hits 12. Notice how it stays low through the entire subsequent range of numbers as long as bits 8 and 4 are turned on.

```
Port A:0b00110000        PORTC:0b00001011
Port A:0b00110001        PORTC:0b00001011
```

Gate 3 is rolling over as we hit 48.

```
Port A:0b10111111        PORTC:0b00001000
Port A:0b11000000        PORTC:0b00000111
Port A:0b11000001        PORTC:0b00000111
```

As expected, gate 4 goes low when we reach 192.

```
Port A:0b11111111        PORTC:0b00000000
```

And here, at 255, all the gates finally go low at the same time.

Here's the full listing of the sketch, including comments.

```
//TTL_Explorer_74xx00
//------------------------------------------------------------
//This sketch counts on PORTA from 0 to 255, sending those
//values to a 74xx00 connected as described below. It then
//reads the 74xx00's output pins with the first four bits
//of PORTC. Both values are displayed.
// -----------------------------------------------------------
//Hardware:
//74xx00, any flavor (mine's an LS)
//
// Wiring:
```

```
// Outputs are wired to PORTC, in order, lowest to highest.
// Inputs are wired to PORTA, in order, lowest to highest.
//Cestino Pin   74xx00  Cestino Pin
//(Pin 40) A0   1   14  Vcc (+ Bus)
//          A1   2   13  A7
//          C0   3   12  A6
//          A2   4   11  C3
//          A3   5   10  A5
//          C1   6   9   A4
// (- Bus) GND  7   8   C2

// ----------------------------------------------------------
// James R. Strickland
// ----------------------------------------------------------
//precompiler definitions.
#define DELAYTIME 10 //How many mS (milliseconds) per update?

//-----------------------------------------------------------
//Zerobee
//-----------------------------------------------------------
//This function returns a string with the binary equivilent of
//the value passed to it in input expressed in 0bxxxxxxxx
//notation, with 8 bits for each byte regardless of their value.
//IMHO this makes binary values easier to read and compare.
//-----------------------------------------------------------
String zerobee(byte input) {
  String temp = "";
  for (int c = 0; c <= 7; c++) {
    if (input % 2) {
      temp = "1" + temp;
    } else {
      temp = "0" + temp;
    }
    input = input / 2;
  }
  temp = "0b" + temp;
  return (temp);
}
```

```
//setup() function - runs only once.
void setup() {
  Serial.begin(115200);
  DDRA = 0b11111111; //Set DDRA to all outputs.
  DDRC = 0b11110000; //Set DDRC so pins C0-C3 are inputs.
  PORTA = 0;     //Initalize PORTA to 1.
  PORTC = 0b00001111; //Set the pullups on bits C0-C3
  Serial.println("Starting...");
}

//loop() function - runs forever.
void loop() {
  Serial.print("Port A:"+zerobee(PORTA));
  Serial.println("\tPORTC:"+zerobee(PINC));
  PORTA++;
  delay(DELAYTIME); //Wait DELAYTIME milliseconds.

  if (PORTA == 0) { //If PORTA reaches zero, loop forever.
    while (true) {}
  } //otherwise let loop() execute again. And again. And again.
}
```

74xx92

The next IC I'm going to demonstrate is the 74xx92, in this case a 74LS92N. As you might be able to see in Figure 6-4, mine was made by Sygnetics, part of Phillips Semiconductor, now known as NXP, apparently manufactured in January of 1978. Where I got it, I have no earthly idea. It doesn't appear to be unsoldered from anything, so it was either out of some old junk where it was socketed, or it was a leftover from another project and sat on a distributor's shelf for a few decades.

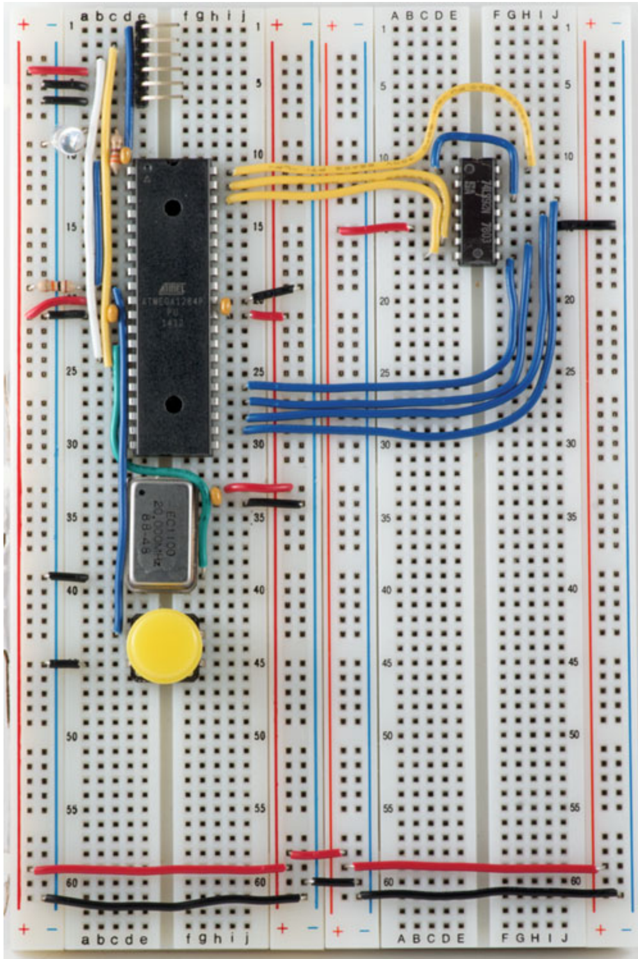

Figure 6-4. *74xx92 TTL Explorer*

The 74LS92N is a counter, or more precisely a divide by 12 device. It does not produce normal BCD (binary coded decimal) values, because the most significant bit is a 6 bit, not an 8 bit.

Looking at the datasheet for this IC (Futurelec has the Motorola version here: `http://www.futurlec.com/74LS/74LS92.shtml`) the first thing to note is that this sheet covers three different counters, the 74LS90, 92, and 93, which are a decade counter (divide by 10), divide by 12 counter, and a 4 bit binary counter, respectively. If you have the 90 or the 93, you can still use this code and mostly the same wiring, but your output will be different.

If you scroll down to the truth table for the LS92, you can see that, from a count of 0 to 5, it produces normal BCD values on output pins Q0 through Q3. Once the count reaches 6, however, everything becomes weird. If it were outputting BCD values, it would count from 0 to 5, skip 6 and 7, and continue at 8.

You should also note that the divide by 2 counter (which outputs to Q0) is not internally connected to the divide by 6 counter. You have to wire them together externally. This is so that you can use the counters separately. If, for example, you wanted to take our 20MHz clock and get a 10MHz clock from it, you could connect pin /CLK0 to the clock. According to the absolute maximum ratings, the 74LS92 can handle signals up to 32MHz on the /CLK0 pin, so we're good there. If we connect the Q0 output to the /CLK1 input, we feed the frequency divided by two to the divide by six counter. If we wanted 10MHz, we could pick it off the Q0 output. If we wanted roughly 1.67MHz, we could pick it off the Q4 output—20MHz/12. And so on. Note that /CLK0 and /CP0 are different notations for the same pin. /CLK1 and /CP1 are the same as well.

But enough talk. Let's wire it up and get it counting.

As you can see from the schematic in Figure 6-5, we'll use PORTA to control the 74LS92, and PORTC to read it, mostly because they're conveniently located. Note that the pins on the 74LS92 are not shown in order once again, and that the connection between pin 12 and pin 1 does not also connect to pin 14.

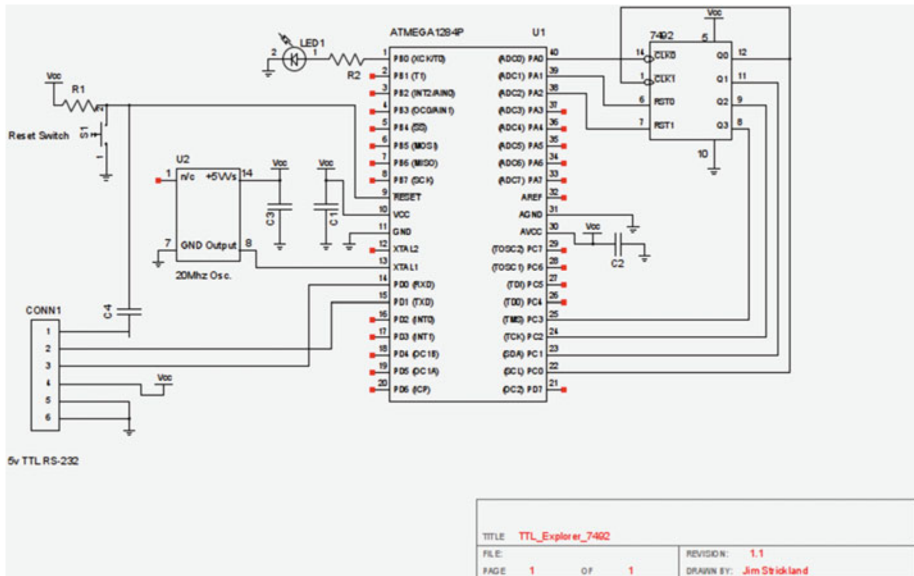

Figure 6-5. *74LS92 TTL Explorer*

Wire pin A0 (pin 40) to pin 14 of the 74LS92 (hereafter referred to as it is in the datasheet, the LS92.) Pin A1 goes to pin 6 on the LS92, and pin A2 goes to pin 7 on the LS92. These lines are the clock or input line, and the reset lines, respectively. These lines control the LS92, but they don't read its output.

To read the output, wire pin C0 (that's pin 22 on the ATmega1284P) to pin 12 on the LS92, C1 to pin 11, C2 to pin 9, and C3 (pin 25 on the ATmega1284P) to pin 8 on the LS92. Then on the LS92, wire pin 12 to pin 1. Finally, on the LS92, wire pin 5 to the + bus, and pin 10 to the GND bus. TTL isn't magic. It needs power too.

The code itself is fairly straightforward. We start by defining delaytime at 100ms.

```
#define DELAYTIME 100
```

Our setup, too, is basically boilerplate. We set the console to 115200 baud, set DDRA so that PORTA is writing with all pins, set DDRC so PORTC is reading with all pins. We also set the pull-up resistors on the first 4 pins of PORTC.

```
void setup() {
  Serial.begin(115200);
  DDRA = 0b11111111;
  DDRC = 0b00000000;
  PORTC = 0b00001111;
  Serial.println("Starting...");
}
```

In the loop function, we begin by setting PORTA so that the reset pins of the LS92 are both high, resetting it. Then hold them both low, getting the LS92 ready to count. We print that so the user can see it, using the zerobee function (not shown—it's the same as in the 7400 demo.) Every time we call PINC, we AND it with 0b1111, that is, 0b00001111. Because pins 5-8 of PORTC are unconnected, we don't want to look at their output.

```
void loop() {
  PORTA = 0b00000110;
  Serial.println("Holding reset pins high. PORTA:" + zerobee(PORTA));
  Serial.println("PIN C should be all zeros. PINC:" + zerobee(0b1111 & PINC) + "\t");
  PORTA = 0b00000000;
  Serial.println("\nReset lines now low. PORTA:" + zerobee(PORTA));
  Serial.println("PIN C should still be 0. PINC:" + zerobee(0b1111 & PINC) + "\t");
```

After that, we run a loop with 24 steps. Each cycle delays DELAYTIME ms, so we can read the output, and on negative steps (as ~CP0 is active low) we display the value of PINC, which is reading the output pins of the LS92.

```
Serial.println("\nGiving the ~CP0 pin 12 negative pulses");
  for (int c = 1; c <= 24; c++) {
    PORTA = (PORTA + 1 & 0b00000001);
    if (PORTA == 0) {
```

```
      Serial.print("Pulse " + (String) (c / 2) );
      Serial.println("\t PIN C: " + zerobee(0b1111 & PINC));
    }
    delay(DELAYTIME);
  }
```

After that, we loop forever so the whole thing doesn't run again. As before, it's more fun to watch the patterns emerge in your own console window. If you don't have a 74xx92 of your own, however, you can follow along in my output. It's here in its entirety, as it's short.

```
Starting...
Holding both reset pins high. PORTA:0b00000110
PIN C should be all zeros. PINC:0b00000000

Reset lines now low. PORTA:0b00000000
PIN C should still be 0. PINC:0b00000000
```

In the previous section, we reset the 74LS92. Remember the use case we talked about in the 74xx00, where I talked about having a software-enabled reset for the ATmega? That's pretty much what the reset lines for the LS92 do. Why, you ask? Sometimes you don't want the counter to go all the way to 12. You could wire those two inputs to any of outputs of the LS92 and chose any value with two bits turned on to reset at. In our case we're controlling them with the ATmega.

Next, we pulse Cestino pin A0 12 times to give the counter some pulses to count. Remember that the CP0 pin on the LS92 is active low, so the counter is counting negative pulses.

```
Giving the CP0 pin 12 negative pulses

Pulse 1         PIN C: 0b00000001
Pulse 2         PIN C: 0b00000010
Pulse 3         PIN C: 0b00000011
Pulse 4         PIN C: 0b00000100
Pulse 5         PIN C: 0b00000101
```

Everything is normal through this point, but look what happens below. What should be our 8 pin (bit 4) is really representing 6, so it turns on at pulse 6. This is not normal binary. It's very important to keep that in mind when reading the output.

```
Pulse 6         PIN C: 0b00001000
Pulse 7         PIN C: 0b00001001
Pulse 8         PIN C: 0b00001010
Pulse 9         PIN C: 0b00001011
Pulse 10        PIN C: 0b00001100
Pulse 11        PIN C: 0b00001101
```

Once again, you can see that bit 4 represents 6 and not 8, and makes our output some strange reading. The timer rolls over at 12.

```
Pulse 12        PIN C: 0b00000000
```

Here's the sketch in its entirety.

```
//TTL_Explorer_74xx92
//-----------------------------------------------------------
//This sketch raises the reset lines of the 74xx92 to reset
//it, then holds them low. It then generates 12 negative
//pulses on CP0 (pin 14 of the xx92), delaying DELAYTIME ms
//in both the on and off states. In off states, it reads
//PINC for the results, formats them with zerobee binary
//pretty printer, and sends them up to the console.
// ----------------------------------------------------------
//Hardware:
//74xx92, any flavor (mine's an LS)
// PORTA0 (pin 40) to pin14
// PORTA1 to pin6
// PORTA2 to pin7
// PORTC0 to pin12
// PORTC1 to pin 11
// PORTC2 to pin 10
// PORTC3 to pin 9
// Pin12 to Pin1
// Pin 5 to + bus
// Pin 10 to - bus
// Note that pin Q3 is outputting /6/ and not /8/
// ----------------------------------------------------------
// James R. Strickland
// ----------------------------------------------------------

//precompiler definitions.
#define DELAYTIME 100

//-----------------------------------------------------------
//Zerobee
//-----------------------------------------------------------
//This function returns a string with the binary equivilent of
//the value passed to it in input expressed in 0bxxxxxxxx
//notation, with 8 bits for each byte regardless of their value.
//IMHO this makes binary values easier to read and compare.
//-----------------------------------------------------------
```

```
String zerobee(byte input) {
  String temp = "";
  for (int c = 0; c <= 7; c++) {
    if (input % 2) {
      temp = "1" + temp;
    } else {
      temp = "0" + temp;
    }
    input = input / 2;
  }
  temp = "0b" + temp;
  return (temp);
}

//setup() function - runs only once.
void setup() {
  Serial.begin(115200);
  DDRA = 0b11111111; //Set DDRA to all outputs.
  DDRC = 0b00000000; //Set DDRC to all inputs.
  PORTC = 0b00001111; //Turn C0 to C4 pullups on
  Serial.println("Starting...");
}

//loop() function - runs forever.
void loop() {
 PORTA = 0b00000110;
 Serial.println("Holding both reset pins high. PORTA:" + zerobee(PORTA));
 Serial.println("PIN C should be all zeros. PINC:" + zerobee(0b1111 & PINC)
 + "\t");
  PORTA = 0b00000000;
  Serial.println("\nReset lines now low. PORTA:" + zerobee(PORTA));
  Serial.println("PIN C should still be 0. PINC:" + zerobee(0b1111 & PINC) + "\t");
  Serial.println("\nGiving the CP0 pin 12 negative pulses");
  for (int c = 1; c <= 24; c++) {
    PORTA = (PORTA + 1 & 0b00000001);
    if (PORTA == 0) {
      Serial.print("Pulse " + (String) (c / 2) );
      Serial.println("\t PIN C: " + zerobee(0b1111 & PINC));
    }
    delay(DELAYTIME); //Delay DELAYTIME ms for each on or off.
  }

  while (true) {} //Loop forever so we don't run again.
}
```

Credit Where Credit's Due

A long time ago now, in a musty basement closet-turned-lab that belonged to the physics department of Concordia College, I needed a device that would take two pulsed inputs, count them, and display the count on seven-segment LEDs. Sounds like a job for 7400 series TTL? You bet. It was then that I had my first lessons in 7400 series logic, from a hardcore nerd and genius I'll call Gene (it was his name). That's where it started. I didn't get very far, and I got distracted by other things (graduation, dating, etc) but that's where it started. I should mention that the source of the pulses was a rat dropping a marble between an LED and a phototransistor, by way of a somewhat makeshift basket. There were two of these circuits. It was a scoreboard for rat basketball. Seriously. It's a long story, but it involved four 7 segment drivers, two 74LS393s, and some other parts I no longer remember.

Ultimately I had to pick up TTL again for myself, and the inspiration, when the time was finally right, was this guy, madmaxx, on youtube: `https://www.youtube.com/watch?v=bCVT1BtlZn0`, followed shortly by Quinn Dunki's Blondihacks blog, here: `http://quinndunki.com/blondihacks/`, and EEVblog, here: `https://www.youtube.com/user/EEVblog`. You'll see Blondihacks again in the credit where credit's due section of another chapter.

Further—More Chips, More Pins, Automatic Configuration

So where could this project go? It's conceivable that with a very large sketch, by hooking up *all* the pins of the 74xx00 series IC to a Cestino port, even power and ground, you could build a tester that would test a very large number of different 74xx00 series ICs.

This isn't as pointless as it sounds. A couple of years ago, I built my second computer from ICs and bare boards, a PC clone. It was starting to show functionality when I got a piece of steel wool on my work bench across the power supply lines and (presumably) shorted the 12v supply to the 5v rail. In addition to nearly starting a fire, this wiped out a bunch of the logic on the board. Being able to test the ICs instead of shotgun replacing them all (I have another device that could test most of them) saved me quite a lot of money.

As for going further with TTL, the sky is more or less the limit, so long as you're not in too much of a hurry. People can and do still make up their own CPUs from scratch, starting with nothing but TTL and a plan. There's a whole web-ring of people who've done it (and more—some have built CPUs up from discrete components like individual transistors.) The ring home is here: `http://members.iinet.net.au/~daveb/simplex/ringhome.html`.

Dr. Harry Porter (and others) have even done it with relays. `http://web.cecs.pdx.edu/~harry/Relay/` (If you speed the sound of Dr. Porter's computer running up in your mind, it's easy to imagine it as the sound of the computer in the original *Star Trek*. I always wonder if there's a story there, and what that original sound effect was a recording of.)

CHAPTER 7

Logic Probe

One thing that is often hard to get your head around when you first start out with electronics is the mind-boggling speed at which things happen. The Cestino is, by modern standards, not especially fast with its 20MHz clock, or 50 nanoseconds per clock cycle. Nanoseconds are tiny. On the 50ns scale, humans move and think with the speed of plate-tectonics. It's a time frame that is simply below our perception without tools.

The Need for Speed

Computers, and the logic from which they are made, are at home in nanosecond scale timeframes. Their great strength has always been the ability to execute our instructions, painfully put together in seconds and minutes, thousands or millions of times a second. (Remember, not all computers share the ATmega's RISC nature and execute an instruction per clock cycle. Also, many actions take multiple instructions.) We haven't really done anything at significant speed yet, and in fact in Chapter 6, we added hundred-millisecond delays just so we could see what the Cestino was up to. Also, the act of transferring data to our host computers adds delays.

The chapters that follow this one will involve much more complex circuits. We'll be using multiple ports, reading and writing data at fairly brisk rates, generating clock signals, and so on. All these things, if we want to debug them effectively, require us to perceive logic states faster than we can see, and preferably without tying up the ATmega's resources as a debugger. Oscilloscopes and logic analyzers have gotten much less expensive in recent years, but they're still hundreds of dollars (new) for tools worth having, and they require a lot of expertise that's really beyond the scope of this book to teach.

Back toward the beginning of the microcomputer revolution, however, there was another tool that did the same kinds of job, albeit one logic circuit at a time. They were called logic probes. We're going to build a very, very simple one, shown in Figure 7-1, and try it out. Like the Cestino itself, we'll leave this circuit assembled on the breadboard, so we can use it to debug the rest of our projects moving forward.

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_7

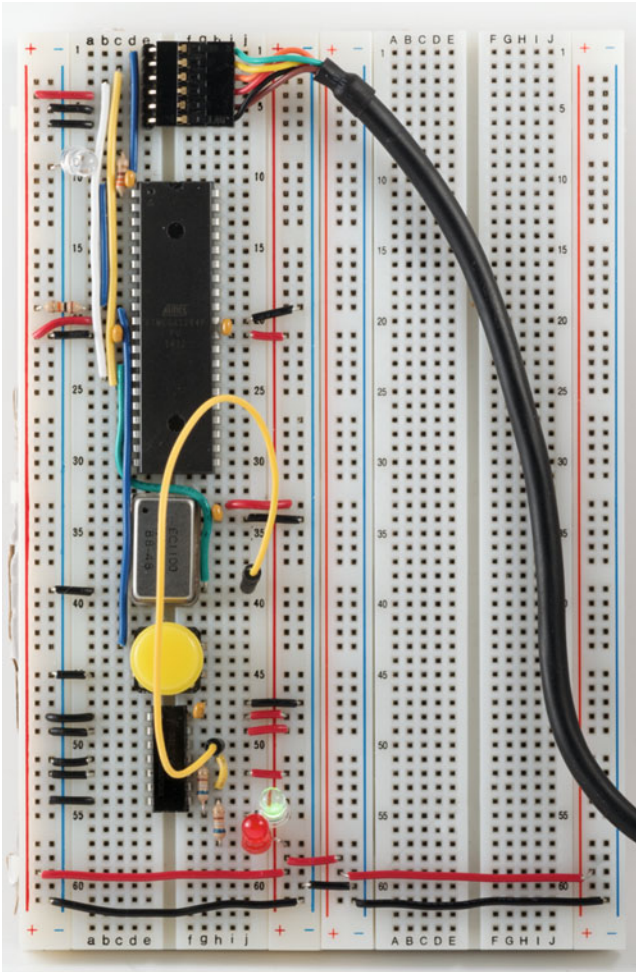

Figure 7-1. *Assembled Logic Probe*

The Stuff You Need

The logic probe we're going to build is rudimentary, so it needs only a few components.

New Parts

2 680Ω resistors

1 0.1µF capacitor

New or Used Parts

1 74xx00 IC, where xx is F, LS, HCT, or any other type of 5v 7400 capable of sinking at least 4mA, and whose inputs go high around 2v. I used the 74LS00 from the previous chapter, arguably the worst choice due to its very low sink current, but it works well.

2 LEDs, whatever color you like. Different colors would be ideal, so you can tell them apart at a glance. High output isn't a good idea, due to the limited current available.

1 long jumper, or a piece of hookup wire 4-6 inches long with two bare ends.

Design the Logic Probe

The simplest possible logic probe, albeit the one with the most limitations, is a pair of LEDs and dropping resistors, one with its anode on the + bus, one with its cathode on the - bus, and the free leads tied together. When connected to a logic circuit, if the circuit is high, the LED with its cathode on the - bus will light. If the circuit is low, the LED with its anode on the + bus, and if the circuit is pulsing, the two will switch back and forth down to a few tens of milliseconds, and both appear to be on continuously below that.

This circuit has some drawbacks. Ideally, a piece of test equipment should not affect the circuit under test enough to alter the results of the test. If you connect the simple LED logic probe to a current where the logic output is loaded to its limit, the current drawn by the LED and its dropping resistor may well cause that output to appear off, or low, when it's not.

Our circuit, shown in Figure 7-2, will use one gate of a 74xx00 (a 74LS00, in my case) as a buffer, and a second as an inverter.

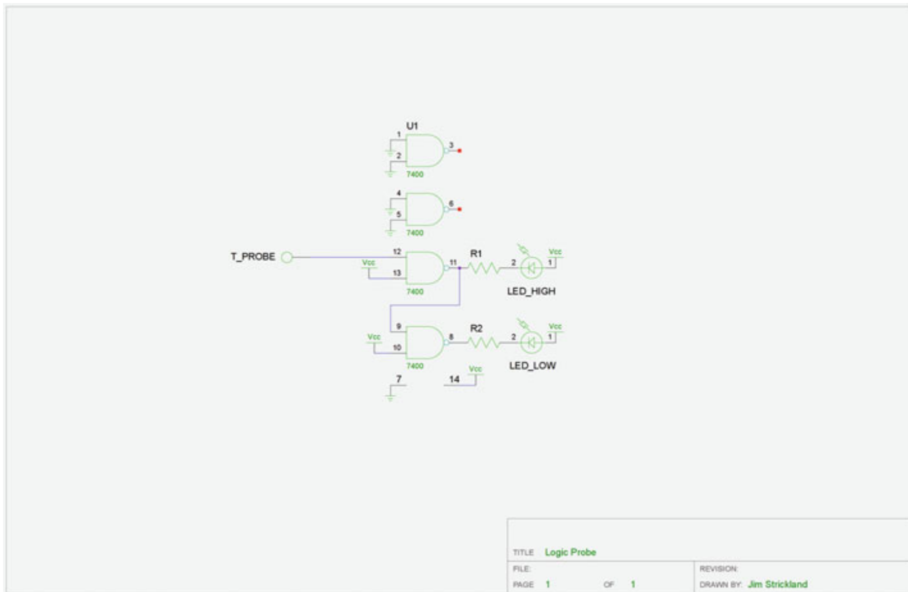

Figure 7-2. *Logic Probe Schematic*

Because the 74xx00 is a quad-NAND gate, recall that its output only goes low if both inputs are high. Also, if you review the TI 74xx00 datasheet, the original 7400 is rated to source only 4mA, but sink 16, and the 74LS00 part I'm using can can source 4mA but can only sink 8mA. It makes far more sense to use the 74xx00 to switch the cathode of the LED to ground than the anode to +5v. This effectively turns the NAND gate into an AND gate, as we invert the NOT output with our LED. You can see in the schematic in Figure 7-2 that we have two gates wired this way, from the + bus to the anode of the LEDs, to dropping resistors (about 680Ω in my case, for a current of nearly 4mA. I don't know where these LEDs came from or what their real characteristics are, so it pays to be conservative with the current.)

The gate driving LED_HIGH has one input tied to the + bus, and one input tied to the T_PROBE, the terminal where the probe is connected. In our case, a piece of wire connected to pin 12 of the 74LS00. The other input of this gate is tied to the + bus. When the T_PROBE input goes high, the gate output will go low, which switches the LED on.

The gate driving LED_LOW is different. It has one input tied to the output of the gate driving LED_HIGH, and the other input tied to the + bus. This means that its output will be low any time the output of the LED_HIGH gate is high, that is, LED_HIGH is not lit. So LED_LOW will be lit only when T_PROBE is connected to a low circuit.

Our logic probe may be operating at moderately high frequencies, so we need to think about timing as well.

One gate of a 7400 is rated to transition states in 22ns (nanoseconds), and the 74LS00 I'm using is rated to do it in 15ns. This translates to maximum frequencies of 45MHz and 65MHz respectively. How do I get that number? Easy. The reciprocal of transition time in seconds is the frequency in Hertz. So 1/0.000000022 seconds (22ns) is 45454545Hz, or about 45MHz. Likewise 1/65000000Hz (65MHz) is 0.000000015384seconds, which rounds to 15ns. I've found that the more I do these calculations, the more basic values wind up memorizing, like multiplication tables. One millesecond (1000ns) = 1 megahertz, 50ns=20MHz, 20ns=50MHz, and 1ns=1GHz, or 1 gigahertz.

Speed is important, and we clearly don't have unlimited speed with the 7400 or even the more advanced 74LS00 that I'm using. Worse, we're stacking two gates together. We could look closely at the transition curves and calculate exactly when the second gate begins its transition from on to off, thus turning the LED on, but our timing isn't that critical. We can assume that all the transitions will be complete in twice the rated transition time, so we shouldn't expect our probe to go faster than about 22.5MHz for the 7400 and 32.5MHz for the 74LS00. This is plenty fast for our projects. The LEDs can't respond nearly that fast, and even if they could, our eyes can't, and even if they could, there are no signals on our breadboard faster than the 20MHz TTL oscillator we're using for the Cestino's clock.

With the timing sorted out, we can talk about input current, which is the whole reason for using the 7400 as a buffer (and an inverter). We don't want our logic probe to affect the logic circuit we're testing.

The 74LS00's inputs draw 20μA (micro-amperes) high, and 0.4mA (mili-amperes) low, giving them a minimum impedance of about 12,500Ω. (The plain 7400's inputs draw more current: 40μA high, and 1.6mA low, for a minimum impedance of about 3kΩ. Which is still better than the LEDs.) Given that all the circuits we're going to be setting up are TTL compatible, we know that they can source and sink currents in the low numbers of milliamperes at least, and we know not to load their outputs to the limit, so the 7400's impedance is enough.

■ **Note** As always, we're treating impedance and resistance as the same thing. They're not. Impedance is resistance at a given frequency. The reactance (AC or pulse resistance of capacitors and inductors) changes based on frequency. For our circuits and the speeds at which we're working we don't have to be too fussy about impedance matching, but keep in the back of your mind that at some point, as your circuits get faster, you will.

Don't have a 74xx00? You can build this logic probe with just about any gate TTL IC. You'll have to figure out the logic, and the output currents, and the dropping resistors for your LEDs, and your input impedance for yourself, but you've got those skills at this point. It's a fun little project to design.

Build the Logic Probe

Having designed the logic probe, let's go ahead and build it. As you can see in Figure 7-1, I built mine below the reset button of my Cestino. That way I can leave it assembled and it's not in the way of the circuits to come, but it is powered and ready to use for debugging them.

Assuming you're using a 7400 or a 74LS00, wire pins 1, 2, 4, 5, and 7 to the - bus. Pin 7 is the 74xx00's ground pin, and 1, 2, 4, and 5 are the inputs of gates we're not using. According to the datasheet, any unused inputs of the 74xx00 will drift high and may oscillate, causing problems for the other gates. Any unused gates should have their inputs grounded or held high. We're grounding them.

Wire pins 10, 13, and 14 to the + bus.

Because we're leaving this assembled, wire the .01µF capacitor from pin 14 to the socket row above it, toward the reset button, and from there to the - bus.

Connect resistor R2 to the row containing pin 8, and leave an empty socket between it and pin 8. Plug the other end into the breadboard on the third socket straight down from there, then plug the cathode of LED_LOW into that row, and the anode of LED_LOW into the + bus.

Connect resistor R1 into the row containing pin 11 of the 74xx00, and again orient it straight down, right beside R2. Plug the other end of R1 into the breadboard socket just below pin 8. Once again, connect LED_HIGH's cathode to the row where R1 is connected, and its anode into the + bus.

If you're cutting the LEDs' leads and bending them to fit the breadboard, remember that the square part of the lead will bend about three times before it breaks. That said, if you bend the anode lead to a right angle on LED_LOW and the cathode lead to a right angle on LED_HIGH, you'll get a nice offset like I did so the two LEDs aren't right on top of each other.

Now for the logic connection. Connect pin 9 to pin 11, and plug a jumper (or a longer piece of hookup wire stripped on both ends) into the socket next to pin 12. You're done. That's all there is to it.

Testing the Logic Probe

Make sure the probe lead of the logic probe is disconnected. Plug it into an unused socket row on the breadboard for safe keeping. When you connect the USB cable to the Cestino and to the host computer, LED_HIGH should light. How can that be, with no logic connected to the probe? Remember how the datasheet said un-connected inputs would drift high? That's exactly what we're seeing.

Now connect the probe lead to the + bus. Nothing happens? That's good. That's exactly what it should do. LED_HIGH should remain on.

If you connect the probe to the - bus, LED_HIGH should go off and stay off, and LED_LOW should come on.

If your LEDs aren't lighting at all, check to make sure that you have them oriented the right way: cathode to resistor leading to the 74xx00, and anode to the + bus. Also, you wired the 74xx00's power and ground circuits on pins 14 and 7 respectively, right?

If your LED_HIGH led came on when you connected the probe to the - bus but not when you connected it to the + bus, your LED_HIGH is wired backward and to the wrong bus. Turn it around and plug the anode into the + bus instead of the cathode into the - bus.

The same thing goes if your LED_LOW comes on while the probe is connected to the + bus. Both LEDs should be connected the same way: anode to the + bus, and cathode to the dropping resistor, even though they show opposite logic values.

To test your probe on pulsed circuits, plug the probe into pin 11 of the 74xx00. Both LEDs should light.

What's happening here? Pin 11 is the output that drives LED_HIGH. At the moment of connection, pin 11 is low, which pulls the input pin 12 low, which sets the output pin 11 high. Which turns the output on. Repeat forever. With pin 11 tied to pin 12 and pin 13 tied to the + bus, our 74xx00's gate oscillates, switching itself on and off as fast as it can. Each time it transitions from one state to the other, it switches one or the other output on, so fast that neither the LEDs nor our eyes can react to the difference, and both LEDs appear lit.

If your logic probe can pass these tests, it's working.

Floating Pins

Now that we have a working logic probe, let's have some fun with it. Plug the probe in to the socket row next to pin 1 (PB0) of the Cestino and hit reset.

Notice anything interesting?

When the Cestino LED flashes on (PB0 goes high), we get a high pulse, and when it goes off, we get a low pulse, exactly the way we'd expect. But a few seconds after the flashes stop, something interesting happens. LED_HIGH comes back on, but the Cestino LED does not.

This is not a malfunction. There are two fairly important facts to learn from this. First, at the end of the reset LED flash sequence, the Cestino holds PB0 low for a few seconds, but then it stops. Why is that?

I admit, I had to look it up. It turns out that the default state for any digital pin in Arduino is the input state: high impedance with no pull-up enabled. Once the flash sequence is finished, the Arduino core sets digital 1 (aka PB0) to that state. It no longer has a high enough current (in fact, it is delivering no current at all) to light the LED, but it isn't grounded either. It's floating.

The second fact is that the 74xx00's input drifts high enough to trigger the LED_HIGH NAND gate even when tied to PB0 (Pin 1, aka Digital 1) of the Cestino. Now we have to ask ourselves if the input of the 74xx00 is high enough to make that pin read high, and to tell that, we'll need a little code.

Logic Probe Test 1

I'm not going to go through this code in any depth, because frankly, you've seen it before. It's a stripped down version of TTL_Explorer_7400. It reads a different port and doesn't write to any ports, and its loop control is different, but literally everything else is the same. The only wrinkle is that we mask all but bit 1 to ensure that we're not reading

noise from the other pins as they float. Of course, we're using this code to read the state of a floating pin tied to a floating pin, so the results it generates have a lot of potential for randomness.

```
//Logic Probe Test 1
//----------------------------------------------------------
//This sketch checks to see if the value of PB0 changes due
// to the 74xx00's input rising. We mask out all the bits
// except PB0 so we don't pick up noise from uninvolved pins.
// ---------------------------------------------------------
//Hardware:
//74xx00 Logic Probe
//
// Wiring:
// Logic probe input connected to PB0/Digital1/Pin 1 of the
// ATmega 1284P. Optional: a 10k pull-down resistor
// from PB0 to the - bus. On my board it makes no difference.
// ----------------------------------------------------------
// James R. Strickland
// ----------------------------------------------------------
//precompiler definitions.
#define DELAYTIME 100 //How many mS (milliseconds) per update?

//global variables
byte counter = 0;

//-----------------------------------------------------------
//Zerobee
//-----------------------------------------------------------
//This function returns a string with the binary equivilent of
//the value passed to it in input expressed in 0bxxxxxxxx
//notation, with 8 bits for each byte regardless of their value.
//IMHO this makes binary values easier to read and compare.
//-----------------------------------------------------------
String zerobee(byte input) {
  String temp = "";
  for (int c = 0; c <= 7; c++) {
    if (input % 2) {
      temp = "1" + temp;
    } else {
      temp = "0" + temp;
    }
    input = input / 2;
  }
  temp = "0b" + temp;
  return (temp);
}
```

```
//setup() function - runs only once.
void setup() {
  Serial.begin(115200);
  DDRB = 0; //Set DDRB so all pins are inputs;
  PORTB = 0; //Clear all pull-up resistors on portB
  Serial.println("Starting…");
}

//loop() function - runs forever.
void loop() {
  for (int c = 0; c <= 20; c++) {
    Serial.println("PORTB:" + zerobee(0x0000001 & PINB));
    delay(DELAYTIME); //Wait DELAYTIME milliseconds.
  }
  Serial.println("Done.");
  while (0 == 0) {}; //Do nothing forever.
}
```

The results are equally uninteresting. Pin 0 of Port B reads zero, over and over again. We can answer the question conclusively: although the gate drifts high, its input does not pull the ATmega's input logically high. This is good to know.

Full Speed Ahead

So we've proven our probe doesn't trip the Cestino's inputs. What else can we learn about the probe—and the Cestino?

Try this. Plug the logic probe's input wire into pin 13 of the ATmega1284P, the clock input pin. Now press the reset button to make sure the Cestino still boots. (It should.)

If you look at the LEDs of the logic probe, both of them should be on. We're feeding the probe a 20MHz signal, straight from the TTL oscillator that drives the ATmega. The 74xx00 can go that fast the way we've wired it. Barely. But the LEDs can't come close. Nevertheless, the probe tells us the most important fact about that line: that there is a pulsed signal on it. Both LED_HIGH and LED_LOW are on, telling us the circuit is switching back and forth faster than the LEDs (and our eyes) can go. For debugging logic circuits, knowing that a line is changing states (rapidly, in this case) is often as useful as knowing the actual pulse pattern and speed. If our Cestino stopped working, for example, plugging the probe in on pin 13 would tell us conclusively that the oscillator is, in fact working (or if it is not.). It's this ability to correctly identify pulsed circuits that sets our simple logic probe apart from the capabilities of our multimeter.

Debugging Ports with the Logic Probe

In the circuits to come, we'll often be feeding a series of numbers to a given port, often for addresses. (I'll explain addresses in Chapter 8. For now, you just need to know that they're binary values on ports.) Because addresses are often in sequence, we can learn some additional things with the simple logic probe. Let's get some code and try it out.

Binary Numbers for the Logic Probe

Once again, the code we're going to use is derived from an earlier project, in this case, from Binary Numbers On Display. The only change we'll make is the DELAYTIME precompiler definition (set it to 10ms), so just cut and paste the code, or make a copy and save it under a new name. It's included here for your convenience.

```
//Binary Numbers For the Logic Probe
//----------------------------------------------------------
//This sketch counts in binary from 0 to 255, then resets.
//It's a really, really short sketch.
// ---------------------------------------------------------
//Hardware:
//The Logic Probe, wired to PC0, for starters, then PC1, PC2,
//and so on.
// ---------------------------------------------------------
// James R. Strickland
// ---------------------------------------------------------

//precompiler definitions.
#define DELAYTIME 10 //How many mS (milliseconds) per update?

//setup() function - runs only once.
void setup() {
  DDRC = 0b11111111; //Set DDRC to all outputs.
  PORTC=1;
}

//loop() function - runs forever.
void loop() {
  PORTC++; //Take whatever is in PORTC and add 1.
  delay(DELAYTIME); //Wait DELAYTIME milliseconds.
}
```

Connect the logic probe to PC0, on pin 22 of the ATmega1284P of your Cestino. Once the sketch is running, you'll notice both LED_HIGH and LED_LOW are on continuously. With only a 10ms delay, we can't see that they're changing. If we go to PC1, on pin 23, however, we are effectively increasing the delay to 20ms, and we can start to see the flicker. On PC2, with 40ms delay, the flicker is pronounced, clearly flashing both LEDs.

On PC3 with 80ms delay, each LED clearly turns off and the other clearly turns on. With 160ms on PC4, the flashing is positively sedate, a little fast perhaps for car turn signals, but not too bad. On PC5, it's much more like a car turn signal indicator, at 320ms or about 1/3 of a second between pulses. On PC6 we're getting a pulse every 640ms, or a two pulses every 3 seconds, and on PC7, we're getting them at less than 1 per second.

It's easy to see how you could debug an address circuit as it counts up addresses and make sure the pulses are visibly slower from the low order bits to the high order bits. This assumes that the addresses are being called in sequence from low to high, of course, but that's what test code is for.

As an exercise, try setting the delay to zero, which should get us a pulse every few cycles of the 20MHz clock. Even though the Arduino core's code is not known to be particularly instruction (and thereby time) efficient, when we let the ATmega1284P run at full speed, even PC7, 1/128th speed from PC0, is still faster than the LEDs (and our eyes) can respond. Debug code had better include generous use of delay(), too.

Credit Where Credit's Due

As usual, I have to credit Quinn Dunki's Blondiehacks blog (`http://quinndunki.com/blondihacks/`) for my newfound appreciation of logic probes. When I started back digging into TTL level debugging, the first thing I bought was the Logic Shrimp, a now-discontinued inexpensive logic analyzer, that let me essentially probe eight channels at the same time with my desktop computer. It's a very useful thing, but it's an awfully specialized tool, and I couldn't see asking readers to buy one. The logic probe was the right answer. As for the circuit, I saw a design for a simple logic probe with two LEDs wired in opposite directions *somewhere* on the net, but I can no longer find it. Buffering that with the 74xx00 was engineering by convenience on my part, solely to increase the input impedance and isolate the circuit under test with an IC I had readily to hand.

Further: Better Logic Probes and Logic Analysers

It goes without saying that the logic probe described here is primitive. It's designed, as I've said repeatedly, mostly to keep its input impedance high and deliver the most basic logic probe functionality. There are numerous ways it could be improved.

Most obviously, a way of actually measuring pulse lengths under about 30ms would be a big help. This could be done with a counter, like the 74xx92 described in the last chapter, which would divide the number of pulses by up to 12. In order to generate visible pulses from the 20MHz TTL oscillator, we'd need a bunch of them, so dual counters like the 74LS393 would be a good choice. We could then wire an LED from the + bus and one from the - bus to the lowest order output bit to get normal logic probe function, and a third LED that could be switched from the low order output to the high order output would provide both pulse indication and show the relative frequency, something my logic probe can't do.

But hey, why stop there? If we added a couple of seven segment drivers and a reset on a known frequency (another oscillator) we could output the frequency itself as multiples of 255 hertz per the oscillator's frequency. The engineering could get complicated pretty fast.

Adding a micro-controller, say, an ATmega328, would let us auto-range the frequency counter and time the pulses.

We could also keep the probe simple and add more channels, although it ceases to be a logic probe somewhere along there and grows into a logic analyzer if you add some accurate timing, some memory, and a nice display system.

This one will do for our needs from here on. Building a better one? That's up to you.

CHAPTER 8

EPROM/Flash Explorer

Every time you load a sketch to the Cestino, the software on your host computer compiles the sketch into instructions and uploads it into the Cestino's flash memory. You know all this. We dealt with it in Chapter 2. If you've watched your Cestino, you know that any sketch you upload remains in flash between reboots and power cycles. Persistent memory is a useful thing.

You've probably heard of firmware as midway between software—the code that runs on a device, and hardware—the device itself. Firmware is a kind of software, but it comes with the device and is available at power up, no external storage required, just like the flash in the Cestino's ATmega1284P. Firmware can be simple, a few kilobytes of instructions for an 8-bit microprocessor, or it can megabytes in size.

Here's an example. I have an old Ampro Littleboard computer from the mid 1980s. Its firmware fits in a 4 kilobyte EPROM, which gives it just enough smarts to boot from a floppy. By contrast, the 128k Macintosh, of approximately the same vintage, had 64kilobytes of ROM containing boot-up procedures, desktop components, and a wide variety of other routines so that this software didn't have to be stored in the machine's RAM. The Mac I'm writing this on routinely gets firmware updates in the 4-5 *megabyte* range, and I can't find documentation for how big its firmware space really is.

Firmware is ubiquitous. Most of the devices in your junk box will have some kind of persistent memory in them, somewhere, and what's in it will be firmware.

How interesting is that?

The answer is, it depends. For really tiny firmware like my Ampro's, what's in there will be mostly machine code, unreadable if you don't happen to be a Z80 microprocessor or dedicated enough to decode each byte and figure out what it told the micrprocessor to do. For others, there's lots of that, but also all the text strings the firmware can display. These can make for interesting poking around, even without disassembling the machine-readable parts of the firmware. They can give you a peek into how the device worked, and if there were any "Easter eggs" in it—little jokes or references included by the original programmers when a little space was left over in the firmware device.

© James R. Strickland 2016

J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_8

■ **Note** The firmware of a device is nearly always copyrighted under very restrictive terms. If you read the license information back when you bought your device, it may have included agreeing *not* to disassemble the firmware. We're peeking where we weren't invited, and if binaries of that ancient firmware wind up online, there are companies who will contact their lawyers.

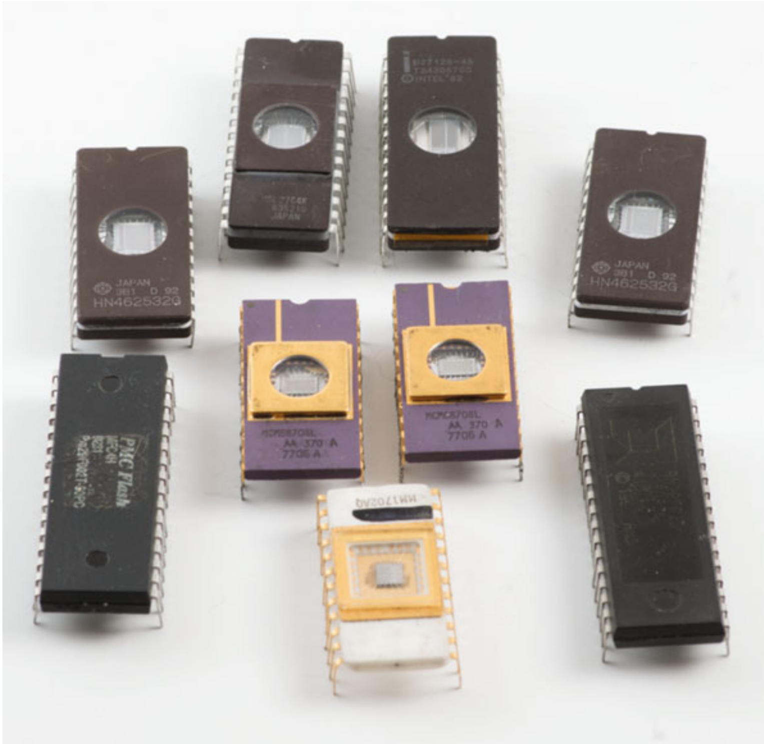

Figure 8-1. *EPROMs and Flash*

The EPROM and Flash we're going to dig into here, some of which are shown in Figure 8-1, are vintage parts, but I burned the ones shown in this book with open source firmware licensed under the GPL, so that I can quote parts of them in this book and not get in trouble. It's your equipment. It's unlikely anyone will care if you look into its guts, but use caution about sharing that data with others, and do so at your own risk.

The Stuff You Need

This project has a tiny little shopping list. We're moving up in the world. Individual components do a lot more.

New Parts

None!

Used Parts

A PROM/EPROM. Nearly any old 5v 27xx PROM or EPROM will work, although really old ICs may have very different wiring pinouts than the early 1980s vintage 27128 that I used. Truly ancient EPROMS like the Intel 1702 (dating to 1971 or so, before the JEDEC standard emerged) will often want a second (and third) power supply voltage, usually negative relative to ground, and we can't power them easily with the Cestino. 2716s should probably be avoided unless you can find the datasheet for your exact part from your part's manufacturer, as they were manufactured just as the JEDEC standard was being put into place and there are several different pinouts for the same IC number, some of which require more than the single 5-volt power supply.

Parallel EEPROMs and Flash ICs will also work (there's a slightly different schematic and sketch for them) and, as long as they don't require a Vpp (programming voltage) above 5v, we can reprogram them as well. (EEPROMS will require the sketch to be modified to erase the bytes singly rather than with a single erase command, and the method will likely be different.)

If you're buying new, I recommend the SST39SF020A-70-4C-PHE, available at Mouser. It's what I used, and should work with the circuit and the sketch provided. Other types may require modifications to the sketch. You can get new 27Cxx EPROMS from the likes of Futurelec, but they'll be blank, and thus, not much fun to read. As always, make sure you get the DIP, CDIP, or PDIP version, as the others won't work in your breadboard.

A Quick Introduction to Hexedecimal

Up until now, we've been dealing with single bytes, and it's been convenient to represent them in binary notation: 0b00000000 to 0b11111111. As our numbers get larger than 255, however, binary notation starts to get painful. When faced with 0b1111111100000000, which is only a 16-bit number, or its decimal equivalent, 65280 it gets very difficult to sort out which bits go where.

As we've already talked about back in Chapter 4, base 10 and base 2 are arbitrary choices. There's another numbering system that's much more convenient to use when dealing with large binary values. It's called hexadecimal, usually shortened to "hex." Hexadecimal is a base 16 number system (hex—six, deci—ten.) It counts 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Hex is very, very convenient to use with binary numbers, as each digit can exactly represent the contents of four bits. No more, no less. In the C, C++ and Arduino worlds, we represent these with 0x in front of them, so low values aren't

confused with decimal values, just as we used the 0b in front of binary values for the same reason. 0b1111 is 0xF, 0b0000 is 0x0, 0b0001 is 0x1 and our ugly 16-bit binary value 0b1111111100000000 aka decimal 65280 becomes 0xFF00. If we need to know which bytes have what values (which we will in this chapter) we just take pairs of digits from right to left. 0x00 is 0, 0xFF is 255, and 0x1FF is 256.

Often times, we're taught math as though it is a great capital-T Truth. It's not, really. Math is a language, a symbolic system to represent quantity. How that quantity represented, among other things, while rigorously defined and internally consistent, is essentially arbitrary. In this chapter we'll be using hex to represent our values because it's convenient. Make math work for you, not the other way around.

ROM, EPROM, EEPROM, and Flash: A Recognition Guide

In the beginning (or at least prior to 1971, which is close enough) there was ROM. ROM stands for read-only memory. These were ICs where the data was simply encoded in their masks, when the ICs were manufactured. Mask ROMs were great, if you were making millions of a given piece of electronics, and the firmware never needed to change. The cost to set them up was enormous, and it took weeks to make them, so you had to get your firmware right the first time.

So things remained until 1970, when Intel's Dov Frohman was troubleshooting another IC entirely, dealing with transistors whose gate connections had broken. He noticed that these transistors had very different properties, and exploited those properties to develop the Intel 1702 EPROM, or Erasable Programmable Read-Only Memory.

Essentially, a memory cell in an EPROM is a single FET transistor with an extra, unconnected gate layer. When the cell is programmed, high voltages (47v on the 1702) between the gate and the drain force electrons to migrate across the insulating layer between the control gate and the floating gate. Since FETs draw essentially no gate current, there they stay. This is called hot electron injection. The charge between the gate and the floating gate is the electronic representation of the data.

When we go to read the data, we apply a (much lower) voltage to the control gate. If there is no charge stuck between the floating gate and the gate, the FET acts normally, and switches on. If there is a charge between the floating gate and the gate, it raises the voltage required to make the FET conduct. We apply our usual low voltage to the gate and… nothing happens. Programming an EPROM is not about programming where the ones should be. It's about programming where the zeros should be.

To erase an EPROM, you shine ultraviolet light in the quartz window in the top of the package (if you look, you can see the actual die of the integrated circuit) for several minutes (depending on the intensity of your UV light source, but 15-20 minutes is the norm) until the insulating layer ionizes and gives the electrons a path back out, turning all the bits back to 1.

When you look in a really old piece of electronics, something from the 1980s or 1990s at the latest, and you find an IC with a sticker of some kind (often silver) over the middle, you've probably got an EPROM. You can take that sticker off if you need to read the numbers on the chip. It won't erase right away. It takes weeks of full sunlight to erase an EPROM. Properly protected (covered with a sticker inside a dark computer case) EPROMs are rated to store data for decades. The truth is, I've never found a used one that was blank, even after 30 years, although I can't vouch for the data's validity.

Most parallel EPROMS will have id numbers that begin with 27, like 2716 or 27128, where the remaining digits will tell you the size in *kilobits*. So a 27128 has 128 *kilobits,* which if you divide by 8, gives you 16 *kilobytes.* That's how the standard says they should be. However, when you're dealing with stone-age electronics like this, you have to remember that the JEDEC standard for EPROMs came out after the first couple generations of them were manufactured, so there are lots of other numbers out there.

Generally speaking, if it's the same width as the ATmega1284P (0.6 inches or a bit over 15mm) and it has a sticker or a quartz window, it's probably an EPROM. If it has a C after the 27, it's a CMOS EPROM, which tells you about its manufacturing technology, and that it will probably take less power and lower voltage to program it. But read the datasheet.

■ **Note** Some EPROMs have different word sizes—particularly 16-bit words—but they're few and far between and easy to recognize by having more pins than usual.

PROMs are a type of ROM. PROM stands for Programmable Read Only Memory. In reality, they're EPROMS without the quartz window. If you could get UV inside them, you could erase them, but the die is sealed up inside the package. If you happened to have an X-ray machine you could possibly erase them, but PROMs were cheaper than EPROMs because they didn't have the erase function tested, in addition to lacking the quartz window.

PROMS have the same IDs as their erasable counterparts and are pin compatible. Presumably the manufacturers and JEDEC felt it was obvious: if your 2764 has a big quartz window in it, it's an EPROM, and if it doesn't, it's a PROM. Still, it's something to watch out for if you're ordering these parts.

EEPROMs work like EPROMs do, save that more recent types can be erased one bit at a time electrically. They do this by field electron emission, that is, electrons quantum-tunnel through very, very thin oxide layers, thus clearing the cell. Quantum mechanics at work. Their numbers always start with 28. Some can be programmed and erased at 5v (or lower) due to the addition of charge pumps (combinations of diodes and capacitors that allow voltage to be multiplied electronically) to produce the higher voltages needed for programming and erasing.

Flash is a specialized type of EEPROM, distinct from EEPROMs mostly by the fact that they are NAND devices, whereas EPROMs and EEPROMs are NOR devices. The difference? NOR types can access only single words of data (bytes, most of the time) like their EPROM and EEPROM ancestors, whereas NANDs can go down to the bit level, which is important if your flash is pretending to be a hard drive. NAND also allows more cells to be in the same area of silicon, as does the ability to erase only in blocks

(sometimes called sectors). Modern flash, of the type found in flash drives, replaces the floating gate with charge trapping. An insulating oxide layer without a metal floating gate is injected with electrons via hot carrier injection, similar to hot electron injection, causing electron charges to migrate into the insulating oxide layer, programming the bit. The bit is erased by quantum tunneling, exactly as it's done in EEPROM. The big advantage to this, besides fewer layers in the manufacturing process, is that multiple cells can share the same trap, and the number of write/erase cycles the flash can endure before it wears out is hugely improved.

Parallel flash ICs will have numbers that begin with 29 or 39, followed by an F (for flash.)

There are lots of other types of programmable ICs. The ATmega1284P is one, obviously, but there are also various logic arrays (GALs, FPGAs, and so forth), serial EEPROM and Flash, and so on. (Serial EEPROMs and Serial Flash will have different ID numbers beginning with 24 or 25, respectively. These can be viewed with the Cestino too, but the sketch would be quite different.)

The devices we're playing with in this chapter are strictly the parallel memory types. What all these devices have in common is this: when you feed one an address on its address lines, it sets its data lines to the data it's storing at that address. So if I have an an Intel 1702 flash, and put 0x0F (turn on the lowest four bits of its address lines), I will get the sixteenth byte (remember, addresses start with 0) that was stored in that EPROM.

Build the EPROM Explorer

I have in my breadboard an Intel 27128 EPROM dating to 1982 (the package is large and they just stamped the date on it directly), courtesy of a friend who emptied his junk box of ancient EPROMs to the benefit of mine. You can see it in Figure 8-2, buried in the nest of wires we're about to build. According to the datasheet, it's a 128 kilobit (16 kilobyte by 8) EPROM with a 250ns access time. That was fast in 1982, according to the datasheet. My 27128 has a 22v programming voltage, which there's no easy way to generate on the Cestino. Oh well. We can read it with 5 volts, so let's see if there's anything interesting in it.

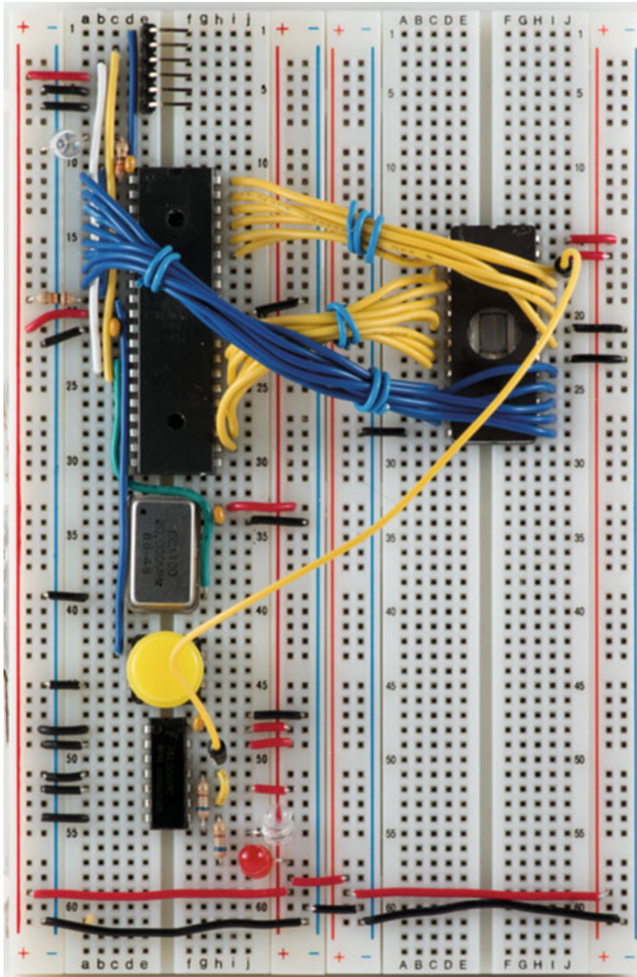

Figure 8-2. *EPROM Explorer*

I know what you're thinking. The schematic in Figure 8-3 looks pretty strange. A bunch of lines are all tied together? All the pins of several of the Cestino's ports are tied together? What? That can't be how it works. Well, you're right.

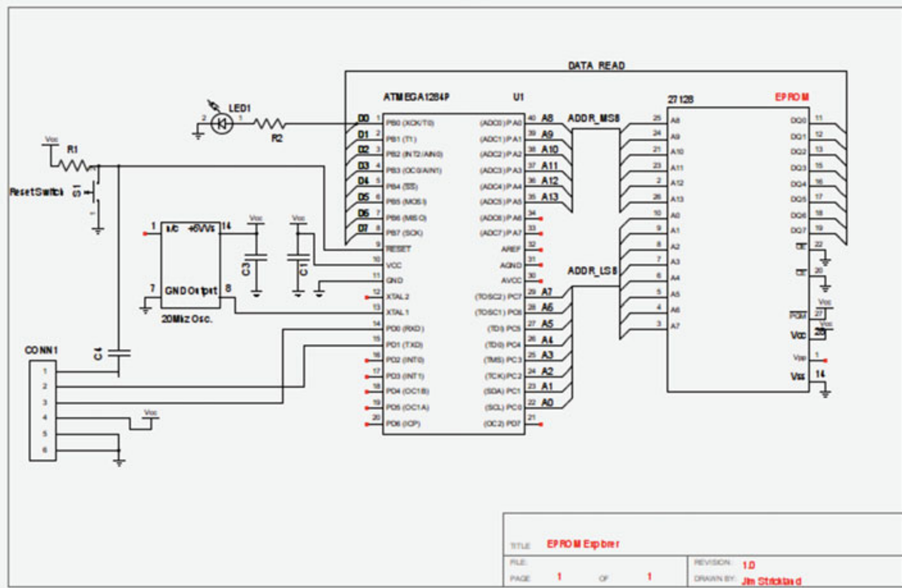

Figure 8-3. *EPROM Explorer Schematic*

Getting On the Bus

If you remember all the way back to Chapter 2, when we built the Cestino, we called the power and ground lines that span both breadboards and the lines up and down the sides of the boards "busses," like the + and - bus. You may have also heard of the PCI-E bus, Universal Serial Bus (USB), ISA Bus, Unibus, Q-Bus. (okay, probably not the last two. They're long-extinct proprietary minicomputer bus standards.) It's time we come to grips with what a bus *is* because we're about to build a couple of them.

A bus is a bundle of wires or traces on a circuit board. It comes from the same Latin root, omnibus (literally 'for all') that omnibus spending bills Congress so loves, and the big yellow vehicle that some of us had to ride to school—school bus—does. The word probably comes to us by way of the bus bar (sometimes spelled buss bar) in electrical work, where multiple circuits might tie to a really thick, hollow bar acting as a conductor. It's a good description of what a digital bus does. It connects multiple devices together so they can communicate with each other. On more complex busses, it also allows for signaling so that devices don't clobber each other's communications on the bus. Sounds like the bus is a standard, as much as electronics, doesn't it? You've got a good ear. Busses are standards, defined in terms of signals. If your device has the same signals in the right order at the right speed, it should be able to communicate on the bus.

Busses can have slots for circuitry to be added in, but they don't have to.

A computer bus standard might have eight data lines and 16 address lines and a bunch of other signals for arbitration for who gets to use the bus when. It might define the interface electronics—voltages, currents, what the sockets are like (if there are any), what speed and order the signals have to get there, and so on. Even more complex, modern

busses, like PCI-E and USB are serial busses. They have multiple lines for bandwidth, but each line is its own communication, and does not depend on the simultaneous arrival of any of its peers.

The busses we're using to talk to the EPROM are simple parallel busses. One for data, one for addresses. For us, it's important that all the lines on our parallel bus be sent simultaneously. Here is where using ports as 8 bit units pays off. The faster we go, the less practical turning signals on and off individually becomes.

The three busses in the schematic are indicated by one line with a lot of lines connected to it. These lines aren't connected together electrically. Instead, they're tied into a single bus, in this case eight wires, corresponding to a single port. In order to maintain sanity, each signal on the bus is labeled at both ends, so A0, A1, A2, and so forth on the IC are connected to A0, A1, A2, and so forth on the port we're using for the low order address byte on the ATmega1284P. The bus line in the schematic corresponds to all eight lines. It makes for a neater (readable) schematic. Now you know how to read it.

In order to make a more photogenic project, I've hanked (tied) the lines for each 8-bit bus together, and color coded them by their function. The two yellow busses are ADDR_MSB and ADDR_LSB, for example. Technically they're one bus, but I split them apart for clarity and hanked them that way. You don't have to do it this way, but you might want to consider it. The prototype was like an explosion in a technicolor spaghetti factory. When I found the method I used in the photos I started liking breadboarding a whole lot more. It does make a logic probe more necessary, since it's harder to see exactly what wire goes from one pin to the other. Fortunately, we have one.

Power, Ground, and Unused Signals

Let's do the easy connections first. Vcc on pin 28 to the + bus. Vss on pin 14 to the - bus. Vpp we don't care about. The datasheet says it should be at Vcc for read. I left mine floating. It doesn't seem to make a difference. If we were actually planning to use the code in the EPROM, it'd be better to wire that to the + bus, but if a little noise creeps in while we're just looking around, it probably won't matter.

/CE (or /E on some datasheets) on pin 20 stands for chip enable. We'd normally use that to switch which EPROM we were talking to, but we only have the one. It's active low, so wire it to the - bus. We always want the EPROM enabled. Likewise, /OE, Output Enable (also known as /G for Gate) on pin 22. It's also active low, and it controls when the EPROM can talk to the data bus. Our answer to that is "always," so wire that to the – bus, too. /PGM is for Program. It's active low, so if we were programming this EPROM, we'd pull it low, with Vpp at 22.5v, and go hot-electron-injecting to write zeros to our EPROM. We won't be doing that, so wire that to the + bus to hold it high.

If you're starting to suspect that the control bus I mentioned previously might involve signals like that, you're exactly right. File that away, you'll see that again before the end of this chapter.

Data Bus

DQ0 to DQ7 are the data bus. They may be called D0-D7 on your datasheet. It means the same thing. This is the 8-bit data port of the EPROM. Not surprisingly, we'll tie it in to an 8-bit port on the ATmega1284P, in this case port B. As you can see in the schematic, connect PB0 (ATmega1284P pin 1) to DQ0 (EPROM pin 11), PB1 (ATmega pin 2) to DQ2 (EPROM pin 12), and so on for all 8 data pins. I'd really encourage you to make all these connections with the same color jumpers if you can. If you want to get fancy and hank the wires like I did in Figure 8-2, I'd suggest waiting until we've debugged this wiring.

If you look at the actual IC (the pins in the schematic are not in order) you'll see that DQ0-3 is on the left side of the IC, right above the ground pin, and the remaining 5 pins go up the other side. If you peek ahead to Figure 8-4, you might notice that while the 39SF020A has more pins, the data pins are in exactly the same place relative to the bottom of the IC. Remember I mentioned the JEDEC standards for EPROMs and flash? Pin placement is one of those standards. Not all EPROM/flash makers adhere to the standards, but when they do it makes wiring a lot easier.

From now on, we'll call this bus DATA_READ.

Address Bus(es)

The 27128 EPROM has 14 address pins, 0-13, which you'd expect for 16kB of address space. In order to span the full range of addresses the EPROM is capable of, we'll need two Cestino ports, although two pins of one port won't be connected to anything.

Here is where endianness matters. If you know your EPROM came out of a big endian machine like an Amiga or an old Motorola 68000 series Mac, you could wire the IC's low order port (A0-A7) to PORT A and the high order port (A8-A14) to PORT C and change the sketch a little, but you'll have to change the sketch anyway, so it'd be easier to change which ports are used for which there.

Remember back in Chapter 2, where we talked about the address bus size, register sizes, and all that? Is any of that coming to mind now? Here's how it all ties in. In a microprocessor or microcontroller with an external memory interface, there's one register, a bit of memory inside the processor, called the address register. When a program tells the processor to write a given value to a given spot in memory, it sets the address register to the the address it was given, sets some control bus bits, then sets the data bus to the value, sets or clears some more control bits, then moves on to the next instruction, which itself sets the address register, reads the instruction in the data port, and so on. We'll get into this in much more detail in chapter 11 when we'll be looking at a microprocessor whose only memory interface is external.

If the ATmega1284P had an external memory interface, it would be little-endian, and you'd have to change the wiring. Sadly, there don't appear to be any ATmegas with external memory interfaces in DIP packaging. Instead, we're doing what's called "bit banging," that is, generating bit patterns with software to trigger hardware. It's slower, but the bottle neck is still communication with the console and our ability to read what comes out. We're not in that big a hurry.

When I talk about an external memory interface, I'm talking about pins the microprocessor controls when selecting memory addresses, and a data bus for reading data and instructions into the microprocessor. The ATmega1284P obviously has both these components, but they're internal, and we can't touch them. If you want to play with external memory and Arduino, the Arduino Mega's 1280 or 2560 ATmega microcontroller does have an external memory interface for up to 64kB of ram, or for reading EPROMs. It'd be a very different sketch.

So let's wire our address bus. It connects two ATmega1284P ports to the 14 address pins on the EPROM. In the photo in Figure 8-2, all the address lines are yellow, and each 8-bit port's wires are hanked together. We'll start with the least significant bit, that is, bits 0 to 7. These are wired to port C of the ATMega1284P. As with the data port, PC0 (ATmega pin 21) connects to A0 (EPROM pin 10), PC1 connects to A1 (ATmega 22 to EPROM 9) and so on, until you get to PC7 and A7.

From now on, we'll call this the ADDR_LSB bus, which is part of the larger address bus.

At this point, you might want to skip ahead to the sketch and debugging, as you can, theoretically, access the first 255 bytes of the EPROM. It's easier to debug things when you don't have 24 wires running across your breadboard.

Back so soon? Okay, let's go ahead and wire up the most significant byte (or part of a byte, in this case). We're using Port A for the ADDR_MSB port. The wiring is a tiny bit more complicated, as PA0 connects to A8, but the indicators in the schematic should make it clear what we're using the ATmega pins for and which pin to connect them to on the EPROM. It's also a little more complicated because the MSB address pins are scattered all over the top end of the EPROM. Again, the pins shown on the schematic are *not* in the order they are on the IC.

PA0 connects to A8 (ATmega 40 to EPROM 25), PA1 to A9 (ATmega39 to EPROM 24), PA2 to A10 (ATmega38 to EPROM 21), PA3 to A11 (ATmega37 to EPROM 23) PA4 to A12 (ATmega36 to EPROM 2) PA5 to A13 (ATmega 35 to EPROM 26). If your EPROM is bigger, connect the additional address lines to PA6-PA8 and set MAX_ADDRESS in the sketch accordingly.

That should cover the wiring. Let's plug the Cestino in (you did unplug it first, right?) and see what we get.

The EPROM_Explorer Sketch

This sketch isn't hugely complicated. It's really a derivative of Binary Numbers on Display, but there are a bunch of functions and some C/C++ wrinkles I haven't shown before, so it does bear a close look.

What this sketch needs to do is: generate 14 bit addresses, put them on the ADDR_LSB and ADDR_MSB pins to feed the EPROM, then read the DATA_READ lines and display whatever printable information is in the EPROM at that address to the console. Sounds like two functions to you? It did to me. Before we cover those functions, though, we need to cover the preprocessor definitions.

```
#define ADDR_LSB PORTC
#define ADDR_MSB PORTA
#define DATA_READ PINB
#define MAX_ADDRESS 0x3FFF
```

Remember that #define runs at compile time, and thus consumes none of the ATmega's memory at run time. The preprocessor substitutes the second value for the first value before GCC goes to work on our code. So we substitute ADDR_LSB with PORTC, ADDR_MSB with PORTA, and DATA_READ with PINB. We're giving aliases to our ports, in case you want to modify the code to use different ports, or make this sketch work on a different Arduino entirely. It also makes the code more readable.

The last define is MAX_ADDRESS, the maximum address in the EPROM, in hexadecimal. Its decimal equivalent is 16383. Since this is the number of bytes this EPROM can store, and since the EPROM is governed by JEDEC standards, we can say the EPROM holds 16 kilobytes. (I'll rant more about this subject in Chapter 9.) If you prefer the international standard names for such things, you can go ahead and call them 16 kibibytes. Which is a mouthful.

The functions come next, and in this sketch they do nearly all the heavy lifting. We'll start with select_EPROM_address, as it's the most complicated.

```
void select_EPROM_address(uint16_t EPROM_address) {
  union {
    uint16_t address_uint;
    byte byte_array[2];
  } address_union;

  address_union.address_uint = EPROM_address;

  ADDR_LSB = address_union.byte_array[0];
  ADDR_MSB = address_union.byte_array[1];
}
```

Remember the new C/C++ code wrinkle I mentioned? It's the union at the top of this function. Up until now, we've been dealing with single byte values, thrown out a single port. This is all well and good, but the 27128 EPROM has 14 bits of address space. You already know from Chapter 4 that as numbers in the Cestino get bigger than 255, the number of bits goes above eight. In this case, we're using a two-byte unsigned integer datatype called a uint16_t. We could subdivide this two-byte variable to get our least significant bits (bits 0 to 7) and most significant bits (8-13) with bitwise ands and rotations, but in C, and by extension C++, there is another way.

In C, every variable is really a pointer to an area of memory. It's an address. What happens if you give two different variables the same address? They both access the same area in memory. And if those two variables don't happen to be the same type? You can access the data in that area of memory in different ways. And that's exactly what the union is doing. A C union is two or more representations of the same value in memory. Two pointers to the same area in memory.

The union we're creating is called address_union. It has two variables: a uint16_t called address_uint, and a two byte array called byte_array[]. Always bear in mind that these two variables are two different representations of the exact same data in memory. So when we set address_union.address_uint to the address that's passed in to the select_EPROM_address() function, we are also setting both bytes of address_union.byte_array[]. What happens next is pretty straightforward. We set the ADDR_LSB port to address_union.byte_array[0] and the ADDR_MSB port to address_union.byte_array[1].

■ **Note** The address_union.byte_array only produces the correct byte order on little-endian systems like ATmega, intel, and so on. If you know your EPROM came out of a big-endian system: say, an old Macintosh, or a Radio Shack Color Computer with a Motorola processor in it, your address byte order will be big endian. To fix this, swap ADDR_LSB and ADDR_MSB in this function. If you know your EPROM came from a 16-bit system like a Macintosh, the data bytes will also be in big_endian, probably spread across two or more different EPROMs.

The next function, like most of the long functions in this book, is a pretty-printer. You could just dump the values from the EPROM straight to the serial console, but you'd quickly find your console full of gibberish if you did. Also, the Arduino serial monitor may be programmed to react in certain ways to certain normally unprintable characters. It knows tabs and linefeeds at the very least. So it's probably a good idea to filter the output so it doesn't make a horrendous mess on the serial monitor. If you really want to see the exact contents of every byte, you could change this function to output the values in hex.

```
void dump_EPROM(uint16_t start_addr, uint16_t end_addr ) {
  String data_line = "";
  for (uint16_t c = start_addr; c <= end_addr; c++) {
    select_EPROM_address(c);

    if ((!(c % 0x40)) && c > 0) {
      Serial.print("0x");
      Serial.print((c - 0x40), HEX);
      Serial.print("\t ");
      Serial.println(data_line);
      data_line = "";
    }

    if (isprint(DATA_READ)) {
      data_line += (char)DATA_READ;
    } else {
      data_line += ".";
    }
  }
}
```

The dump_EPROM() function takes two uint16_t parameters: start_addr and end_addr. These are the start address and end address, respectively, and they go straight into a for loop definition just like you'd expect, right after we define a String object called data_line and set it empty.

For each address the for loop generates, we go to that address in the EPROM using select_EPROM_address(). Before we read anything, however, we do some housekeeping.

We're going to accumulate every 64 bytes (or markers for unprintable bytes) we gather from the EPROM in data_line, and then print them en masse. It's faster, and less prone to noise related errors, particularly if we happened to be using pins on port D. (Which we will before the end of this chapter.) The easiest way to make data_line print when it should is to take the modulus (remainder) of the address we're at by 64. That's the next thing we do. If there is no remainder of the address divided by 64 (0x40), and if c is not zero, we print the start address of the line in hex, then a tab, then the contents of data_line. Then we clear data_line.

This is our first trip through the loop, and c is 0, so we fail the second test. Nothing happens.

Next, we pass the data on DATA_READ (remember, this is PINB, or the read register of PORT B, which is connected to our data pins on the EPROM) to the C isprint function. If that function tells us this is a printable character, we add it to data_line. If it's not printable, we add a period to the data line instead, so everything will continue to line up in the serial console.

Then we go back around the loop. When c hits 64, we Serial.print an address, then Serial.println data_line, then clear data_line to accumulate the next line.

```
void setup() {
  Serial.begin(115200);

  DDRC = 0b11111111;
  DDRA = 0b11111111;
  DDRB = 0b00000000;
  Serial.println("\r\rRunning");
}
```

The next function is setup(). Very little happens here, but what does happen is critical. We set up the serial console, set ports C and A to output on all pins, as they're our ADDR_LSB and ADDR_MSB ports, respectively, and port B to input on all pins because it's DATA_READ, and tied to the data pins of the EPROM. (Note that this is 0b for binary instead of 0x for hex.) After that, we print a message to the serial console that tells us when the program starts.

```
void loop() {
  dump_EPROM(0, MAX_ADDRESS);
  Serial.println("\r\rDone.");
  while (0 == 0) {};
}
```

Loop is even less interesting. It calls dump_EPROM from 0 to max_address, prints a message to tell us when we're done, and then drops into a while loop that runs forever to keep loop() from being called again.

Here's the complete sketch, comments and wiring guide intact.

```
//EPROM_Explorer
//------------------------------------------------------------
// Dump the readable characters of a 27128 EPROM to the
// Console.
// -----------------------------------------------------------
//Hardware:
//For a 27128 or similar EPROM:
//ATmega Pin Pin Name Port name 27128 Pin  Pin Name Function
//          1 PB0      DATA_READ        11  DQ0      DATA
//          2 PB1      DATA_READ        12  DQ1      DATA
//          3 PB2      DATA_READ        13  DQ2      DATA
//          4 PB3      DATA_READ        15  DQ3      DATA
//          5 PB4      DATA_READ        16  DQ4      DATA
//          6 PB5      DATA_READ        17  DQ5      DATA
//          7 PB6      DATA_READ        19  DQ6      DATA
//          8 PB7      DATA_READ        19  DQ7      DATA
//

//         22 PC0      ADDR_LSB         10  A0      ADDRESS
//         23 PC1      ADDR_LSB         09  A1      ADDRESS
//         24 PC2      ADDR_LSB         08  A2      ADDRESS
//         25 PC3      ADDR_LSB         07  A3      ADDRESS
//         26 PC4      ADDR_LSB         06  A4      ADDRESS
//         27 PC5      ADDR_LSB         05  A5      ADDRESS
//         28 PC6      ADDR_LSB         04  A6      ADDRESS
//         29 PC7      ADDR_LSB         03  A7      ADDRESS
//
//         35 PA5      ADDR_MSB         26  A13     ADDRESS
//         36 PA4      ADDR_MSB         02  A12     ADDRESS
//         37 PA3      ADDR_MSB         23  A11     ADDRESS
//         38 PA2      ADDR_MSB         21  A10     ADDRESS
//         39 PA1      ADDR_MSB         24  A09     ADDRESS
//         40 PA0      ADDR_MSB         25  A08     ADDRESS

// ------------------------------------------------------------
// James R. Strickland
// ------------------------------------------------------------
```

```
//preprocessor definitions.
#define ADDR_LSB PORTC
#define ADDR_MSB PORTA
#define DATA_READ PINB
#define MAX_ADDRESS 0x3FFF

// -------------------------------------------------------------
// select_EPROM_address(address)
// -------------------------------------------------------------
// This function sets ADDR_LSB, ADDR_MSB, and the first two
// bits of ADDR_BANK_CTRL to the uint16_t (four byte unsigned
// int) address passed to it. These ports are wired to the
// EPROM IC's address pins.
//
// To break the address down into single byte chunks, we use
// a union, which essentially maps two variables (a 2 byte
// unsigned int and an array of 2 bytes, in this case) to
// one area of memory. Then we can peel off the bytes as we
// want them. Note that this code is endian-sensitive. If
// you run it on some future big-endian Arduino, you'll need
// to change the byte order.
// -------------------------------------------------------------

void select_EPROM_address(uint16_t EPROM_address) {

  //Declare the union we're going to use for address processing.
  union {
    uint16_t address_uint;
    byte byte_array[2];
  } address_union;

  address_union.address_uint = EPROM_address;

  ADDR_LSB = address_union.byte_array[0];
  ADDR_MSB = address_union.byte_array[1];
}

// -------------------------------------------------------------
//dump_EPROM(start_addr,end_addr)
// -------------------------------------------------------------
// This function dumps the contents of the EPROM to the console.
// data_line is an Arduino String object. We initialize it empty.
// Iterate through all the addresses, start_addr to end_addr,
// inclusive.
// If we've gotten to 64 characters gathered, print the start
// address of the line, then print the data_line and clear it.
// If the data on DATA_READ is printable, add it to the current
// data_line. Otherwise add a period.
// -------------------------------------------------------------
```

```
void dump_EPROM(uint16_t start_addr, uint16_t end_addr ) {
  String data_line = "";
  for (uint16_t c = start_addr; c <= end_addr; c++) {
    select_EPROM_address(c);//iterate through all the other addresses

    if ((!(c % 0x40)) && c > 0) { //if we're at 64 characters
      Serial.print("0x");
      Serial.print((c - 0x40), HEX);
      Serial.print("\t ");
      Serial.println(data_line);
      data_line = ""; //clear the data line
    }

    if (isprint(DATA_READ)) { //if the data on DATA_READ is printable
      data_line += (char)DATA_READ; //add it.
    } else {
      data_line += "."; //otherwise add a period.
    }
  }
}

// ----------------------------------------------------------
// setup()
// ----------------------------------------------------------
// Set up serial and port data directions. Initalize ADDR_BANK_CTRL
// so that the EPROM is ready to be read. Tell the user that we're
// running. Runs once.
// ----------------------------------------------------------

void setup() {
  Serial.begin(115200);

  DDRC = 0b11111111; //ADDR_LSB - Address LSB Port
  DDRA = 0b11111111; //ADDR_MSB - Address MSB Port
  DDRB = 0b00000000; //DATA_READ - Data port
  Serial.println("\r\rRunning");
}

// ----------------------------------------------------------
// loop()
// ----------------------------------------------------------
// Call dump_EPROM, then do nothing forever.
// ----------------------------------------------------------
```

```
void loop() {
  dump_EPROM(0, MAX_ADDRESS); //Dump the entire EPROM.

  Serial.println("\r\rDone.");
  while (0 == 0) {}; //do nothing forever
}
```

Output

What kinds of things can you see inside an EPROM? Well, here's what was in mine.

```
Running
0x0  U.......11/28/10XTIDE110-=XTIDE Universal BIOS (XT)=..v1.1.5 (1
0x40 1/28/10)........>..............................................
0x80 ...............................................................
```

Hey, it's an IDE BIOS extention for IBM PC/XTs. It dates to 2010? Well, yes. Remember how I said the contents of EPROMs tended to be copyrighted and I loaded these myself? This BIOS is the open source XT-IDE BIOS by Tomi Tilli. You can read all about it in Credit where Credit's due.

```
0xC0  .................................%s@ %x...Master.Slave .IDE %s
0x100 at %x: .not found...Floppy Drive. Hard Drive .Booting from %s %
0x140 x.%x ... .Boot Sector.found.%s %s!...Error %x!...Boot menu callb
0x180 ack via INT 18h.FDD.HDD.%c to %c boots from %s with %s mappings.
0x1C0 ..Copyright 2009-2010 by Tomi Tilli.%s %c.Foreign Hard Disk.ROM
0x200 Boot.Capacity : .%s%u.%u %ciB.%s%u.%u %ciB/ %u.%u %ciB.Addr..Bl
0x240 ock.Bus.IRQ.Reset.%s.%5u.%c%2u. %c%c.%5x.L-CHS.P-CHS.LBA28.LBA48
0x280 .%sUnknown.%s5." or 3." DD.%s%c%c", %ukiB.....1.........QP.....
```

Lots of strings. They tell us the age of the EPROM's code, many of the error messages, and so on.

```
0x2C0 .@.....0.XY.0123456789ABCDEFuxsScCUS....1...[].UVS..1.1...<.t...
0x300 ..B..[^]..UWQR.........1.<.t.<%t#........).~.Z.....R..ZY_]......
0x340 .....<.t.....0...v.Q........Yu............F.EE.g....F.EE..u....
0x380 ........V.EE.`.....V..^.....Q.....F.EE.x..V.EE.1....`.j.t.......
0x3C0 ..PR..........ZX.........W1.SQ......1...R..1...Z......0...G..u..
0x400 .Y[_.S0.......R.h...ZB[.S1.K...R.h...ZB[.UWVQ....1..............
0x440 ...t..............s...0.Y^_]....u.......0.........u....Selection
0x480 Timeout %us..1.................. ....Q....N.0..N....(.. ...YQ..
0x4C0 ..V........Y..~..tG......:.~..t:.F..u4......'.~..t'.}..N..V.0.;V
0x500 .r..V...R...ZA9.r.............Q.F..u...;N.t.. .... .........V..|
0x540 .Y.....V....s.1..k..o.....N.0...Q.i....Y...r.1..<..N.0..F..u...t
0x580 zQ.Y....YQ.@....Y...f1.....n.....tX...'..Q.~..t.1.....N.0.Q.....
0x5C0 .Y............. .....f.... ..............N.0.II................
0x600 ..~..t..V...u..7..f.Q....YP......^....*V.B.v.(...0..............
```

Readable strings are getting thinner. What's in here are instructions for an 8088 in binary. Some of them don't pass the isprint() test, so they're not shown at all. That's pretty much it for human-readable data.

```
0x9C0 ..1.....l........P.1....l..:...Xt.....RP1.......XZ...r...P1.....
0xA00 l..;.l.u.X.....PQRSUVW...r..).P...P..u..&_^][ZYX.....1..........
0xA40 ...1...........c.......1..................+..U&.L.&..N........
[Lots of Machine Readable and Empty Lines Cut]
Done.
```

Debugging

Got gibberish? Well, you're in the right place. The contents of your EPROM were meant to communicate with a computer, not a human being. It's possible there is nothing readable by humans in there.

I haven't seen any like that so far. Almost all EPROMs you find have some code that was meant to interact with a human being, and that means that there are words in there, even if they're as banal as "Please insert system disk" or "Boot device not found," or messages like that. You should get *something*. If you don't, before you give up on that EPROM, try some debugging. There are a lot of circuits connected in this project, and transposed wires and bad connections are easy to do. (I looked at a lot of gibberish, and I knew there were readable strings on my EPROM.)

If you got no readable characters at all, or certain characters are always wrong, this is usually a data bus problem. Make sure the data lines are all connected, and in the right order. Use the logic probe to check the data pins on the EPROM, then the data pins on the ATmega and make sure you get the same results. If you don't, there's a wiring problem.

If you're sure your data lines are right, go through the ADDR_LSB lines next. If these are wrong, especially if the first two or three bits are wrong, all your characters will be in the wrong order. Put a delay() statement in dump_EPROM's loop, maybe a hundred milliseconds or so. Hook the logic probe up to A0 and run the sketch. A1 should pulse half as fast, A2 half of that, and so on. If that doesn't give you answers, increase the delay to a couple thousand milliseconds to give yourself enough time to test both ends of the address line, from the EPROM end to the ATmega end. If all these lines check out, you should get something readable at some point.

The most puzzling bugs are when you get readable strings and then what are obviously words, but some characters are wrong. These are usually address bus problems. Note the start address of the line the first messed up word occurs on. That will give you some idea what address line changed between the last completely good line and the first mangled one. Check both ends of those address lines while the sketch is running (put delay() calls in if you have to).

If everything clears up when you put delay() into the loop, you've got a loose wire or some other kind of bad connection. Adding resistance and/or capacitance to a circuit introduces a time delay. It could be, if your EPROM is old enough, that even bit-banging the EPROM is too fast for it, but it's not likely. Our 20MHz Cestino does nothing at all in less than 50ns, and there are a lot of software steps going on for each attempt to

read the EPROM. Still, if you get values that don't change from one address to the next and they're in words so they look like they should, add a 1ms delay inside the read loop. The earliest EPROMs I've read about were still 250ns devices. 1000ns (1ms) should give them plenty of time.

Build the Flash Explorer

For those who couldn't find a programmed 27xx series EPROM to read, or whose EPROM turned out to be blank, I offer this version of the project: Flash Explorer, shown in Figure 8-4.

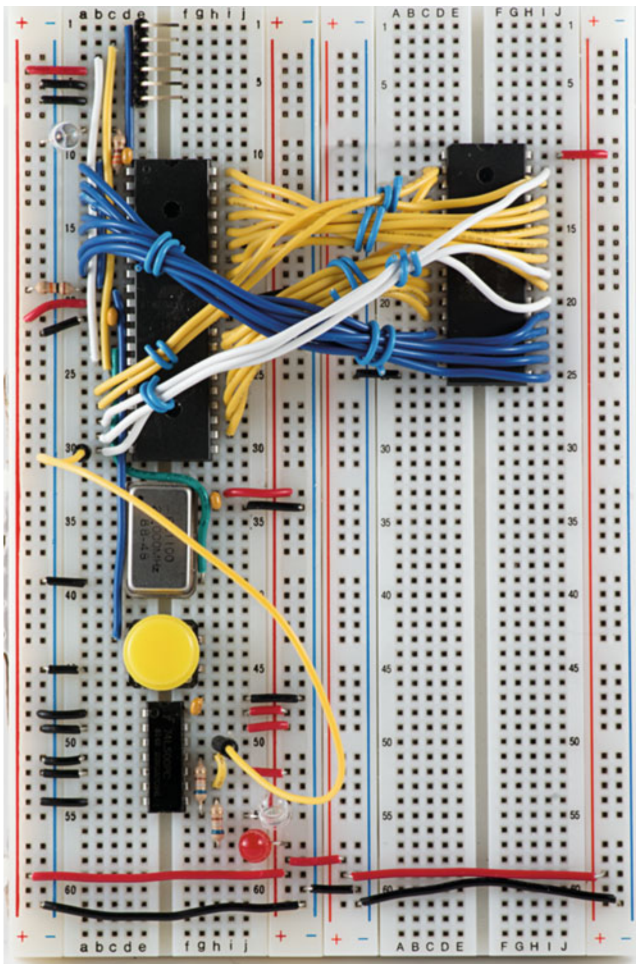

Figure 8-4. *Flash Explorer*

I'm not going to go into as much depth, as reading a flash is exactly the same as reading an EPROM, except that the device is bigger and needs more address lines. We will, however, write to this flash IC, so even if you got a new, blank one from Mouser or Digikey, you can still have some fun with your device. I built mine with a 39SF020A, a common 256k8 device that's still available at the usual suppliers. If you use a different IC, your pinout may be slightly different, although the JEDEC standard was well in place by the time flash chips hit the scene. The command sequence and particularly the registers for writing may be quite different, so you may have to modify the sketch.

Another important difference: the flash device I used has a primitive controller in it. Although you can ignore the controller to read data from the flash just like an EPROM, (the datasheet even says so) erasing and programming the flash device *requires* telling the controller what you want. This adds to what's in the sketch. It's also an indicator of how much later flash ICs like my 39SF020A are than the 27128. It was reasonable for IC designers in 2001 to assume that any device trying to program this flash would have a microprocessor, rather than simple TTL circuits. Twenty years makes a difference.

Power, Ground, and Unused Signals

As with any IC, the power and ground lines are easiest to connect first. If you look at Figure 8-5, you can see that Vcc (power) and Vss (ground) are on pins 32 and 16, respectively.

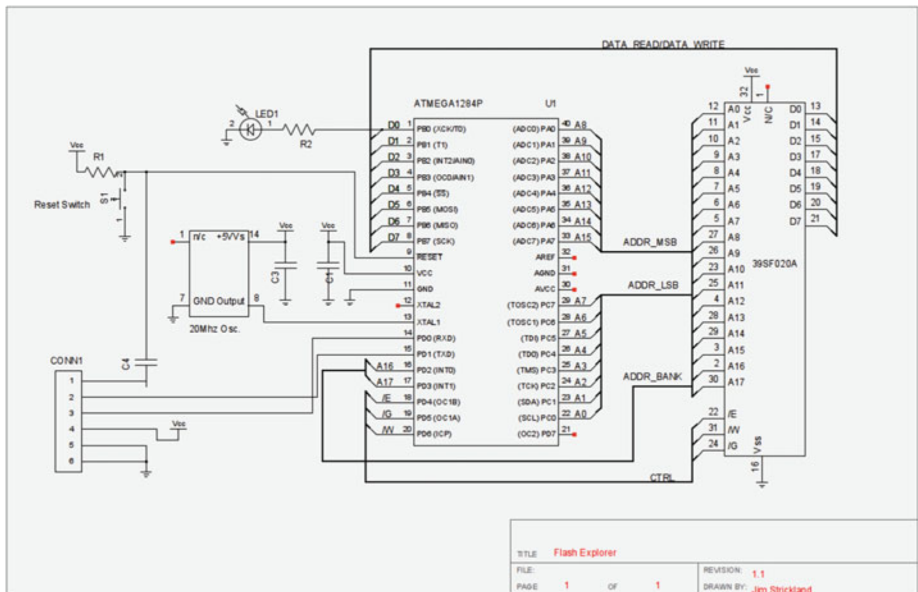

***Figure 8-5.** Flash Explorer Schematic*

It doesn't show in the diagram because the pins aren't in order, but this is the equivilant position they're in on the EPROM, relative to the top and bottom of the IC. With the pin-1 end of the package facing north, Vcc is on the northern most pin on the east side, and Vss is on the southern most pin on the west side of the package, though the flash chip has 32 pins instead of the EPROM's 28. Your JEDEC standard at work. Wire these to the + and – bus, respectively.

Data Bus

Just as with the EPROM explorer, the data pins are D0-D7 (possibly called DQ0-DQ7 on your datasheet), and like on the EPROM, they'll be the three pins north of Vss on the west side of the IC and the southernmost five pins on the east side, with D0 being the lowest pin number and D7 being the highest. This is the JEDEC standard again. Wire these to PB0-PB7 on the ATmega1284P, just like you did with the EPROM.

Address Bus(es)

If you now expect the ADDR_LSB (lowest significant byte of the address bus) to be one pin north of DQ0 and going backward up the west side of the chip, you're wise to the JEDEC standard. That's exactly where they are, in this case from pins 12 to 5 for A0 to A7, respectively. Wire these to PC0-PC7 on your ATmega. That's PORT C, from pin 22 to pin 29.

The JEDEC standard gets more complicated for the ADDR_MSB pins, the most significant byte of the address bus, but even their semirandom scattering is very similar to that of the 27128 we looked at in the EPROM explorer. In order of the ATmega1284's pins (which are PORT A once again,) connect PA0 to Pin 27, PA1 to pin 26, PA2 to pin 23, PA3 to pin 25, PA4 to pin 4, PA5 to pin28, PA6 to pin 29, and PA7 to pin 3.

We now have enough address lines to talk to the first 64k of this 256k flash. If you want to test here, like we did with the EPROM, I won't blame you.

We do have two more address lines we need to find some way of switching, or we won't ever get past 64k in this flash. We're out of completely unused ports, but if we're careful with the sketch, we can go ahead and use the pins in PORT D that aren't part of RS232 communications with the console. We'll call these the ADDR_BANK lines. Wire PD2 and PD3 to flash pins 2 and 30, respectively, for A16 and A17.

A Brief Diversion: Bank Switching

Microprocessors today have dozens of memory control lines, and their 64 bit registers can address all those lines at a time. The flip side of all this luxury is that they're all SMD—Surface Mount Devices—usually in hobbiest-unfriendly packages that assume the whole board will be *baked* in order to solder their pins to the board. In the 1970s and 1980s, it wasn't so. Most 8-bit microprocessors, as we'll see, had 16 address lines, and 16-bit memory registers to address them, which, if you count it out in binary on your fingers, you'll find is 64k. You may, if you're old enough, remember that some computers could address more than that, despite being 8-bit microprocessors. The secret was memory bank switching.

In a nutshell, because we're not really using this technique in this book, memory bank switching was a hardware technique that generated some of the address line signals, and allowed a microprocessor to map different parts of the real memory map in different places of the memory the processor could see. Popular Z80 operating systems broke memory into 16kb banks, and so long as the code controlling the bank switch was not in the bank being switched, you could tell the memory controller to switch another bank in and have more memory, albeit with the same addresses as the bank you switched out. The cost, however, was high. Your program, or at least the operating system, had to keep track of what bank had what data (or program code) in it, and switching banks slowed execution of the program down. Most programmers breathed a sigh of relief when flat memory maps (memory maps without segmentation or banking) became all the rage in microcomputers.

We're not really using normal memory address lines with the ATmega. As I mentioned previously, it hasn't got any on the outside where we can get at them, and we're bit-banging our address signals. But because we've treated ports A and C as our address bus in this chapter, I named the extra two pins ADDR_BANK as an homage to the old days.

Control Bus

In the EPROM, you will recall, we pinned all the control lines to their useful settings and left them there. Because communications with the flash chip are a lot more complicated, we need to make those lines switchable. Because we haven't got any other unused pins on the ATmega1284P, we'll use more of port D's pins. Yes, it will get a little complicated switching port D to do our bidding with all these not-really-related signals tied to it, but it's manageable.

The first signal we'll deal with is /E, which may also be labelled /CE on your datasheet. This is the chip enable. It determines whether the flash pays attention to address information coming in. Wire that to PD4 on the ATmega1284P. We had this signal on the EPROM too, but we just pinned it to the - bus to keep the IC enabled.

The next signal is /G for Gate, sometimes called /OE for Output Enable. Wire it to PD5 on the ATMega1284P. The EPROM had one of these too, and again, we pinned it to the - bus to keep the output enabled. Not so here.

The last signal is /W, for Write, which may be labelled /WE. Once again, the EPROM had this too, but it was useless to us so we pinned it to the + bus. We need that pin in this project, so wire it to pin PD6 on the ATmega1284P.

If your Flash chip is a 512k8 model, you may need to move the signal down one pin, so you can keep your A18 line with A16 and A17.

Congratulations. We've now used all but one GPIO pin on the ATmega1284P.

The Flash_Explorer Sketch

Flash is ... different. Although we can read with the same combination of /CS and /G being low, flash isn't protected from stray data by virtue of requiring up to 48v on its programming pin. Flash is smart. When we want to write to it, we have to send a specific pattern of bytes, chosen by JEDEC as the least likely pattern to occur randomly, to specific locations on the flash (these are registers, like the Cestino's PORT registers), and we have to have our control signals set properly too.

In order to send any bytes to the flash, we first have to tell the flash to listen for them, by setting the /CS signal low, the /G signal high, setting the address and data lines how we want them, and pulsing the /W signal low. The flash will latch (store) our address signals on the falling edge of the /W pulse (as /W goes from +5v to 0v), and it will latch the data signals on the following rising edge, as /W goes from low to high. When we want to read data from the flash again, we have to raise /W and lower /G. We can't get away with just pinning signals low anymore. We have to use them.

We'll use this mechanism to send a command to the flash by sending a specific series of bytes, then send data to write, and we have to do this for each byte we want to write to the flash.

We send a different sequence of bytes to those registers if we want to erase one sector of the flash (which we won't), or we want to erase the whole IC at once (which we will.) It does more, so it's more complicated. Let's get started.

■ **Note** If it works right, this sketch ERASES THE FLASH. If you intend to use the device the flash IC came out of, you may want to disable the set_flash_signals, send_to_flash, program_flash, and erase_flash functions. Use at your own risk.

```
//preprocessor definitions.
#define ADDR_LSB PORTC
#define ADDR_MSB PORTA
#define DATA_READ PINB
#define DATA_WRITE PORTB
#define DATA_DDR DDRB
#define ADDR_BANK_CTRL PORTD
#define MAX_ADDRESS 0x03FFFF
#define FLASH_READ 0b0100
#define FLASH_WRITE 0b0010
#define FLASH_WAIT_WRITE 0b0110
```

The preprocessor definitions are pretty much the same, except for the addition of ADDR_BANK_CTRL, to represent both functions of port D. The max address is much higher, 3FFFF. There are also control bit settings define for FLASH_READ, FLASH_WRITE, and FLASH_WAIT_WRITE.

These are bit patterns to set to the /E, /G, and /W pins to read from the flash, write to the flash's registers (this does not write data to the flash itself, only sends messages to the flash's controller), and a safe state that won't interrupt write sequences between pulses

of the /W signal. What's important about these three words is that they correspond to /W, /G, and /E, in that order, left to right, and that a 1 means the signal is high, or disabled, as these signals are active low. So FLASH_READ is /E enabled, /G enabled, /W disabled. FLASH_WRITE is /E enabled, /G disabled, and /W enabled, and FLASH_WAIT_WRITE is /E enabled, /G disabled, and /W disabled. This will make more sense when we use it in set_flash_signals.

```
String message = "Over hill, over dale, \
Thorough bush, thorough brier,Over park, over pale, \
Thorough flood, thorough fire, I do wander everywhere.\
- Shakespeare";
```

This piece of code defines the message we'll write to the flash, when the time comes. It could be anything. I chose Shakespeare. Note that the backslashes (\) tell C++ that this line continues past the linefeeds.

```
 void set_flash_signals(uint8_t bits) {
  uint8_t temp = ADDR_BANK_CTRL & 0b00001111;
  temp |= (bits << 4);
  ADDR_BANK_CTRL = temp;
}
```

As I mentioned earlier, the 39SF020A requires that the /E and /W signals be enabled (low) and that /G be disabled (high) to send bytes and addresses to the flash. Furthermore, it requires that /G *never* go low during the write, or the write will abort. This is to protect the flash while the system it's attached to is powered up. If a random control signal pattern happened to hit the write settings during powerup, a random message could be sent to the flash controller, and the flash could wind up in the wrong mode to be read from. Instead, the 39SF020A will abort the write if the signals aren't set right through the entire process. Much different from our rather relaxed EPROMs. To handle this, we have the set_flash_signals() function.

First, we store ADDR_BANK_CTRL, masked with 0b00001111, into temp. This stores all the address settings and protects the RS232 lines from our code, but clears all the control signals. Placing this value in temp rather than ADDR_BANK_CTRL directly ensures that /G is not enabled during the write, even for a few dozen nanoseconds.

Next, we rotate the bits argument left by four spaces, and OR temp with them. This means that the three bit setting that was passed in the bits parameter now occupies bits 4-6, where our control signals are.

Finally, we set ADDR_BANK_CTRL to this new computed value, and we're done.

```
void select_flash_address(uint32_t flash_address) {
  uint8_t temp;

  ADDR_LSB = 0;
  ADDR_MSB = 0;
  ADDR_BANK_CTRL &= 0b11110011;
```

```
  // Initialize address registers for safety.

  union {
    uint32_t address_uint;
    byte byte_array[4];
  } address_union;

  address_union.address_uint = flash_address;
  ADDR_LSB = address_union.byte_array[0];
  ADDR_MSB = address_union.byte_array[1];
  temp = ADDR_BANK_CTRL;
  temp &= 0b11110011;
  temp |= (address_union.byte_array[2] << 2);
  ADDR_BANK_CTRL = temp;
}
```

Select_flash_address works in much the same way select_EPROM_address did, save that it takes a uint32_t, an unsigned 32 bit integer as its argument, and the union breaks it into an array of four bytes instead of two. The fourth byte is unused, as are all but two bits of the third byte. Select_flash_address is also a lot more paranoid about initializing address registers for safety.

To set A16 and A17, we do the same kind of thing we did in set_flash_signals(). Set temp to ADDR_BANK_CTRL, and temp with 0b11110011 to preserve all the bits of ADDR_BANK_CTRL except our two address bits, which we zero out, rotate address_union.byte_array[2] left by two bits and or the result with temp, then set ADDR_BANK_CTRL to temp. This ensures that ADDR_BANK_CTRL does not change until we've computed its new value completely.

```
void send_to_flash(uint32_t address, byte data) {

  select_flash_address(address);
  DATA_WRITE = data;

  set_flash_signals(FLASH_WRITE);
  set_flash_signals(FLASH_WAIT_WRITE);
}
```

The send_to_flash() function sends bytes and addresses from the sketch to the flash IC's controller. It doesn't write data to the flash's memory. That's the job of program_flash(). It's a communication routine. It's simple, and fast.

The first thing we do is set the address lines using the select_flash_address function. We set the data lines directly to the DATA_WRITE register, and then we do a curious thing. We call set_flash_signals with FLASH_WRITE, setting /W low, and then set it right back to FLASH_WAIT_WRITE, which sets /W high again without changing /G. If you look at the datasheet for the SST 39SF020A, you'll see that the /W (/WE, really. It means the same thing) pulse width has to be a minimum of 40ns (nano-seconds) long. Because

our Cestino runs at 20MHz, each of its clock pulses is 50ns long, so even with these two commands back to back, there's no way for the Cestino to generate a pulse that is too short. If you use a different, older, slower flash here, you may need to insert a delay.

You may have noticed that this function doesn't set the flash signals to a known state on the way in. This needs to be handled by the calling function.

```
void program_flash(uint32_t address, byte data) {
  DATA_DDR = 0xFFFF;
  set_flash_signals(FLASH_WAIT_WRITE);

  send_to_flash(0x5555, 0xAA);
  send_to_flash(0x2AAA, 0x55);
  send_to_flash(0x5555, 0xA0);

  send_to_flash(address, data);

  DATA_WRITE = 0;
  DATA_DDR = 0x0;
  delay(1);

  set_flash_signals(FLASH_READ);
  //Set the flash signals back to FLASH_READ mode.
}
```

Program_flash is not a complicated function, but it has to be exactly right, or we won't successfully program data to the flash IC. This particular version works with the 39SF020A, but read *your* datasheet and get the commands for your flash IC.

Program_flash sets the DATA_DDR to write mode, then sets the flash signals to FLASH_WAIT_WRITE. This turns /G off (high), turns /CE on. It then sends three values to three specific registers on the flash. These aren't stored. They're intercepted by the flash controller, and interpreted to mean "program the next data you get to the next address you get. So we send that data to that address. Programming flash takes some time. 30µs, to be exact, so I put a 1ms delay (1000µs) to take care of it.

```
void erase_flash() {
  DATA_DDR = 0xFFFF;
  set_flash_signals(FLASH_WAIT_WRITE);

  // for 39SF020A
  send_to_flash(0x5555, 0xAA);
  send_to_flash(0x2AAA, 0x55);
  send_to_flash(0x5555, 0x80);
  send_to_flash(0x5555, 0xAA);
  send_to_flash(0x2AAA, 0x55);
  send_to_flash(0x5555, 0x10);
  select_flash_address(0);
```

```
  DATA_WRITE = 0;
  DATA_DDR = 0x0;
  set_flash_signals(FLASH_READ);
}
```

Erase_flash works a lot like program_flash. It sets the DDR on the DATA port to write, sets the flash signals to FLASH_WAIT_WRITE, then sends a command sequence to the flash IC's controller, in this case one that's six bytes long to six specific registers. Again, check your data sheet for the right commands for your IC.

Erasing the flash takes time, about 20ms, according to the datasheet. This function doesn't do that, so we have to make sure we wait after calling it. Your wait may be longer if you use a different flash IC.

```
void dump_flash(uint32_t start_addr, uint32_t end_addr ) {
  char address[12];
  uint32_t c;
  DATA_DDR = 0x0;
  set_flash_signals(FLASH_READ);

  sprintf(address, "0x%06lX :", start_addr);
  String data_line = String(address);

  for (c = start_addr; c <= end_addr; c++) {
    select_flash_address(c);//iterate through all the addresses

    if ((!(c % 0x40)) && c > 0) { //if we're at 64 characters
      Serial.println(data_line);
      sprintf(address, "0x%06lX :", c);
      data_line = String(address);
    }

    if (isprint(DATA_READ)) { //if the data on DATA_READ is printable
      data_line += (char)DATA_READ; //add it.
    } else {
      data_line += "."; //otherwise add a period.
    }
  }

  Serial.println(data_line);
```

You may notice that I've changed dump_flash somewhat from its ancestor, dump_eprom. Formatting lines so they line up is often an exercise in frustration with Arduino, because Arduino and/or the Arduino serial console can't handle tabs correctly. To that end, I've used sprintf(), an old school C formatted output tool, to get the addresses all the same length. Sprintf doesn't know anything about the String object, so the first thing we do is set it up an array of 12 chars for it to put its output in so we can get at it. After that, we create a uint32_t (32 bit unsigned integer) called c, which will be the index of our big

loop and hold the address we want to read, set the DATA_DDR so the DATA port is in read mode, and call set_flash_signals with FLASH_READ to put the flash in read mode. Reading is just like it is with EPROMs. We can just set the signals and go.

Next, we generate the beginning of each line by calling sprintf() with a pattern 0x%06lX, followed by a space and a colon. There's a mixture of stuff in here. 0x is literal, so our addresses are properly denoted as hex addresses. You'll get the 0x in your output. %06yadda tells sprintf to make every address six digits long, and to prepend zeros as required. The l tells sprintf to expect a long integer, and X tells it to output the address in hex. The final space and colon are also literal, and will be in the output. Short version: sprintf formats start_addr into numbers that look like this: 0x000001: and puts that value into its 12 character array. We promptly put that character array into a String object called data_line. There. We have the first line of the dump.

Next, we iterate through all the addresses from start_addr to end_addr, incrementing c each time. We call select_start_address with c as its argument.

Then we check to see if our data line is full, at 0x40 or 64 items of data. If it is, we Serial.println data_line, then go through the sprintf process again, this time with c as the argument, and set data_line to it, starting a new data line.

No matter what's happened prior, either the creation of a new data line, or nothing, the next thing we do is see if the DATA_READ register is a printable character with the c isprint function. If it is, we append the value of DATA_READ, cast as a char (character) to the data_line string. If it's not, we append a period to hold the space.

When we reach the end of our loop we print data_line, to ensure that even if our end_addr was not on a 64-bit boundary, we get our data.

```
void setup() {
  Serial.begin(115200);

  DDRC = 0b11111111;
  DDRA = 0b11111111;
  DDRD = DDRD | 0b11111100;
  ADDR_BANK_CTRL = FLASH_BUS_READ;
  Serial.println("\r\rRunning");
}
```

Setup is very similar to the one in EPROM_Explorer. It does configure port D for its role as ADDR_BANK_CTRL, leaving the last two bits set however they were, and it sets ADDR_BANK_CTRL to FLASH_BUS_READ. Yes, this clobbers RS232 communications. Briefly. Once. We can live with it. It then sends us a message to tell us we're running.

```
void loop() {
  Serial.println("Dumping Flash.");
  dump_flash(0x00, MAX_ADDRESS);
  Serial.println("Erasing Flash.");
  erase_flash();
  delay(100);
  Serial.println("Done Erasing Flash.");
```

```
  Serial.println("Programming Flash.");

  for (int c = 0; c <= message.length(); c++) {
    program_flash(c, message.charAt(c));
  }
  Serial.println("Dumping Flash Again.");
  dump_flash(0x00, 0xff);
  Serial.println("\r\rDone.");
  while (0 == 0) {};
}
```

As with EPROM_Explorer, loop() exists to call the functions we've built. It dumps the entire flash from 0 to MAX_ADDRESS, then wipes the flash. It then waits 100ms to allow the flash to complete its erasure, then programs the first few dozen bytes of the flash with the Shakespeare quote we set in message, above, then dumps the first 64 bytes of the flash to show that we actually got a successful write.

As always, the entire sketch is here, for your convenience.

```
// Flash_Explorer v2.0
// Revised to work properly with 39SF020A.
//-------------------------------------------------------------
// This sketch dumps the contents of the flash (an old BIOS
// flash in this case), then erases it and programs a bit of
// Shakespeare at the beginning, then dumps the first 255
// bytes of the flash again to show that it's been erased
// and reprogrammed.
// ---------------------------------------------------------
//Hardware:
//For a 39SF020A or similar flash IC:
//ATmega Pin    Name   Port name Flash Pin  Name   Func
//         1    PB0    DATA_READ        13  D0     DATA
//         2    PB1    DATA_READ        14  D1     DATA
//         3    PB2    DATA_READ        15  D2     DATA
//         4    PB3    DATA_READ        17  D3     DATA
//         5    PB4    DATA_READ        18  D4     DATA
//         6    PB5    DATA_READ        19  D5     DATA
//         7    PB6    DATA_READ        20  D6     DATA
//         8    PB7    DATA_READ        21  D7     DATA
//
//        16    PD2    BANK_ADDR        02  A16    ADDRESS
//        17    PD3    BANK_ADDR        30  A17    ADDRESS
//        18    PD4    CTRL             22  /E     Chip Enable
//        19    PD5    CTRL             24  /G     Gate
//        20    PD6    CTRL             31  /W     Write Enable
//
```

```
//        22    PC0    ADDR_LSB         12  A0     ADDRESS
//        23    PC1    ADDR_LSB         11  A1     ADDRESS
//        24    PC2    ADDR_LSB         10  A2     ADDRESS
//        25    PC3    ADDR_LSB         09  A3     ADDRESS
//        26    PC4    ADDR_LSB         08  A4     ADDRESS
//        27    PC5    ADDR_LSB         07  A5     ADDRESS
//        28    PC6    ADDR_LSB         06  A6     ADDRESS
//        29    PC7    ADDR_LSB         05  A7     ADDRESS
//
//        33    PA7    ADDR_MSB         03  A15    ADDRESS
//        34    PA6    ADDR_MSB         29  A14    ADDRESS
//        35    PA5    ADDR_MSB         28  A13    ADDRESS
//        36    PA4    ADDR_MSB         04  A12    ADDRESS
//        37    PA3    ADDR_MSB         25  A11    ADDRESS
//        38    PA2    ADDR_MSB         23  A10    ADDRESS
//        39    PA1    ADDR_MSB         26  A09    ADDRESS
//        40    PA0    ADDR_MSB         27  A08    ADDRESS

// ----------------------------------------------------------
// James R. Strickland
// ----------------------------------------------------------

//preprocessor definitions.
#define ADDR_LSB PORTC
#define ADDR_MSB PORTA
#define DATA_READ PINB
#define DATA_WRITE PORTB
#define DATA_DDR DDRB
#define ADDR_BANK_CTRL PORTD
#define MAX_ADDRESS 0x03FFFF
#define FLASH_READ 0b0100
#define FLASH_WRITE 0b0010
#define FLASH_WAIT_WRITE 0b0110

// This is the message we'll be programming into the flash.
// You can change it if you want. The \ marks mean "line
// continues after the line break."
String message = "Over hill, over dale, \
Thorough bush, thorough brier,Over park, over pale, \
Thorough flood, thorough fire, I do wander everywhere.\
- Shakespeare";

// ----------------------------------------------------------
// set_flash_signals(uint8_t bits)
// ----------------------------------------------------------
```

```
// This function sets the signal bits of ADDR_BANK_CTRL (Bits
// 4-6) to the value of the first three bits passed in the
// bits field. ADDR_BANK_CTRL is a port register and the state
// of the signals on it needs to not have unknown transitional
// states, so we compute the new byte with temp and set the
// final value to ADDR_BANK_CTRL.
// ----------------------------------------------------------
void set_flash_signals(uint8_t bits) {
  uint8_t temp = ADDR_BANK_CTRL & 0b00001111;
  //mask the control bits off of ADDR_BANK_CTRL
  //and store the result in temp.

  temp |= (bits << 4);
  // OR temp with bits, rotated left by 4 places.

  ADDR_BANK_CTRL = temp;
  //set ADDR_BANK_CONTROL to the computed value.
}
// ----------------------------------------------------------
// select_flash_address(address)
// ----------------------------------------------------------
// This function sets ADDR_LSB, ADDR_MSB, and the first two
// bits of ADDR_BANK_CTRL to the uint32_t (four byte unsigned
// int) address passed to it. These registers are wired to the
// flash IC's address pins.
//
// To break the address down into single byte chunks, we use
// a union, which essentially maps two variables (a 4 byte
// unsigned int and an array of 4 bytes, in this case) to
// one area of memory. Then we can peel off the bytes as we
// want them. Note that this code is endian-sensitive. If
// you run it on some future big-endian Arduino, you'll need
// to change the byte order.
//
// We have to extract the lowest two bits of the third byte
// and rotate them two spaces to the left, then AND them in
// to the ADDR_BANK_CTRL register, since we don't want to
// disturb the signal lines or the RS232 lines.
// ----------------------------------------------------------

void select_flash_address(uint32_t flash_address) {
  uint8_t temp;
```

```
  ADDR_LSB = 0;
  ADDR_MSB = 0;
  ADDR_BANK_CTRL &= 0b11110011;
  // Initialize address registers for safety.

  union {
    uint32_t address_uint;
    byte byte_array[4];
  } address_union;
  //Declare the union we're going to use for address processing.

  address_union.address_uint = flash_address;
  //Set address_union.address_uint to the address we were given.

  ADDR_LSB = address_union.byte_array[0];
  //set the ADDR_LSB register to the first byte of the union.

  ADDR_MSB = address_union.byte_array[1];
  //set the ADDR_MSB register to the second byte of the union.

  temp = ADDR_BANK_CTRL;
  temp &= 0b11110011;
  temp |= (address_union.byte_array[2] << 2);
  ADDR_BANK_CTRL = temp;
  // set bits 3 and 4 of ADDR_BANK_CTRL to the first two bits
  // of the third byte in the union.
}

// -------------------------------------------------------------
// send_to_flash(address,data)
// -------------------------------------------------------------
// This function sets the ATmega's data port to write mode selects
// the address pin settings for the flash IC, and writes (data) to
// the data port. We use it to pass commands to the flash IC.
//
// Note that this does not directly write to the flash's storage.
// This function actually talks to the flash's built-in controller,
// but no data is written (programmed) to the flash without passing
// the flash a specific sequence of commands.
//
// NOTE BENE: This function assumes that the flash signals will be
// set to FLASH_WAIT_WRITE when it enters. This must be handled by
// the calling function, since changing /OE (gate) between
// transmitted bytes aborts command sequences.
//
```

```
// First we set the address lines to the address by calling
// select_flash_address().
// Then we set the data lines to the data we're given.
// Then we pulse the /W signal low by calling set_flash_signals with
// FLASH_WRITE, and then with FLASH_WAIT_WRITE.
// And then we're done.
// ----------------------------------------------------------
void send_to_flash(uint32_t address, byte data) {

  select_flash_address(address); //Set the address lines
  DATA_WRITE = data; //Set the data lines

  set_flash_signals(FLASH_WRITE);
  set_flash_signals(FLASH_WAIT_WRITE);
  //pulse /W low. It only has to go low a few nanoseconds.
  // Nothing on the cestino happens in less than 50, so we're
  // not going too fast for the 85ns 39SF020A. If your flash
  // is slower, you may have to add a delay.
}

// ----------------------------------------------------------
// program_flash(address,data)
// ----------------------------------------------------------
// To actually store (program) data onto the flash, we have to
// send it a specific pattern of addresses and data bytes before
// we send it the address we want and the byte of data we want
// stored there.
// To that end, we set DATA_DDR to output so we can write data,
// then call set_flash_signals to set it to FLASH_WAIT_WRITE
// mode. This disables /OE (gate) but does not enable /W (write).
// We call send_to_flash four times. The first three tell the
// flash controller what we want to do, and the fourth call
// gives it our address and data.
// Then clear the DATA_WRITE pins and set DATA_DDR back to
// read mode. Delay 1ms to because flash is slow and we're
// not polling its status line to tell when it's done. Set the
// flash's control signals to FLASH_READ mode and we're done.
// ----------------------------------------------------------
void program_flash(uint32_t address, byte data) {
  DATA_DDR = 0xFFFF;
  set_flash_signals(FLASH_WAIT_WRITE);

  //for 39F0020. These may be different for other flash ICs.
  send_to_flash(0x5555, 0xAA); //Tell flash to store data
  send_to_flash(0x2AAA, 0x55);
  send_to_flash(0x5555, 0xA0);
```

```
  send_to_flash(address, data); //Give flash our data and address.

  DATA_WRITE = 0; //clear the DATA_WRITE pins.
  DATA_DDR = 0x0; //set DATA_DDR back to read mode.
  delay(1); //Delay one ms because flash is slow.
  //We could probably save a few microseconds monitoring the
  //status bit, but that'd be a lot more code.

  set_flash_signals(FLASH_READ);
  //Set the flash signals back to FLASH_READ mode.
}

// -----------------------------------------------------------
// erase_flash()
// -----------------------------------------------------------
// If you guessed that erasing the entire flash would be another
// sequence of commands sent to specific addresses, you guessed
// right. That's what this function does.
//
// Set DATA_DDR to write mode, and set the flash's signals to
// FLASH_WAIT_WRTTE - which is /W disabled, /OE disabled, and
// /CE enabled. Call send_to_flash 6 times to give it the
// command sequence to erase the entire flash.
// Then clear DATA_WRITE, set DATA_DDR to read mode, and
// set the flash's signals back to FLASH_READ.
//
// NOTE BENE: Erasing the flash takes a few ms, and this
// function doesn't include the wait, so the calling function
// needs to do it. The flash returns gibberish if you try to
// read it while it's erasing. It goes without saying that
// writes fail, too.
// -----------------------------------------------------------
void erase_flash() {
  DATA_DDR = 0xFFFF;
  set_flash_signals(FLASH_WAIT_WRITE);

  // for 39SF020A
  send_to_flash(0x5555, 0xAA);
  send_to_flash(0x2AAA, 0x55);
  send_to_flash(0x5555, 0x80);
  send_to_flash(0x5555, 0xAA);
  send_to_flash(0x2AAA, 0x55);
  send_to_flash(0x5555, 0x10);
  select_flash_address(0);
  //These are specific to the 39SF020A. Other flash chips have
  //different registers that are similarly unlikely patterns
  //to occur by accident.
```

```
  DATA_WRITE = 0;
  DATA_DDR = 0x0;
  set_flash_signals(FLASH_READ);
  //Clear data_write and set DATA_DDR back to read mode. Then
  //set the flash signals back to FLASH_READ mode.
}

// -------------------------------------------------------------
// dump_flash(start_addr,end_addr)
// -------------------------------------------------------------
// This function dumps the contents of the flash to the console.
// data_line is an Arduino String object. We initialize it empty.
// Iterate through all the addresses, start_addr to end_addr,
// inclusive.
//
// Start by creating a char array for the line's address, a
// uint32_t called c to iterate through the addresses with,
// then set DATA_DDR to read mode, and set the flash's signals
// to FLASH_READ mode.
//
// We're going to use the sprintf function to properly format our
// addresses, since Arduino's tabs don't work properly. Sprintf
// takes a pattern of text and variable display information,
// and fills the variables in based on that pattern. In this case,
// the pattern is 0x%06lx. The first two characters, 0x, are
// literal. You'll see them in the output. %06lx tells sprintf
// "this is a 6 digit number. Fill in preceding zeros if you
// need to. The variable will be a long integer (32 bits in
// the Arduino implimentation we're using), and it should be
// displayed in hexidecimal.
//
// The normal C printf would send that to the console, but we'd
// like to put it in a String object. To do that, we use sprintf,
// pass it the char array address[12] Once it's there, we pull
// it into the String object data_line, where it makes up the
// beginning of the line.
//
// After that, we iterate through the address range we're given,
// and add the character value of DATA_READ to the string if it's
// printable, and a . if it's not. When data_line is 64 characters
// long, we serial.println it, clear it, and set its beginning
// the same way we did before, then get back to work.
// When we're done, we serial.println data_line once more to
// get any data that might have been in an incomplete data line.
// -------------------------------------------------------------
void dump_flash(uint32_t start_addr, uint32_t end_addr ) {
  char address[12];
  uint32_t c;
```

```
  DATA_DDR = 0x0;
  set_flash_signals(FLASH_READ);

  sprintf(address, "0x%06lX :", start_addr);
  String data_line = String(address);

  for (c = start_addr; c <= end_addr; c++) {
    select_flash_address(c);//iterate through all the addresses

    if ((!(c % 0x40)) && c > 0) { //if we're at 64 characters
      Serial.println(data_line);
      sprintf(address, "0x%06lX :", c);
      data_line = String(address);
    }

    if (isprint(DATA_READ)) { //if the data on DATA_READ is printable
      data_line += (char)DATA_READ; //add it.
    } else {
      data_line += "."; //otherwise add a period.
    }
  }

  Serial.println(data_line);

}

// ------------------------------------------------------------
// setup()
// ------------------------------------------------------------
// Set up serial and port data directions. Tell the user we're
// running. Runs once.
// ------------------------------------------------------------

void setup() {
  Serial.begin(115200);

  DDRC = 0b11111111; //ADDR_LSB - Address LSB Port
  DDRA = 0b11111111; //ADDR_MSB - Address MSB Port
  DDRD |= 0b11111100; //ADDR_BANK_CTRL Bank address and control port.
  Serial.println("\r\rRunning");
}

// ------------------------------------------------------------
// loop()
// ------------------------------------------------------------
// Mostly we just call functions here.
// Dump the entire flash
// Erase the flash
```

```
// Program the flash.
// Dump the first 64 characters of the flash again so we can
// see what we programmed. Then do nothing forever.
// ----------------------------------------------------------
void loop() {
  Serial.println("Dumping Flash.");
  dump_flash(0x00, MAX_ADDRESS); //Dump the entire flash.
  Serial.println("Erasing Flash.");
  erase_flash(); //wipe the flash.
  delay(100);
  Serial.println("Done Erasing Flash.");

  Serial.println("Programming Flash.");

  for (int c = 0; c <= message.length(); c++) {
    program_flash(c, message.charAt(c));
  }
  Serial.println("Dumping Flash Again.");
  dump_flash(0x00, 0xff);
  Serial.println("\r\rDone.");
  while (0 == 0) {};
}
```

Output

I used a 39SF020A BIOS flash I made. Here's what's on it.

```
Running
Dumping Flash.
0x000000 :..Xi 8088 BIOS, Version 0.8. Copyright (C) 2010 - 2012 Sergey Ki
0x000040 :selev..Distributed under the terms of the GNU General Public Lic
0x000080 :ense.....none.: .; .Main Processor:       .Mathematics Co-
0x0000C0 :processor:  .Intel 8088 '78..WARNING: This CPU does not disable
0x000100 : interrupts after loading segment registers!...Intel 8088 '81 or
0x000140 : later, or older Intel 80C88...Harris / Intersil / newer Intel 8
0x000180 :0C88...NEC V20...Intel 8087...Display Adapter Type:    .EGA/V
0x0001C0 :GA (Video BIOS Present)...CGA...MDA or Hercules...Floppy disk dr
0x000200 :ives:     Drive 0: .; Drive 1: .360 KB, 5.25".1.2 MB, 5.25".
0x000240 :720 KB, 3.5".1.44 MB, 3.5".2.88 MB, 3.5".PS/2 Aux Device (Mouse)
0x000280 ::  .Present...Absent...Serial Ports:        .COM.Parall
0x0002C0 :el Ports:       .LPT.Testing RAM (ESC to skip): ...ERROR:
0x000300 : Faulty memory detected at ..Total Conventional RAM:   .Reserv
0x000340 :ed for EBDA:     .Available Conventional RAM: . KiB...Boot
0x000380 :failed, press any key to try again......No ROM BASIC...Found BIO
0x0003C0 :S extension ROM at .0, initializing......Booting OS......ERROR:
0x000400 :RTC battery is bad...ERROR: NVRAM checksum is invalid, loading d
```

```
0x000440 :efault values to NVRAM...Press F1 to run NVRAM setup..........NV
0x000480 :RAM Setup Menu:..f - Change first floppy type..g - Change second
0x0004C0 : floppy type..p - Print current settings..w - Save changes and e
0x000500 :xit..q - Exit without saving changes....Enter your selection: ..
0x000540 :.Floppy Setup Menu:..0 - No floppy..1 - 360 KB, 5.25"..2 - 1.2 M
0x000580 :B, 5.24"..3 - 720 KB, 3.5"..4 - 1.44 MB, 3.5"..6 - 2.88 MB, 3.5"
0x0005C0 :..q - Return to the main menu....Enter your selection: ...$.2.@.
0x000600 :M...[....6ff~ff.|``|ff|.|ff|ff|.~```````.8lllll..~``|``~...~<~...
0x000640 :<f...f<.ffn~vff.<fn~vff.flxpxlf..6fffff.........fff~fff.<fffff<.
```

[Lots of machine readable code and empty lines cut]

This is what a PC BIOS looks like. Once again, it's an open source BIOS that I burned there with my EPROM/flash burner. Yours will probably have much sterner copyright warnings.

```
0x0062C0 :...P...?..a.....a.@u....a.$V....0.^....<it.<It.<ru..e.<Ru..^....
0x006300 :....X.IOCHK NMI detected. Type 'i' to ignore IOCHK NMIs, or 'r'
0x006340 :to reboot.......................................................
0x006380 :................................................................
```

[Lots of machine readable lines cut]

```
0x007F00 :...................................P..@...... .. ....u......!...
0x007F40 :!. . .&k..X..........PSQR..@....>...tj.................t.....<v.
0x007F80 :......R...Q.uH...J.uA............< s.. .3.u*..8.r....&.u......u.
0x007FC0 :..8.r......Z.....ZY[X........R1.....Z..%.........[...12/26/12 ..
```

[Many, many empty lines cut]

```
Erasing Flash.
Done Erasing Flash
```

We've erased the flash.

```
0x03FFC0 :................................................................
Erasing Flash.
Done Erasing Flash.
Programming Flash.
Dumping Flash Again.
0x000000 :Over hill, over dale, Thorough bush, thorough brier,Over park, o
0x000040 :ver pale, Thorough flood, thorough fire, I do wander everywhere.
0x000080 :- Shakespeare...................................................
0x0000C0 :................................................................

Done.
```

The Shakespeare quote got written. We can see it in the dump here. It's from *A Midsummer Night's Dream*.

Credit Where Credit Is Due

I'm certainly not the first person to read EPROMs and read or write flash from an Arduino. It's one of those standard projects that pops up every couple years when someone (like me) does it from scratch again. I saw those projects. I knew it could be done.

The hanked breadboard wiring technique is something I cribbed just recently from here: `http://forum.6502.org/viewtopic.php?f=4&t=3329` This guy's doing amazing (and fast) projects on massive breadboards, and he's getting them to work, despite the naysayers.

The XT-IDE Universal Bios, which I burned onto the 27128 and then showed in the Output section of the EPROM explorer is here: `https://www.lo-tech.co.uk/wiki/XTIDE_Universal_BIOS`

Sergey Malinov's Xi (PC XT clone) BIOS is here, at `http://www.malinov.com/Home/sergeys-projects/sergey-s-xt#TOC-What-is-required-to-build-a-functioning-computer`- all the way at the bottom in the files marked "bios-0.x.x.tar.gz". It's a complete PC XT BIOS for Sergey's Xi PC-on-a-board project, and it's open source. Awesome.

Further

The obvious place to go with this project would be a general purpose programmer for EPROMs, EEPROMs, and flash. There are a few problems that you'd need to solve to get there.

First: we have not in any way guaranteed the integrity of the data we're reading and writing to the device. For looking at strings, that's not a big problem. For machine readable data (software), it's critical. Fortunately, this is a software problem. My thought here is to reach back to 1980 and dig up Kermit. Not the one Disney owns. The other one. Kermit is a data transfer protocol written by Frank DaCruise in 1980 to transfer information from microcomputers to the minicomputers most colleges had by that point. It is designed for noisy phone modem/RS232 connections not unlike the Cestino's RS232 to USB connection, it is well documented. It is simple enough to implement on even the smallest microcomputer platforms of the day, which were far less capable than the Cestino. It's been ported to a huge variety of platforms including most modern desktop OSs and is part of many other terminal emulation packages. Best of all, it's open source.

The next problem would be a place to store the binary to be burned onto the device in question, be it EPROM, flash, or whatever. You could add an I2C flash to the Cestino fairly easily, but it would take two pins out of PORT C, which would complicate the design. You could add an SD card interface using SPI, which would take three pins out of port B. None of these solutions are unworkable. The sketch would simply have to be careful about when it reads, when it writes, and to return the pins to a known state, just as we did with PD0 and PD1 in this project. You could add an IO expander, like the MPC23S18 that could give you 16 IO lines in exchange for the same three SPI lines plus a chip select line for each SPI device. It would complicate the sketch further, but not that badly. It might be slower, but we're not in that big a hurry.

The most complicated part, for me at least, would be the power supply needed for the various programming voltages. I would probably start with a standard ATX power supply. These are plentiful in my junk box, and people keep giving me more. They produce some, but not all, of the needed voltages, and plenty of current to convert to the rest. It wouldn't need to be the most modern or largest to have plenty of resources for an EPROM/flash programmer.

Connecting the correct programming voltages to the correct pins is another complicated problem. The solution in my head is to bring all the board's lines out to a band of pin headers, and then have a separate configuration board that connects the pin headers together, one for each type of IC you intend to program, and here the JEDEC standard helps a lot. You could have a single configuration board for all JEDEC 28 pin EPROMs, one for all 32 pin flashes, and so on. Old fashioned? Sure. My Ebay EPROM/ Flash programmer does all that in software. Will it work even after the platforms it was designed to work with are themselves junkbox refugees? Yes.

CHAPTER 9

ATA Explorer

This chapter's project is about connecting a hard drive to your Cestino. Figure 9-1 shows what our final result looks like. Do you have an old drive from back in the day? This will be a fun chapter. You'll enjoy it.

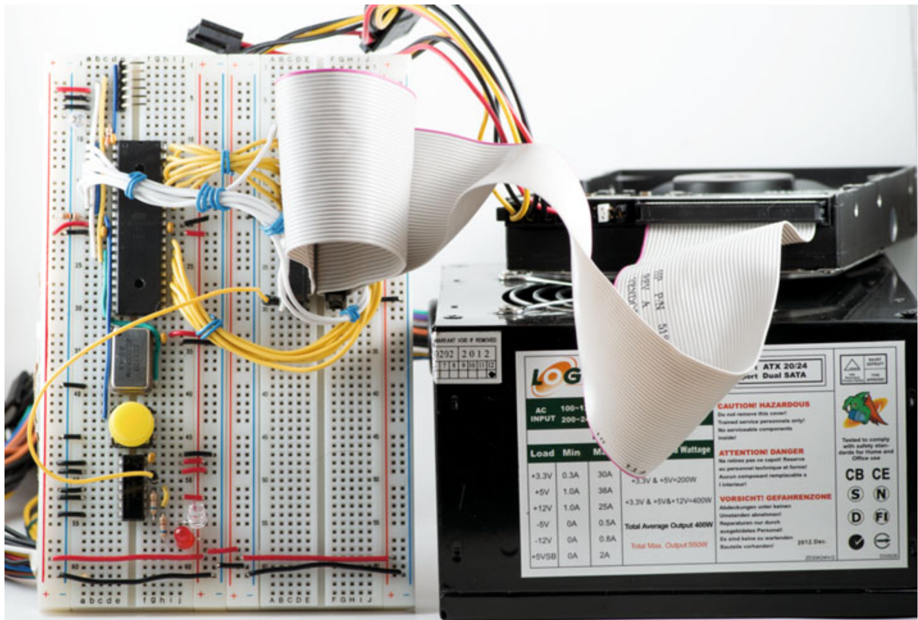

Figure 9-1. *ATA Explorer*

One day, perhaps a decade or two out, the last manufacturer will quietly stop making mechanical hard drives, and the era of rotating media will draw to a close. What was once a glorious era of whirring, whining, and clicking drives will fade away, leaving only silent, fast, and efficient electronic storage. Floppies are gone. DVD-Rom is slowly being eclipsed by SD cards. And hard drives? The writing is on the wall for them too. The most modern "drives" connect directly to the PCI-E bus in your computer or to USB 3.0, and

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_9

the various artifacts of drives are implimented in firmware to keep operating systems and software happy.

The demise of mechanical drives (and drives of any kind) is not a bad thing for us. After all. The drives wind up in the junk box. They might have sensitive data on them, and having them properly destroyed is a nuisance. Let's have some fun with one.

If your junk box is like mine, the oldest drives in it are what we now call Parallel ATA, and used to call IDE: drives with a 40 pin ribbon connector in the back. That's good. They're the ones we can connect to the Cestino and access, despite our rather pokey logic speeds and sloppy timing.

The Stuff You Need

This project is more complex than most of the previous, so it has more parts, including several that won't go on the breadboard.

New Parts

40 pins worth of extra long pin headers

These are available at Adafruit or Schmartboard.com. Make sure they're the kind with equal lengths of pin on both sides of the plastic divider, just like we used on the Cestino itself for the TTL-232 connector. Regular pin headers won't work. You'll have to cut one of the rows in half and take out one pin for the key pin. If you have scraps of pin header with only a few pins each, you can combine these and they'll work.

■ **Note** You might be tempted to use one of those nifty breakout boards for modern 40 pin Raspberry Pis as a convenient way to connect the 40 pin PATA cable to your breadboard. Unfortunately, the various ground and power busses of the Pi are tied together in the breakout boards, and you wind up with the whole Cestino board shorted out. Alas.

Used Parts

A 40 pin PATA cable

Look closely. If it has only 32 pins and has a twist in some of the wires? That's a floppy cable. A PATA cable will have no twists. It may have 80 conductors, but both connectors will have only 40 pins. (The other conductors are wired to existing grounds and are used to kill noise and crosstalk between the wires of the 40 pin standard.) It must also have two drive connectors and one motherboard connector, all of them exactly the same. Most important, the drive connectors should /not/ say "Master," "Slave," "Drive 0," or "Drive 1" on them. If they do, you may have a cable that wants to select your drive ID for you. The kind we need is absolutely the most common: a piece of straight-through ribbon cable with three 40 pin connectors on it, wired in parallel. We'll be using this cable to connect the drive to our breadboard.

An ATX power supply and a paperclip

Really, any power supply designed to power hard drives through a 4 pin molex connector will work. (It must have both 12v and 5v pins.) You can buy special power supplies for this purpose, or you can use an existing drive case with the cover off, but the easiest way I know is to haul out that old ATX power supply from the junk box, plug one of its drive power connectors into the drive, and hotwire it with the paper clip. I'll cover the hotwiring later. If your ATX power supply has a power switch on the back, so much the better. It's very handy to have, but it's not required. Any ATX power supply has far, far more current than we need to drive one old hard drive. Whether it came out of some micro-ATX set-top box or your old tower, it will be fine as long as it works.

■ **Note** Power supplies don't live forever. Inexpensive ones in particular will have cheap electrolytic filter capacitors to filter noise out of the 5v and 12v outputs. These capacitors dry out over time, particularly in high heat environments (like the inside of a hard-working power supply), or from disuse. The worst case is they'll short out, which will prevent the power supply from starting. Even if it does start, there may be so much noise on the 5v line from dried out filter capacitors that the drive can't communicate effectively. Replacing the caps exposes you (and your soldering) to high voltage, so I don't recommend it. Power supplies are cheap.

If you have an even more ancient XT or AT power supply, you can use that, too, and you don't even have to hotwire it. The power supply will be labeled somewhere what standard it's part of.

A PATA drive

PATA drives are easy to recognize. Desktop drives have a wide connector across the back with 40 pins, followed by a set of jumpers to select whether a given drive is drive 0 or drive 1 in a given PATA chain. Make sure your drive is set to 0 or Master. There may be other jumpers to allow such features as cable select and so forth. Next to the jumpers will be the power connector.

If the drive has a 50 (or more) pin interface and talks about addresses, it's a SCSI drive, and is not useful for this project. Likewise, if it has a pair of horizontal L shaped connectors, it's a SATA drive. While SATA is a close relative of PATA, the demands of SATA in terms of speed make communicating with a bit-banged Cestino interface impossible.

If the drive you're looking at is a 2.5 inch drive from a laptop with 44 pins, you are looking at a 44 pin PATA interface, which includes all four power pins plus the standard 40 pins of the PATA interface. If you have one of these, you'll need an adapter to connect it to the 40 pin PATA interface, but it will work. These adaptors are still common and still very cheap. You may even have one in your junk box.

Some drives won't work. If the drive is larger than 128GiB, it's going to want 48-bit logical block addressing, and this sketch can't do that. If it's so old that it doesn't have logical block addressing at all, likewise, it won't work. The best vintage of drive seems to be late 1990s to early 2000s.

The Bad Old Days

Just for a moment, let's flash back to the heady days when IDE/ATA was new and awesome. My first PC, a mighty 10mhz 8088 clone machine, cobbled together from leftovers in 1990, came with two 5.25 inch floppy drives and the first hard disk I ever owned: a 5.25 inch half-height 40 mebibyte RLL hard drive. (The "modern" 3.5 inch by 1 inch format was available by 1988, but the old 5.25 half-heights were cheaper.) Therein lay the problem. When PC hard disks were first introduced in 1980, with the Seagate ST 506, they used a standard called MFM—modified frequency modulation—essentially the same encoding scheme used for most floppy drives right up to the end. RLL drives were different. They could get more bits onto the same physical drive mechanism. This was a big win. Drives were expensive.

Encoding technology is complicated business. RLL stands for Run Length Limited, where a run is the number of bits in a row that can occur without a flux reversal on the drive. RLL is like a family name. Technically MFM is a Run Length Limited format, too, as is IDE/ATA. In those days, however, RLL referred to a specific encoding standard not otherwise named.

The bad news was that the RLL standard was like the Betamax standard. MFM was much more common. When I upgraded that machine to a 286, with a 16 bit bus, finding a 16 bit RLL card to support it was a real problem.

The underlying issue was this: In those days, most of the smarts to control a drive were on the drive control card, and there were dozens of standards: RLL, MFM, ESDI, and SCSI leap to mind, but there were many others. Each had its advantages and proponents. None of them would interoperate. All of them came with little driver patches stored in ROM that loaded into your PC at boot time, but if your operating system had its own drivers, they needed to know about your particular hard drive controller. Worse, drives were changing so fast that the second generation of a "standard" might not interoperate with the card that drove your first generation drive. You might not be able to operate two drives separated by a few years on the same controller. The way the data was written to the drive was evolving, drive speeds were evolving, and all these things necessitated different controller technology.

Arriving at the same time as ESDI and a year before SCSI was IDE, the standard we now know as PATA, properly called in its documentation as ATA.

It was awful.

IDE was a marketing term for Integrated Drive Electronics. This meant that the drive's controller was attached to the drive itself rather than an addon board in the computer. The IDE interface on the host side was very simple. It basically repeated the signals of the PC ISA bus to the drive.

We hated it. It was PC specific, for one thing. Also, whereas the sophisticated controller boards for MFM, RLL, ESDI, and SCSI would do all the actual computing required to access drive information, IDE offloaded this to the host computer. In those days, CPUs were so slow you could feel the difference.

Where ATA, now PATA and SATA shined and still shine is simplicity. You plugged a cable from the drive to the host adapter, and plugged the host adaptor into the bus of your PC. Manufacturers loved it because for the cost of adding a connector to the motherboard, more or less, the system could have a hard disk interface. We learned to love it because it drove the cost of hard drives down very, very quickly, to the point that I bought my first 1GiB drive (supposedly, see the rant below) by 1995, and you'd better believe it was PATA.

Today, of course, nearly all hard drives and their SSD counterparts use either PATA or SATA, and the controllers are built into the System-On-A-Chip or (for PCs) the SouthBridge part of the chipset. Most of the old standards are gone, SCSI is changed virtually beyond recognition, and even PATA has begun to slip beneath the waves. For personal computing, at least, it's SATA or SD cards (which communicate over SPI, among other things. The Cestino can do that.)

The advent of SSD (Solid State Drives, in case you wondered) raises the question as to whether even SATA will survive. It is now possible to simply attach an SSD directly to a computer's main bus, bypassing the extra host adapter hardware altogether. I very much expect that we will see flash or similar solid state storage as part of the memory map of a computer soon. The whole paradigm of booting from disk, saving to disk, and so forth may slowly fade away when storage is just a matter of transferring data from one area of RAM to another, where the flash is.

A short rant about units. Once upon a time, you bought a floppy disk that held 360 kilobytes. You reasonably expected it to hold, minus space for its format, 368,640 bytes. In the RAM and ROM/Eprom world, JEDEC standards required that RAM be measured in binary units, that a binary kilobyte was 2^{10} bytes or 1024 bytes, a binary mega was 2^{20} or 1048576 bytes, and a binary gigabyte was 2^{30} or 1,073,741,824 bytes, and so on. The hard drive world had no such standard, although generally they lived with the JEDEC standard. In the early 1990s, the race to build and market the first gigabyte drives began. Drive manufacturers began to lie. A gigabyte, they argued, was a billion bytes. This was pure marketing evil. A (base 10) gigabyte was, in reality, only about 954 of what we used to call megabytes, and they got dragged into court in a class action lawsuit, which was ultimately settled out of court. Their defense for their marketing evil was that the international standards upon which the metric system is based clearly define kilo as 10^3 mega as 10^6 and giga as 10^9, which although true is irrelevant.

The units we use today, and which I'll grudgingly use for this chapter, have "bi" in the middle to make sure there is no weasel room to define them as anything other than binary units. Kibibytes, mebibytes, gibibytes, and tibibytes are the units that hard drives are measured in, when they're measured properly (and honestly.) Ram and any other devices conforming to the JEDEC standards, however, are safe to talk about in kilobytes, megabytes, and gigabytes. The unit name changes because hard drive manufacturers were dishonest. End of rant.

ATA

When they were first introduced, the drives we now call PATA were IDE drives. This was a marketing term for Integrated Drive Electronics. These electronics provided a standard interface, particularly to the PC ISA bus, for a drive regardless of its underlying technology. We're going to dig deep into what those electronics actually do and how to use them.

Even in the early days, when floppies ruled, drive control electronics were relatively sophisticated. You selected some registers in the controller, wrote some bytes to them, and the controller handed you bytes, or stored your bytes. The controller handled the analog to digital and digital to analog conversion, data clocking, encoding and decoding of flux transitions, and so forth, and by the 1980s these were usually a single IC. (In the late 1970s, it was different. The reference designs for floppy controllers used dozens of logic ICs. When the Apple 2 needed a floppy drive interface, Steve Wosniac replaced all that logic with an EPROM, a pair of 555 timers, and some latches, and let the CPU handle everything else, which let Apple produce the drives and interfaces at an affordable price.)

If this sounds a lot like what we did in Chapter 8 writing to the flash IC, you've got a good ear. If you suspect that a hard drive might have similar notions to both floppies and the flash IC about control registers and so on, you're more right than you may know. Most complex ICs are controlled that way. A lot of retro-computer enthusiasts decry the use of modern microcontrollers like the ATmega1284P in retrocomputing projects, but the truth is that the classic ICs: 8250 UARTS for RS232 communications, 8255 programmable peripheral interface (used for PC parallel interfaces like printers), were more like dedicated microcontrollers than logic. The controller on a PATA drive is no exception.

PATA spanned about a decade of drive development, from the late 1980s before it was standardized through 2013, or so, when the last manufacturer switched over to serial ATA, or SATA. Drives evolved in size from hundreds of mebibytes to hundreds of gibibytes. As a result, the standard evolved over the years, adding faster communication modes, larger address space, and so on. We'll be using the original 1994 version of the standard: X3.221-1994.

Even if they've been withdrawn like X3.221-1994 was in 1999, standards are usually owned by someone. I had to get permission from ANSI to use X3.221-1994. While the folks at ANSI entirely reasonable and helpful about the whole process, it's something to keep in mind when putting your projects on the web, particularly if they include register names and commands: someone owns that, and it's probably not you. You should also read the disclaimer at the front of this book that was required by ANSI.

We'll be using the original PIO mode, now called PIO 0, for programmed input/output. It's the simplest, and slowest, way to access the drive. There are other, faster modes, particularly as you get into more modern ATA standards.

■ **Note** ATA1 aka X3.221-1994 has only 28 bit large block addressing, and thus can access only 128GiB of drive space. That said, the sketch presented here will work if your drive is bigger than that, but nothing above 128GiB will be visible, and you may not be able to write to the drive.

Bit Width

PATA drives transfer data in 16 bit words to and from a 16 bit data register not unlike the two ports we're using on the Cestino to send and receive those bits. We'll be using two ports, A and C, as usual, but they'll be the data bus rather than the address bus. It's very important to note that while PATA can transfer 16 bit words, the data register is the only 16 bit register we can get at. The rest of the accessible registers are all 8 bit, so for those we will read only the LSB (least significant byte) of the 16 bit data bus.

Endian-ness

Because we're dealing with 16 bit words, we have to deal with endian-ness, too. Which endian type is the drive? Well, it varies. When you read the drive information block, it is big-endian. Once the drive is storing your data, it no longer cares what the endianness is. It will store it however the host computer sends it. If your drive came out of a PC, it's little-endian, the standard used by Intel since the beginning. If it came out of other systems, particularly a PowerPC era Macintosh, it is probably big-endian, as the PowerPC could go either way and Apple came to PowerPC from Motorola CPUs, which were always big-endian. (68k Macs used SCSI drives, which this sketch won't work with.)

The sketch assumes the data on your drive, if any, will be little-endian. If you know the drive came out of a big-endian machine, you'll need to make a slight alteration in the sketch. I'll point it out when we get there, but you should know that I haven't tested the sketch with a big-endian data drive. Intel era Macs have the same endian-ness as PCs: little.

Anatomy of a PATA Drive

In order to do this project, we have to dig a little deeper into the PATA drive. You could argue that 90 percent of that is really the built-in controller, but from our perspective on the outside of the drive, the controller and drive are one unit, just like a flash IC or an EPROM, so we have to treat them that way, more or less.

There are a number of parts to the drive. I have them in rough order by level of abstraction from our point of view outside the drive: signals with outside connections, registers you can get at with signals, and things you can access once you can get at the registers.

Control Signals

ATA drives have quite a few control signals. We'll be using most of them, although we'll ignore the DMA transfer signals and a few others, and save ourselves some wiring. These signals allow TTL and similar electronic logic to control things like the flow of data to and from the sector buffer at speed, just as they did in the EPROM/FLASH explorer in Chapter 8.

DD0-DD15

These are data IO lines. The first eight are used when communicating with the drive controller, and all 16 are used for transferring data.

/DIOW, /DIOR

These are the Drive I/O Write and Read signals. As with the Flash Explorer project in Chapter 8, this means reading and writing *to the drive controller*, not necessarily to the drive. Essentially, these lines tell the drive controller how to set up its data direction settings, just as we will on the Cestino.

DA0-DA2

Drive Address Bus lines 0-2. The drive controller has many registers. We use these lines to select which one we want. In addition to these are the following.

/CS1FX, /CS3FX

These are chip select signals. They tell the drive controller whether we want to access the command block registers or the control block registers. Together with the DA0-DA2 address signals, these select the specific register to address. These signals are active *low.*

Because there are three address signals and two select signals, in theory, we could have 32 registers, and they can have different meaning depending on whether we read them or write them. I suspect part of why ATA has had such a lengthy history is that they left plenty of room in the registers to expand things. Thirty-two registers with two modes (read or write) each are a lot.

/Reset

Resets the drive.

/DASP

This signal indicates when a drive is active, or that drive 1 is present. Because we only have drive 0, we're using this in its “Drive active” mode. When it goes low, it means the drive activity light should be lit. Sounds like a job for the logic probe section of the Cestino.

Registers

A register, you will recall, is an area of memory in a microcontroller or microprocessor in which a single data word (often a single byte) can be placed. Usually each bit has a specific meaning. It's the same for the registers in the PATA controller. Only the data register holds more than 8 bits (it holds 16). Here's the list of registers and what they do.

Alternate Status Register

Address: 0b110. Chip select: /CS3FX.

This register contains the status bits of the drive. It's a copy of the main status register, but getting at it doesn't muck around with the drive completing other actions. We use it for every status query. In order, from highest to lowest bit, just as you'd expect in 0b notation the bits are:

- BSY: Drive is busy. This means the drive can't accept any new commands from the command block registers. Literally, it means the drive has access to the command block registers, so the host should not.
- DRDY: Drive is Ready. The drive can respond to a new command. The host can go ahead and write to the command block registers.
- DWF: Drive Write fault. Something bad happened during the write.
- DSC: Drive Seek Complete. The host told the drive to go to a particular track, and the heads are there. Hard drives are physical mechanisms. This takes a certain amount of time.
- DRQ: Data Request. The drive is ready to transfer data between the host and the drive, either direction. Data can be in bytes or 16 bit words.
- CORR: Corrected Data. The drive had a data error but fixed the data. We don't actually use this bit, since we're not doing critical data transfers.
- IDX: Index. Set on once every revolution of the drive. Basically means: drive is spinning.
- ERR: Error. Something bad happened. Read the error register to find out what.

The Alternate Status register can also be written to, in which case is it called the Device Control register. When it is, raising bit 2 will reset the drive, and raising bit 1 enables interrupts for the drive. We'll be flipping bit 2 once in a while.

Data Register

Address:0b000. Chip Select: /CS1FX.

This is a 16 bit register (all others are 8 bit). When we want to read or write data to the drive, this is the register we use.

Drive Address Register

Address: 0b111. Chip Select: /CS3FX.

This register contains drive select and head register information, inverted. It's a read-only register, and we don't use it.

Sector Count Register

Address: 0b010. Chip Select: /CS1FX.

This register tells how many sectors the drive expects to transfer on the next read or write operation. This is really for DMA and large scale transfers. We're going one sector (LBA block) at a time.

Sector Number Register

Address 0b011. Chip Select: /CS1FX

When doing a transfer, start with the sector in this register. In LBA mode, which we're going to use, this register contains the lowest 8 bits of the 28 bit LBA block address. At the end of a command, you can read the lowest 8 bits of the current LBA block address from this register. This one's important. We use it.

Cylinder Low and High Registers

Addresses: 0b100, 0b101. Chip Select: /CS1FX

In Track/Sector mode, these registers would hold the low and high bits of the cylinder of the drive to use. In LBA mode, they hold bits 8-15 and 16-23 of the 28 bit block address. We use these too.

Drive/Head Register

Address: 0b110. Chip Select: /CS1FX.

In Track/Sector mode, the first four bits of this register select which of up to 16 heads to use. It also contains one bit (bit 4) to select which drive to use, and one bit (bit 6) to select whether we're in LBA mode or not. If we are in LBA mode, bits 0 to 3 contain the highest four bits (24-27) of the block address.

Status/Command Register

Address: 0b111. Chip Select: /CS1FX

Contains the same information as the Alternate Status Register, but has a lot of other side effects and interactions. We use the Alternate Status Register instead. When you write to it, this register is the Command Register, and we send many commands to the drive through this register.

The Sector Buffer

The Sector Buffer is a batch of memory in the drive that contains one sector of data: 512 bytes. This is the minimum size block the PATA drive will transfer data in. It's also the size of an LBA block, and conveniently, it's the size of MS-DOS (and thus Windows) filesystem data blocks. The Sector Buffer is used mostly for transferring data onto and off of the physical disk(s), but the Identify Drive command loads the sector buffer with pre-programmed drive information without reading the physical disk at all. It's the sector buffer that is read and written through the 16 bit data register.

Commands

As with registers, there are lots of commands. In the sketch, because there are so few parameters to set on other registers directly, I've included a few parameters in the CMD command list, and I'll include them here as well. Unlike the registers, I'm only going to list the commands we use.

Sleep

Value: 0x99. Register: Status/Command.

This command tells the drive to spin down and go to sleep. In sleep mode, most other commands will fail, and because of how the sketch's ready detection works, most will hang the sketch until the drive is reset.

Identify Drive

Value: 0xEC. Register: Status/Command.

What it does: Tells the drive controller to copy its identity information—the drive serial number, number of tracks, sectors, cylinders, and heads, number of LBA blocks and potentially a lot more from wherever it's stored in the controller to the sector buffer, where it can be read like a normal sector.

Sequence of Events: The drive sets the BSY signal, copies the identity information into the sector buffer, sets DRQ, and generates an interrupt (which we ignore.). When we next read the status buffer (which normally the host would do when the interrupt gets set) the drive clears the interrupt and DRQ.

Read Sectors (With Retry)

Value: 0x20. Register: Status/Command.

What it does: This command copies the data from the physical disk(s) at the address set in the address registers into the sector buffer. It can copy sequences of blocks, but we're not using that functionality. Retrying means that if it encounters an error it can try to read the block again and correct the data in the sector buffer.

Sequence of Events: The host (the Cestino) sets the address parameters and LBA mode bit, if any, then writes the command to the command register. The drive sets DRQ and BSY, then seeks the requested track and sector (or LBA address).

If we're transferring more than one block, when the read completes, the address of the last sector will be in the address registers. If the read doesn't complete without an error, the address registers will point to the sector where the error occurred.

When the read is complete, the drive sets INTRQ, the interrupt request. The host reads the status register on interrupt, which causes the drive to clear INTRQ. The drive then clears DRQ, and if there's another sector to be transferred, sets BSY again.

We skip a lot of the host steps in the sketch, and it still works anyway.

Write Sectors (With Retry)

Value: 0x30. Register: Status/Command

What it does: Transfer the contents of the sector buffer onto the physical disk(s). Again, retries are allowed to correct the data.

Sequence of Events: The host (Cestino) sets the address registers for the sector (or block) we're writing to, along with setting the LBA flag (in our case) on the Drive/Head register. The host then writes the command to the Status/Command register and things begin to happen.

The drive seeks the track/sector/head or LBA address and sets DRQ when it's ready to receive the first (and only, in our case) sector/block of data. The host writes the sector/block to the sector buffer. The drive clears DRQ and sets BSY instead.

When the drive is done, it sets INTRQ, the host reads the status register, the drive clears the register and, if no more sectors/blocks are expected, clears BSY.

Once again, the sketch skips a bunch of these steps, particularly the interrupt steps. We get away with this because we're only writing one sector (block) at a time.

LBA Mode

Value: 0x40. Register: Drive/Head (LBA3)

This is not a real command. It's an addon to the Drive/Head register address to ensure Large Block Access mode is set. This simplifies the whole cylinder/track/sector/head business to a flat 28 bit address that points to a given 512 byte block, exactly the same in all other respects as a sector. It's ANDed with the four address bits going into that address.

The Physical Disk(s)

This is my catch-all for the storage media. Most data either winds up here or comes from here. I'd love to be able to explain in the usual detail how bits go to the disk and become magnetic transitions on the rotating media, but the truth is that the huge variety of PATA drives spans huge swaths of different recording technology. Perpendicular Magnetic Recording? Flash? M-RAM? Bubble Memory Cartridge? Something else entirely? It doesn't matter. The entire point of the ATA interface is that *we don't care how it works*. The controller presents whatever the storage technology is as blocks of 512 bytes, and

uses one of the standard methods of addressing, and what happens under the hood isn't our problem. There may be special considerations, like in the old days when you had to park disks before turning them off or the heads would touch down wherever they happened to be, but finding drives that old that still work isn't very likely at this point.

Build the ATA Explorer

Okay. Now that we know in principle what's in a PATA drive and how to talk to it, let's build the project. We'll start with the ATX power supply. If you're not using one, you can skip ahead.

ATX PS Pinout
(Connector View)

Left	Right
Org +3.3v 1	11/13 +3.3v Org
Org +3.3v 2	12/14 -12v Blu
Blk Com 3	13/15 Com Blk
Red +5v 4	14/16 PsOn Grn
Blk Com 5	15/17 Com Blk
Red +5v 6	16/18 Com Blk
Blk Com 7	17/19 Com Blk
Gry Pwr Ok 8	18/20 -5v Wht
Pur +5vSB 9	19/21 +5v Red
Yel +12v 10	20/22 +5v Red
Yel +12v 11	23 +5v Red
Org +3.3v 12	24 Com Blk

Figure 9-2. *ATX Power Supply Connector Pinout (Connector View)*

Hotwire the ATX Power Supply

Got an ATX power supply? Does it have a big, two row motherboard connector like the one in Figure 9-2? It might be a 20 or 24 pin connector. I've covered both in Figure 9-2. The alternate pin numbers are shown on the right, and I'll refer to them the same way in the text. Unplug the power supply from the wall, just on general principles.

The wires of the motherboard connector are color-coded. Pin 14/16, as you can see from Figure 9-2, is Power On, and its active low. It's the only green wire in the bunch. Conveniently, it's surrounded by Com (ground) pins, so take your paperclip and stick it in pin 14/16's socket, and from there to either pin 13/15 or pin 15/17.

Most ATX power supplies made in the last decade and a half will start without a load, so go ahead and put the connector down and plug the power supply in. If there's a power switch on the supply, make sure it's on. The fans on the supply should start. If they don't, disconnect the power and check your connections. The supply should do nothing if you get it wrong. The only wire that's live besides the 3.3v PS On sense wire is +5vSB, the purple wire, on the other side of the connector. Obviously if the paperclip glows red hot, turn everything off right away.

Power supply starts up? Great. Grab your multimeter and measure from pin 10 to ground and from pin 4 to ground. You should get twelve and five volts DC, respectively.

Power supply didn't start? Some really old ones need some kind of load to start, so if nothing else seems amiss, go ahead to the next step anyway.

Shut everything off and/or unplug it from the wall, and plug one of the drive power connectors into your hard drive. Check your polarity. Worn molex drive connectors can go in upside down, and it's not good for the drive. If it starts now, you're in business. The drive may or may not spin up, depending on its settings. If it does, it's a very good sign. You might want to tape your paperclip in place with electrical tape.

Power supply still didn't start? Check your paperclip. Make sure it's plugged in to the right pins and big enough to make contact with the socket.

There are a lot of tutorials on the net to open up the power supply and turn it into a bench power supply. I have one like that. It works very well, but follow those tutorials at your own risk.

Likewise, it's certainly possible to power your Cestino from an ATX power supply, but if you get a 12 volt rail instead of a 5 volt, you will probably fry all the electronics connected to your breadboard, all at once. Been there, done that.

■ **Safety Note** ATX and similar PC power supplies are generally safe for humans on the outside, but inside the metal case are voltages that could kill or severely injure you. These voltages may still be present even with the power disconnected from the wall. If the power supply has been wet, smoked/been on fire, or if it's shocked someone already, it's dangerous. Don't use it. Also, be aware at all times that a running power supply has plenty of energy to heat things red hot and start fires.

Set Up the PATA Cable and Pins

The prototype for this project amounted to lots of pin-to-socket wires plugged into the 40 pin connector of a drive. It worked. You can do it that way if you really want, but I'll tell you it was a misery to wire and wasn't very reliable. That's why I came up with this new method.

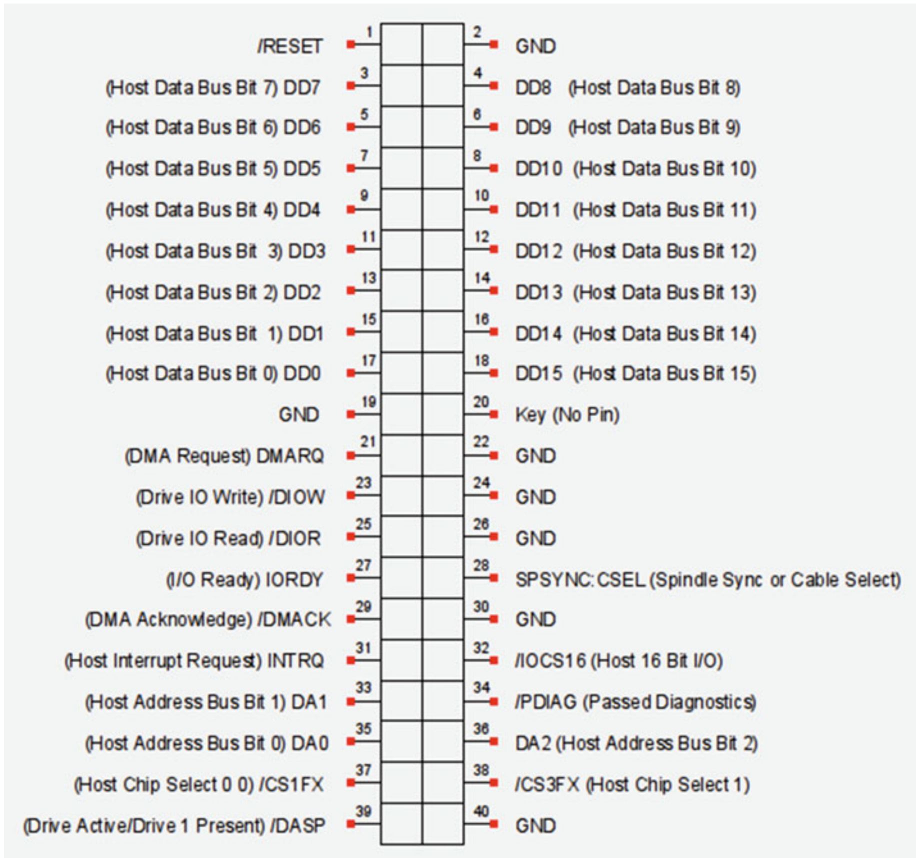

Figure 9-3. *PATA Connector Pinout, Drive Side*

Your PATA cable should have three connectors that look like Figure 9-3 above. One will be further from the other two. It was meant to connect to your system's host adapter, either on a card or on the motherboard itself. The two connectors close together were meant to connect to drives. They should all be the same. My cable had "master" and "slave" printed on the ribbon cable next to the connectors, so I went through all the pins and checked them with my multimeter to make sure they're all straight through. They are.

The pinout diagram in Figure 9-3, like most of the diagrams you'll find online, is how the pins appear from the back, when plugged into your breadboard. They're also how the pins are in the drive's connector. When you're looking at cable connector's socket side, they're backward.

On the end of the cable with two connectors closer together, take the outer connector and plug 20 pins of your extra long pin header supply into the row of sockets closest to the plastic bump or furthest from the blocked socket that may be present on the opposite side. My cable has both. This connector will serve the odd numbered pins of the interface.

If your cable has the blocked socket 20, you'll need to cut a piece of pin header with 9 pins and one with 10. Take the middle drive connector and plug these pin headers in to the side with the blocked socket. They'll only go in the correct way, with 9 pins on one side of the blocked socket and 10 pins on the other.

If your connectors don't have a blocked socket (they must, in that case, have the plastic bump) you can go ahead and take the middle connector and put another 20 pins worth of pin header into the row of sockets farthest from the plastic bump. We won't use pin 20 in any case. Do whichever is easiest. This will be the even pin connector.

Your ATA cable should look like Figure 9-4.

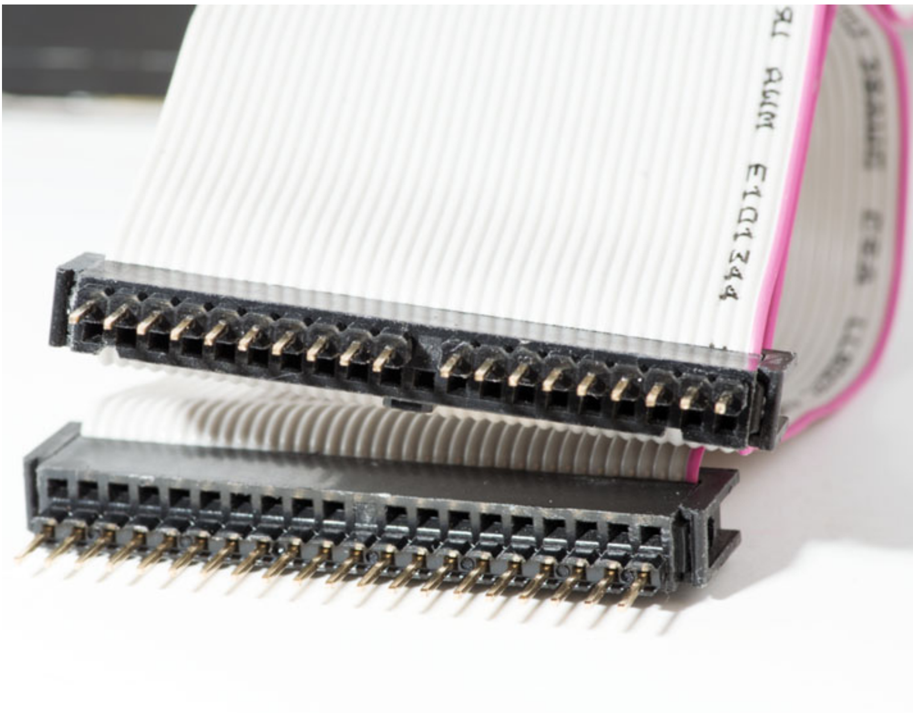

Figure 9-4. *ATA Cable Set Up for Breadboard Connection*

With your Cestino breadboard reset-button-end down (assuming you built it like the one in the pictures), plug the end connector (odd pins) into your second breadboard on the side of the IC trench closest to the Cestino, and the middle connector (even pins) into the other side of the trench. This should put the motherboard connector extending off to the right of the Cestino.

There's nothing special about this orientation, other than it puts the connections of the PATA cable in the same order they appear in Figure 9-3, which makes it easier to keep the wiring straight.

Wiring Up

Before we get into the port wiring, we need to wire up all the grounds. As you can see in the schematic in Figure 9-5, there are a lot of them, and they all need to be connected. On the PATA interface, pin 19, on the odd connector, and pins 2, 22, 24, 26, 30, and 40 on the even connector should all be wired to the - bus.

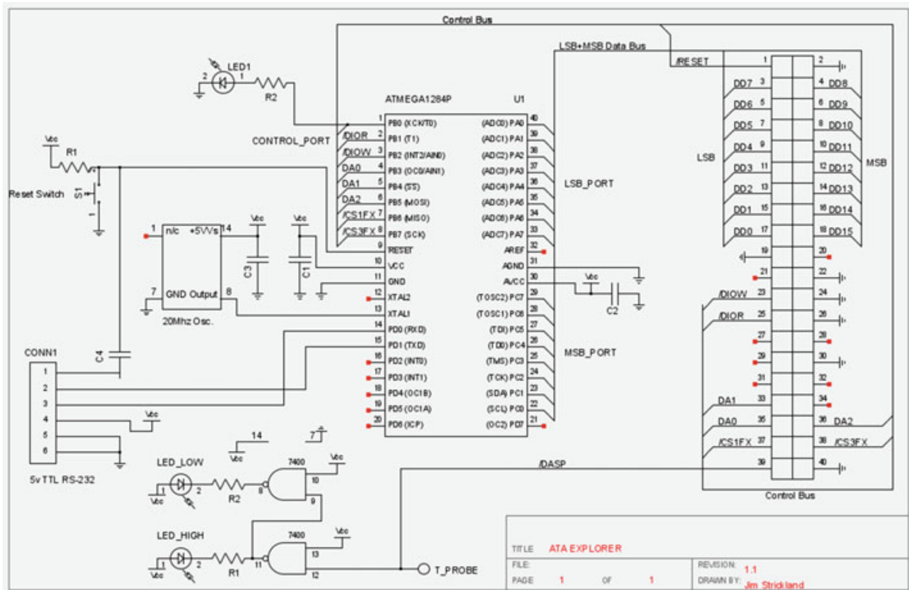

Figure 9-5. *ATA Explorer Schematic*

This is another three port setup, although for wiring convenience it's different from the one we used in Chapter 8. We'll use PORT A for the LSB port—the least significant byte of our data bus, in order, so PA0 connects to DD0, and PA7 connects to DD7. We'll use PORT C for the MSB of that bus, and PC0 will connect to DD8, PC1 will connect to DD9 and PC7 will connect to DD15. We'll use PORT B for the control bus.

Remember that PORT A is backward, running from pin 40 of the ATmega1284P for PA0 to pin 33 for PA7. Wire these pins to DD0 to DD7, on the odd numbered side of the connector, where DD0 is pin 17 and DD7 is pin 3. These wires will all cross over each other. Make sure you leave some length for that. Once you have the port wired, go ahead and hank the wires up to keep them out of trouble.

We'll wire port C to the MSB data pins next. Port C goes from pin 22 of the ATmega1284P for PC0 to pin 29 for PC7, and pins DD8 to DD15 go down the even connector, starting at pin 4 and going to pin 18. I led these wires around the bottom of the PATA connection area, mostly for a pretty photograph. Once you have these wires in place, go ahead and hank them up, too.

■ **Note** Hanking the wires can pull them partway loose from the breadboard socket. Check your connections after you finish hanking a group of wires. A loose wire or two can make data transfers fail in bizarre ways.

Finally, wire PB0 (ATmega1284P pin 1) to /RESET (pin 1) of the PATA interface, PB1 to /DIOR (pin 25), PB2 to /DIOW (pin 23), PB3-PB5 to DA0-DA2 (pins 35,33, and 36, respectively), PB6 to /CS1FX (pin 37) and PB7 to /CS3FX (pin 38).

The Sketch

This is a big sketch, so I'm going to break it up into sections more obviously than I have in previous sketches.

Preprocessor Definitions

I've been using preprocessor definitions (#defines) for a variety of small tasks. Usually there have been a small handful of them. They take center stage in this project, because there are a lot of bit patterns we need to set the control signals of the drive into in order to get what we want.

The first group of #defines lays out what ports do what. If you want to change which ports the sketch uses, changing the values here will do that for you (but you have to sort the wiring out for yourself.)

First, we define the control port, which covers the /RESET signal, /DIOR and /DIOW read and write signals, the DA0-DA2 address signals, and the /CS1FX and /CS3FX signals.

```
#define CONTROL_PORT PORTB
#define CONTROL_DDR DDRB
#define CONTROL_PINS PINB
```

The next group of defines covers the least significant byte of the data bus.

```
#define LSB_DDR DDRA
#define LSB_PORT PORTA
#define LSB_PINS PINA
```

As you might expect, the most significant byte of the data bus comes next.

```
#define MSB_DDR DDRC
#define MSB_PORT PORTC
#define MSB_PINS PINC
```

While we're defining our Cestino ports, let's also define read and write mode, for convenience. These values are passed to the DDR register on the Cestino for the particular port we're dealing with. These are the usual 1 for write and 0 for read codes.

```
#define DATA_READ 0b00000000
#define DATA_WRITE 0b11111111
```

Remember all those registers? We select them with a combination of bit patterns set to the control port. For reference, the port is laid out, in 0bxxx order, like this:

/CS3FX,/CS1FX,DA2,DA1,DA0,/DIOW,/DIOR, and /RESET.

```
#define REG_DEFAULT     0b11000111
```

This is a register where sending signals does nothing. We use it for safety.

```
#define REG_STAT_COM     0b10111111
```

The Status/Command Register, where we send nearly all of the CMD bit patterns, except the one that isn't really a command.

```
#define REG_ASTAT_DEVCOM 0b01110111
```

This is the register we read the status from. It's the Alt Status/Device Command register.

```
#define REG_ERR_FEAT     0b10001111
```

Error/features register. If we need to check for errors, this is where we go.

```
#define REG_LBA0         0b10011111
#define REG_LBA1         0b10100111
#define REG_LBA2         0b10101111
#define REG_LBA3         0b10110111
```

These 3.5 bytes are the registers where we put all 28 bits of a Large Block Access address. LBA is a "flat address space" for 512 byte blocks of the drive, rather than messing around with track, sector, and head, which changes from drive to drive.

```
#define REG_DATA          0b10000111
```

This is the data register, the only 16 bit register in the system. All large data transfers go to and from this register, although some may be only 8 bits wide.

```
#define REG_HEAD REG_LBA3 //Same register, different modes.
```

The head register has a number of functions besides the half of it used for LBA3, so we define it again for clarity.

Does this list start to look like the registers list? It should. Likewise, the command #defines will look exactly like the commands list. All of these commands except the last are sent to the Stat/Com register.

```
#define CMD_SLEEP 0x99
```

Put the drive to sleep/spin down.

```
#define CMD_INIT 0x91
```

Initialize the drive so we have a known state.

```
#define CMD_IDENTIFY 0xEC
```

Copy the drive's id info to the drive sector buffer so we can read it with the data register.

```
#define CMD_READ_WITH_RETRY 0x20
```

Copy a block from the physical disk(s) to the sector buffer, so we can read it with the data register.

```
#define CMD_WRITE_BUFFER 0xE8
```

Write data to the sector buffer. We don't actually use this command.

```
#define CMD_WRITE_WITH_RETRY 0x30
```

Write the sector buffer to the physical disk(s). The catch is, as we write our data to the buffer, it's copied to the disk(s) right now.

```
#define CMD_LBA_MODE 0b01000000
```

Set the drive to LBA mode. This is the one command that isn't really a command. It's just defined here because it acts like one. It's applied to the head register to make sure the drive is in LBA mode.

When we read signals from the status register, they come out in this order: BSY DRDY DWF DSC DRQ CORR IDX ERR. If we AND the following masks with the status register, we can pop out the one bit we're interested in with a single step. We don't use most of these statuses, but they're here in case you need them down the road.

```
#define BSY 0x80 //0b10000000
```

Busy

```
#define RDY 0x40 //0b01000000
```

Ready

```
#define DF  0x20 //0b00100000
```

Drive Write Fault

```
#define DSC 0x10 //0b00010000
```

Drive Seek Complete

```
#define DRQ 0x08 //0b00001000
```

Data Request

```
#define CORR 0x04 //0b00000100
```

Corrected Data

```
#define IDX 0x02 //0b00000010
```

Index

```
#define ERR 0x01 //0b00000001
```

Error

There are also #defines for the read/write line statuses. It's a big script, so there are a lot of things #defined that could be left as bit patterns in smaller scripts. The signals, in 0bxxx order are: /DIOW, /DIOR, and /RESET.

```
#define HW_OFF  0b111
```

Set the read-write lines all off.

```
#define HW_READ 0b101
```

Set the read-write lines for reading.

```
#define HW_WRITE 0b011
```

Set the read-write lines for writing.

```
#define HW_RESET 0b110
```

Lower the reset line to reset.

Because we have to switch from big endian to little endian reading the sector buffer, I went ahead and defined BIG_ENDIAN and LITTLE_ENDIAN for transfer_sector_buffer(), which transfers data to and from the drive's sector/LBA buffer. We'll talk about that function shortly.

```
#define BIG_ENDIAN true
#define LITTLE_ENDIAN false
```

Debug mode is a special case. This sketch took a lot of work to get right, and there's a lot of debug code in it, mostly Serial.println() commands. I could have gone through and taken them out, as I have in the shorter sketches, but instead, when I put them in, I enclosed them in #ifdef and #endif precompiler macros.

What these do is *if* DEBUG is defined, those statements get compiled in. If it's not defined, they don't. Here's the definition of DEBUG, which you can see is commented out.

```
//#define DEBUG
```

If you define DEBUG, every function in the sketch that has debug code will have something to say when it's called. It can get overwhelming. But there's a trick. You can #define DEBUG inside a particular function. It will stay defined for the rest of the sketch, /or/ until it hits another precompiler macro called #undef. As the name suggests, #undef undoes a #define. So if you #define DEBUG at the beginning of a function, and you #undef DEBUG at the end, you can compile in the debug statements for that function alone. The precompiler is a powerful thing. We've barely scratched the surface of what it can do.

Global Variables

Global variables are really considered poor form in most code, but for sketches they are sometimes necessary and frequently useful. We absolutely need, for example, a place to put the contents of the drive's sector buffer in the Cestino's memory, so we define that variable globally as a 512 byte array called block_buffer. This can be a little confusing.

Remember that this array is /only/ on the Cestino. If we want it up to date with the sector buffer on the drive, we have to call a function to do that.

```
byte block_buffer[512];
```

These next variables hold information about the drive's cylinders, heads, and sectors. We don't use this data much, since we're in LBA mode, but we do use it. We use the LBA_sectors uint32_t a lot more. It's the number of LBA blocks on the drive.

I've used Block and Sector fairly interchangeably in this sketch. Because we're dealing exclusively with LBA blocks, which are exactly the same size and functionality as sectors, the difference is not important to us, except in the drive_sectors variable, which holds the number of actual sectors. We only use that data once, and then only for information messages.

```
int drive_cylinders = 0;
int drive_heads = 0;
int drive_sectors = 0;
uint32_t LBA_sectors = 0;
```

The non_zero_read variable is used as an easy mechanism for other functions and code to tell whether the last read of the drive sector buffer into block_buffer[] had any data in it. It's a clumsy mechanism, but it works.

```
boolean non_zero_read = false;
```

Low Level Functions

These are the lowest level functions. These directly touch the Cestino's ports, and by extension the control signals of the drive. They use nearly all the #defines.

get_drive_status_byte()

This function seems like a nicety, but it's absolutely critical for the rest of the sketch. All of the functions wait on something, usually for the drive to complete an action. That information is gathered by calling this function over and over again in a loop, sometimes forever.

This function takes no parameters. When called, it preserves the state of the CONTROL_PORT, defines a variable status_byte, and sets it to zero, selects the ASTAT/DEVCOM register, sets the LSB port up to read /and/ turns on its pull-up resistors, sets the CONTROL_PORT lines to read mode, reads the port and stores the value in status_byte, then resets CONTROL_PORT to its previous values. It then returns status_byte. If this seems like a belt and suspenders and duct tape approach, it will become obvious that I had a lot of unexpected interactions between functions early in the development of this sketch.

Note the #ifdefs and the debug code at the beginning and end of this function.

```
byte get_drive_status_byte() {


#ifdef DEBUG
  Serial.print("get_drive_status_byte() called.");
#endif

  byte temp_control_port = CONTROL_PORT;
  byte temp_lsb = LSB_PORT;
  byte temp_lsb_ddr = LSB_DDR;

  byte status_byte = 0;

  CONTROL_PORT = REG_ASTAT_DEVCOM;

  LSB_DDR = DATA_READ;
  LSB_PORT = 0b11111111;
  set_drive_hw_lines(HW_READ);
  status_byte = LSB_PINS;
  set_drive_hw_lines(HW_OFF);

  CONTROL_PORT = temp_control_port;
  LSB_PORT = temp_lsb;
  LSB_DDR = temp_lsb_ddr;

#ifdef DEBUG
  Serial.print(" Returning:");
  Serial.print(status_byte, BIN);
  Serial.print("\n");
#endif
  return (status_byte);
};
```

write_command_to_register()

Most functions other than get_drive_status_byte() read and write commands to registers the same way, so instead of having the same code over and over again, they call this function.

This function returns no data, and takes the parameters reg, the register to be written to, and command, the command to be written to it. As we did with get_drive_status_byte, we preserve the existing DDR state of LSB (this function can be called with LSB in either read or write state), then set LSB to write mode.

We set CONTROL_PORT to the register (which will be one of our REG_whatever #defines), which will set the chip select and address signals, then set LSB_PORT to the command, which will be one of the CMD_whatever #defines.

Once the signals are set, we call another function called set_drive_hw_lines, which sets the /DIOR and /DIOW lines to HW_WRITE. Then we call it again to set the /DIOR and /DIOW lines back to HW_READ. This is called strobing the read and write lines, and it's what tells the drive to read the data port and write that data to the register we've selected.

After that, we set CONTROL_PORT, LSB_PORT and LSB_DDR back the way we found them.

```
void write_command_to_register(byte reg, byte command) {
  byte temp = LSB_DDR; //Preserve LSB r/w state
  LSB_DDR = DATA_WRITE; //Set LSB's DDR
  CONTROL_PORT = reg; //Set CONTROL_PORT to register address.
  LSB_PORT = command; //Set LSB_PORT to the command byte.
  set_drive_hw_lines(HW_WRITE); //Strobe write/read lines.
  set_drive_hw_lines(HW_OFF);
  LSB_DDR = temp; //Restore the state of LSB's DDR.
}
```

set_drive_hw_lines()

The last of our three low-level functions is set_drive_hw_lines(). This function sets the / DIOR, /DIOW, and /RESET signals. Notice that all three of these signals are active low, so the logic will look backward.

Once we're past the debug code, we copy the value of the CONTROL_PORT register into temp. We then AND temp with 0b11111000, which preserves whatever is in the highest 5 bits, and sets the lowest three bits to zero.

Next, we OR temp with status, which will contain one of our HW_whatever defines, so the correct bits are set on and off.

Then we apply the result to CONTROL_PORT.

It's very important to do the bit twiddling of the value to be set to CONTROL_PORT in a separate variable and not on the CONTROL_PORT register itself, as the intermediate states of CONTROL_PORT can make the drive do very peculiar things.

```
void set_drive_hw_lines(byte status) {
#ifdef DEBUG
  Serial.print("set_drive_hw_lines() called with status:");
  Serial.print(status, DEC);
  Serial.print(" CONTROL_PORT was: ");
  Serial.print(CONTROL_PORT, BIN);
#endif

  byte temp = CONTROL_PORT;
```

```
  temp = temp & 0b11111000;
  //wipe the hw line bits

  temp = temp | status;
  //OR the remaining bits with the new setting (status)

  CONTROL_PORT = temp; //Set CONTROL_PORT to the new value.

#ifdef DEBUG
  Serial.print(" CONTROL_PORT now ");
  Serial.print(CONTROL_PORT, BIN);
  Serial.print("\n");
#endif
}
```

Testing

Once you have all the defines and all the globals in place, it's a good idea to make a basic setup() function and a loop() function to test these routines. If they don't work right, nothing that depends on them will. Your setup() function will need the Serial.begin(115200) line, and it will need to set CTRL_DDR to DATA_WRITE.

After that, #define DEBUG and call the low level functions to see if the debug values look reasonable.

Once your low level functions look good, it's time to move on to the rest of the sketch, but seriously. Stop here and test your low level functions. Debugging high level functions without full confidence in your low level functions is unpleasant.

Utility Functions

The next group of functions I call Utility Functions. They're not low level, in that they don't directly talk to Cestino ports to manipulate the drive's signals, but at the same time, they do not themselves implement complete parts of this sketch. They vary wildly in length, and quite a few depend on each other as well as on the low level functions.

wait_for_drive_drq()

This function forces the sketch to wait for the drive data request signal to be set in the drive status register. It returns no data and takes no parameters. If the DRQ signal is never set, the sketch can hang here forever.

We begin with the usual debug code, then get the drive status byte and store it to status_byte. We then store the OR of the DRQ and BSY status masks together in the mask variable. Then we begin the loop.

The loop checks to see if status_byte AND mask is logically true, that is, any of its bits are true, and if the result equals the DRQ mask. If both BSY and DRQ are set, this will be the case. Otherwise the loop reads the drive status byte into status_byte again, and repeats. Forever if need be.

```
void wait_for_drive_drq() {
#ifdef DEBUG
  Serial.print("wait_for_drive_drq() called: ");
#endif
  byte status_byte = get_drive_status_byte();
  byte mask = DRQ | BSY;
  while (!((status_byte & mask) == DRQ)) {
    status_byte = get_drive_status_byte();
  }

#ifdef DEBUG
  Serial.println("DRQ");
#endif
}
```

wait_for_drive_not_busy()

This function works much like the wait_for_drive_drq() function except that it waits for the drive to be /not/ busy. We read the status byte into status_byte, and set the mask for BSY into mask, then loop while status_byte AND mask do not equal zero, reading the status byte into status_byte over and over again until it changes.

```
void wait_for_drive_not_busy() {

#ifdef DEBUG
  Serial.print("wait_for_drive_not_busy() called: ");
#endif
  byte status_byte = get_drive_status_byte();
  byte mask = BSY;
  while (!((status_byte & mask) == 0)) {
    status_byte = get_drive_status_byte();
  }
#ifdef DEBUG
  Serial.println("Busy Clear");
#endif
}
```

wait_for_drive_ready()

wait_for_drive_ready is slightly different from the other two, in that it has a crude timeout mechanism. At the beginning of the setup() function, we check to see if a drive exists on the interface. Unfortunately, if there isn't one, wait_for_drive_ready is never set (obviously), so we have to wait for this function to time out. This function also inserts a 1ms delay in each loop cycle, so 1,000 cycles in ms_to_wait will make the function wait a full second, and the minimum time it will wait is one second. This function returns no data, but does take the number of ms to wait in an integer as a parameter.

we begin with the usual debug code, and as should be familiar by now, set store the drive status byte in status_byte and OR the masks for BSY and RDY together and store them in mask. We then initialize our ms counter to zero.

The loop tests to see if our mask AND the status byte are not equal to RDY, and also whether the counter c is less than or equal to ms_to_wait. If both these conditions are true, it reads the drive status byte again, delays 1 ms, increments c, and loops. It will continue to loop until either the RDY bit is set, or the timeout is exceeded.

```
void wait_for_drive_ready(int ms_to_wait) {


#ifdef DEBUG
  Serial.print("wait_for_drive_ready() called: ");
#endif
  byte status_byte = get_drive_status_byte();
  byte mask = BSY | RDY;
  //look at the highest two bits of the status byte.

  int c = 0;
  while ((!((status_byte & mask) == RDY)) && (c <= ms_to_wait)) {
    status_byte = get_drive_status_byte();
    delay(1);
    c++;
  }
  //While the highest two bits of the status byte are not
  //equal to 0x40 (RDY), delay 1ms, increment c,
  //and do it again.

#ifdef DEBUG
  Serial.println("DRIVE READY");
#endif
}
```

string2uint32_t()

In loop, at one point we read a 28 bit LBA address from user input. Unfortunately, the user input comes in as a stream of characters from the Serial.readString() function. While in normal C there are nice ways to convert character arrays to integers, there are no nice ways to convert character arrays to 28 bit uints, so we have to implement our own.

This function takes a String object as its input and returns a uint32_t as its output.

We begin by initializing a uint32_t called temp to 0. We then call the trim() method of the String object input, to ensure that we don't have trailing characters that will mess up our counting.

The loop counts from 0 to the result of the length() method of the input String object. Since each iteration represents another digit to the right of the last digit we read, we multiply temp by 10. Then we get the next character in the String object using the charAt() method and c as the position. This will give us the value of that character, but there's a catch. It will give us the numeric representation of the character /itself/. Because the Cestino (like all Arduinos) uses old school ASCII, that means the number 1 has a numeric value of 49. We do know, from looking at an ASCII table, that the numbers are in the table from 0 to 9, in order, lowest to highest. If we know that 0 is 48, we could subtract 48 from whatever numeric value we read in, but it's easier to subtract the numeric value of the character 0, which accomplishes the same thing. 49-48 is 1, and that's the correct value for the number 1. We add that derived value to temp, and loop. When we run out of characters in the input String object, we return temp.

```
uint32_t string2uint32_t(String input) {
  uint32_t temp = 0;
  input.trim();
  for (int c = 0; c < input.length(); c++) {
    temp = temp * 10 + input.charAt(c) - '0';
  }

#ifdef DEBUG
  Serial.print("string2uint32_t called with a string of >" +
               input + "< Returned:");
  Serial.println(temp, DEC);
#endif

  return temp;
}
```

transfer_sector_buffer()

This function transfers the drive's sector buffer to the Cestino's block_buffer array, /or/ transfers the block_buffer array to the drive's sector buffer for writing.

This function returns no data, but it takes a boolean to determine whether we're writing. (If we're not writing, the boolean is false, so we're reading.) It also takes another boolean to determine whether we're big_endian (using the BIG_ENDIAN and LITTLE_ENDIAN #defines for clarity). If big_endian is true, we read or write big endian words. otherwise we read or write little endian words.

We begin with debug code. Then we wait until the drive's BSY signal is cleared, and we set the non_zero_read global variable to false. Then we select the REG_DATA register. (Remember, the data register is 16 bits wide.)

```
void transfer_sector_buffer(boolean write, boolean big_endian) {
#ifdef DEBUG
  Serial.println("transfer_sector_buffer called.");
#endif

  wait_for_drive_not_busy();
  non_zero_read = false; //set false before we start.
  CONTROL_PORT = REG_DATA; //select the REG_DATA register.
```

If write is true, we set the DDR for both LSB_PORT and MSB_PORT to DATA_WRITE, otherwise we set them to DATA_READ. (See how all those #defines get used here?)

```
if (write) {
    LSB_DDR = DATA_WRITE;
    MSB_DDR = DATA_WRITE;
  }
  else {
    LSB_DDR = DATA_READ;
    MSB_DDR = DATA_READ;
  }
```

After that, we begin the loop that does the actual reading and/or writing. It iterates on a counter from 0 to 512 by two, since block_buffer is made up of 8 bit bytes, and the sector buffer is made up of 16 bit words.

```
for (int c = 0; c < 512; c += 2) {
```

Writing

If we're we're big endian, we set the MSB_PORT to the value of block_buffer[c], and LSB_PORT to block_buffer[c+1]. This flips the order so the bytes are in the right order for a big endian word.

if we're little endian, we set LSB_PORT to block_buffer[c], and MSB_PORT to block_buffer[c+1].

Then we set the drive hardware lines to HW_WRITE.

```
    if (write) {
      if (big_endian) {
        MSB_PORT = block_buffer[c];
        LSB_PORT = block_buffer[c + 1];
      }
      else {
        LSB_PORT = block_buffer[c];
        MSB_PORT = block_buffer[c + 1];
      }
      set_drive_hw_lines(HW_WRITE);
    }
```

Reading

Set the drive signals to HW_READ.

If we're big endian, we read the MSB_PINS into block_buffer[c] and the LSB_PINS into block_buffer[c+1], much the way we did in the write section. This puts them in little endian order in block_buffer, which the rest of the sketch expects.

If we're little endian we read LSB_PINS into block_buffer[c] and MSB_PINS into block_buffer[c+1].

```
else {
      set_drive_hw_lines(HW_READ);
      if (big_endian) {
        block_buffer[c] = MSB_PINS;
        block_buffer[c + 1] = LSB_PINS;
        non_zero_read = MSB_PINS | LSB_PINS | non_zero_read;
      }
      else {
        block_buffer[c] = LSB_PINS;
        block_buffer[c + 1] = MSB_PINS;
        non_zero_read = MSB_PINS | LSB_PINS | non_zero_read;
      }
    }
```

Cleanup and Exit

Finally, we set the drive hardware lines to HW_OFF, a safe position. The loop ends here and we repeat until we've read or written all 256 words.

When the loop exits, we hit some more debug code and the function exits.

```
set_drive_hw_lines(HW_OFF);
  }

#ifdef DEBUG
  Serial.println("transfer_sector_buffer done.");
#endif
}
```

dump_block_buffer()

Dump block buffer is a pretty-printer for the contents of the block_buffer array on the Cestino. It generates two columns of information, the hex dump of the block_buffer, where every byte of data, printable or not, is represented in hexidecimal, and a text print of any printable characters.

It takes no parameters and returns no data.

We begin by initializing a counter, c, and three String objects: hex_data, human_readable_data, and line, to their 0 or empty values.

```
void dump_block_buffer() {
  Serial.println("Dumping Block Buffer");
  int c = 0;
  String hex_data = "";
  String human_readable_data = "";
```

Each line of output will contain hex_data and human_readable_data, plus some formatting information. In order to generate the rows and columns of these two tables in a sensible fashion, we use two more counters, row and col, and two loops.

```
  for (int row = 0; row < 32; row++) {
    for (int col = 0; col < 16; col++) {
```

Here's a catch. Arduino's hex printing will "helpfully" truncate leading zeros from hex values, so 0x05 comes out 0x5. This messes up our table, so if the value at block_buffer[c] is less than 0x10, we add a leading zero to hex_data.

```
      if (block_buffer[c] < 0x10) hex_data += "0";
```

Next, we use the String function from the Arduino library to return the text version of block_buffer[c] as hex. We add that to the hex_data string, then add a blank space.

```
      hex_data += String(block_buffer[c], HEX);
      hex_data += " ";
```

Next, we test to see if block_buffer[c] is a printable character. If it is, we add it, cast as a char, to human_readable_data. If it's not printable, we add a period instead.

```
      if (isprint(block_buffer[c])) {
        human_readable_data += (char)block_buffer[c];
      } else {
        human_readable_data += ".";
      }
```

Then we increment c and repeat the column loop.

```
      c++;
    }
```

Once we exit the column loop, we Serial.println hex data, and a space, a pipe character, and another space, then human_readable_data. Then clear both those variables for re-use. Then we repeat the row loop.

```
    Serial.println(hex_data + " | " + human_readable_data);
    hex_data = "";
    human_readable_data = "";

  }
```

Once both loops have exited, we tell the user how many bytes were dumped (which should always be 512.) Then we exit.

```
  Serial.print("Bytes dumped:");
  Serial.print(c);
  Serial.print("\n");
}
```

read_write_drive_LBA_block()

Here it is, the function that actually makes our drive useable as a storage device. read_write_drive_LBA_block takes a uint32_t block number, and a boolean to determine whether it's writing or reading. It returns no data.

The first thing that happens in this function is we declare a union named block_num_union, with a uint32_t representation called uint, and an array of four bytes called byte_array. As always, both these representations refer to the same area of memory. They're just two ways to get at the same information, and as usual, we're using them to store and access long addresses, in this case a 28 bit LBA. The usual caveat applies, too, that if you're running this sketch on a non-little-endian Arduino, this data structure will return data in the wrong order. We set block_num_union to block_num. using the uint representation.

```
void read_write_drive_LBA_block(uint32_t block_num, bool write_enable) {
  union {
    uint32_t uint;
    byte byte_array[4];
  } block_num_union;
  block_num_union.uint = block_num;
```

Then we get to the usual debug code.

```
#ifdef DEBUG
  Serial.print("read_drive_LBA_block called on block ");
  Serial.print(block_num);
  Serial.print("\n");
#endif
```

Next, when the drive's not busy, we write the first three bytes of our LBA address in block_num to the lowest three LBA address registers using write_command_to_register.

```
  wait_for_drive_not_busy(); //wait for the drive to signal it's not busy.
  write_command_to_register(REG_LBA0, block_num_union.byte_array[0]);
  write_command_to_register(REG_LBA1, block_num_union.byte_array[1]);
  write_command_to_register(REG_LBA2, block_num_union.byte_array[2]);
```

The last four bytes of the LBA address are a little more complicated, since we have to also apply the LBA_MODE command to the high bits of the same register or our LBA addresses won't work. We do this by ORing CMD_LBA_MODE to the third byte of the LBA address. We're not changing any of the address bits, just turning on a bit in the other half of the register. After that, we wait for the drive's busy signal to go off.

```
write_command_to_register(REG_LBA3, (block_num_union.byte_array[3] |
CMD_LBA_MODE));
  wait_for_drive_not_busy();
```

If we are writing, and the write_enable boolean is true, send the command CMD_WRITE_WITH_RETRY to REG_STAT_COM to initiate the write. Otherwise send the command CMD_READ_WITH_RETRY to initiate the read.

```
if (write_enable) {
    write_command_to_register(REG_STAT_COM, CMD_WRITE_WITH_RETRY);
  } else {
    write_command_to_register(REG_STAT_COM, CMD_READ_WITH_RETRY);
  }
```

Then we wait until the drive tells us it has data for us. When it does, we call transfer_sector_buffer, pass it write_enable so if we're writing, so is it, and tell it we're little endian. We also reset the drive. We shouldn't need to, but breadboards with multiple power supplies and big ribbon cables attached (like ours) can get very noisy. This is a kludge to fix a bug where, in a noisy environment, reads following writes would read the wrong sector.

```
wait_for_drive_drq();
  transfer_sector_buffer(write_enable, LITTLE_ENDIAN);
  reset_drive();
```

Cue the debug code and exit.

```
#ifdef DEBUG
  Serial.println("read_write_drive_LBA_block done.");
#endif
}
```

High Level Functions

All done debugging? You're sure? Okay, let's go on to the high level functions, including the final versions of setup() and loop()

reset_drive()

This function resets the drive. It's really, really short. So short, in fact, that the debug code almost outweighs the code. It takes no parameters and returns no data. First, we have the usual debug boilerplate.

```
void reset_drive() {
#ifdef DEBUG
  Serial.println("reset_drive() called.");
#endif
```

Then we call set_drive_hw_lines to set the signals to the drive to the HW_RESET pattern. This means set /Reset low. Delay 1ms, then set it high again.

```
  set_drive_hw_lines(HW_RESET);
  delay(1);
  set_drive_hw_lines(HW_OFF);
```

For good measure, we send the drive a CMD_INIT, too, and wait a little more than a second for the drive to come back. Why 1024ms? Why not?

```
  write_command_to_register(REG_HEAD, CMD_INIT);
  wait_for_drive_ready(1024);
```

After that is the usual debug code, and we exit.

```
#ifdef DEBUG
  Serial.println("reset complete.");
#endif
}
```

sleep_drive()

Even simpler than resetting the drive is sleeping the drive. The usual debug boilerplate /does/ outweigh the code on this one. Sleep_drive() returns no data and takes no parameters. We begin with the usual debug code.

```
void sleep_drive() {
#ifdef DEBUG
  Serial.println("sleep_drive() called.");
#endif
```

Then we write CMD_SLEEP to REG_STAT_COM. That's it. That's all it does. The drive should spin down, and most other functions that wait for signals on the drive? Will hang waiting for those signals. You have to reset the drive to wake it up.

```
write_command_to_register(REG_STAT_COM, CMD_SLEEP);
```

Then we exit after the usual debug code.

```
#ifdef DEBUG
  Serial.println("ZZZ");
#endif
};
```

Identify Drive

This function is actually the first high level function I got to work. It's a good test, as it exercises most of the low level functions and several of the utilities without requiring the drive to actually do anything other than copy data from ROM to the sector buffer. Identify Drive takes no parameters and returns no data. We begin with the usual boilerplate debug code.

```
void identify_drive() {
#ifdef DEBUG
  Serial.println("identify_drive() called");
#endif
```

Next, we wait for the busy flag to be clear, which it most likely is.

```
wait_for_drive_not_busy();
```

After that, we write CMD_IDENTIFY to the status/command register, and transfer the sector buffer to the Cestino's block_buffer[] array.

■ **Note** The drive identity information is big endian. We have to send that flag to transfer_sector_buffer or the information will be unreadable to us.

Also, because the number of LBA blocks is sent in a 32 bit big endian number and transfer_sector_buffer only handles 16 bit big endian, we have to swap the two 16 bit words of that value when we use them. Which we will.

We don't dump it because, as you'll see, sometimes we want to do additional processing on it to make it human-readable.

```
  write_command_to_register(REG_STAT_COM, CMD_IDENTIFY);
  transfer_sector_buffer(false, BIG_ENDIAN);
```

We exit with the usual debug code. If you think writing the debug code was a bit tedious? It was.

```
#ifdef DEBUG
  Serial.println("identify_drive() done.");
#endif
}
```

setup()

The setup function is normally short, almost trivial except that serial communications won't work without it. Not so in this sketch. We do a lot of setup to set the drive up, and we also identify the drive and display its ID information. All this before loop even starts. As always, setup returns no data and takes no parameters.

We begin with the usual serial setup, and a loop that waits until serial actually wakes up before the sketch will run. Your terminal window must be open to run this sketch.

```
void setup() {
  Serial.begin(115200);
  while (!Serial) {
  }
  CONTROL_DDR = DATA_WRITE;
```

Of course there's debug code. This one warns us that debug is set. With all the functions in place, it's fairly obvious when global debug is set, but for completeness, it's here.

```
#ifdef DEBUG
  Serial.println("DEBUG is set.");
#endif
```

Next, we reset whatever drive is out there, then try to read the status byte. If it's all 1s, there's no drive operational out there. The sketch goes into an infinite loop.

You either need to plug the drive in, connect it, or you've got other problems. When I smoked my demo drive just now by shorting its ID pins, I got the floating bus message. Fortunately, I had another.

```
  reset_drive();
  Serial.print("Checking for a drive...");
  byte status_byte = get_drive_status_byte();
  if (get_drive_status_byte() == 255) {
    Serial.println("Floating bus - Drive not detected.");
    while (true) {
      //loop forever.
    }
  }
```

If we do find a drive out there, tell the user, and call identify_drive(). We need to know this information so we don't try and send it to LBA addresses it hasn't got.

```
else
  {
    Serial.println("Found a drive."); //Got valid drive
    identify_drive();
```

Next, we fish through the block buffer to pull various strings out for drive information. Most of these pieces of information have a standard location in the sector buffer (and from there, our block_buffer[] array) defined in the spec. All these loops essentially do the same thing. Start at a known location, read a byte, serial print it as a character.

```
    Serial.print("Model: ");
    for (int c = 54; c <= 74; c++) {
      Serial.print((char)block_buffer[c]);
    }

    Serial.print("\nSerial Number: ");
    for (int c = 20; c <= 30; c++) {
      Serial.print((char)block_buffer[c]);
    }

    Serial.print("\nFirmware Version: ");
    for (int c = 46; c <= 50; c++) {
      Serial.print((char)block_buffer[c]);
    }
```

The numeric values are a little different. Some of them need some extra processing, notably combining into 16 bit numbers. So we rotate the first byte left by 8 and add the second.

```
    drive_cylinders = block_buffer[2] << 8;
    drive_cylinders += block_buffer[3];
    Serial.print("\nCylinders: ");
    Serial.print(drive_cylinders);
    drive_heads = block_buffer[6] << 8;
    drive_heads += block_buffer[7];
    Serial.print("\nHeads: ");
    Serial.print(drive_heads);
    drive_sectors = block_buffer[12] << 8;
    drive_sectors += block_buffer[13];
    Serial.print("\nSectors: ");
    Serial.print(drive_sectors);
```

LBA sectors are a 32 bit number, and because of how our 16 bit endian switcher works, we get digits out of place. Because it only happens here, we just fix it manually.

```
    Serial.print("\nLBA Sectors: ");
    LBA_sectors = block_buffer[122];
    LBA_sectors = (LBA_sectors << 8) | block_buffer[123];
    LBA_sectors = (LBA_sectors << 8) | block_buffer[120];
    LBA_sectors = (LBA_sectors << 8) | block_buffer[121];
    Serial.print(LBA_sectors);
  }
```

And here, as promised, is the Serial.println() that makes sure everything is actually sent to the terminal. Then we exit.

```
  Serial.println();
}
```

loop()

For once, we're actually going to let loop() run more than once. The loop function defines, then generates a menu of options we can do to the drive. our choice is then fed into a case statement that implements the actual code for the option we've chosen. As always, loop takes no parameters and returns no functions. It repeats forever, unless we park the Cestino in our own repeat-forever loop. Which we do in "Quit".

The first thing we do is define the menu text. It's one big, hairy #define, and it's important that the text of the define be against the left margin, as any spaces introduced will be present in the menu when it's printed. It's just text, though. All it would do is look bad.

```
void loop() {
#define menu "\n\n**** MENU ****\n\
1\tReset Drive\n\
2\tSleep Drive\n\
3\tIdentify Drive\n\
4\tSeek Non-Empty Block/Sector\n\
5\tWrite to Block/Sector\n\
6\tQuit\n"
```

Now that our menu's defined, let's initialize our selector variable to 0. Or 0x00. Our switch will work on single characters, and the characters it looks for will be numbers. To avoid confusion between numbers and their /character/ values, I put the character values in hex. After that, we display the menu and wait for the user to press return in the little text line at the top of the serial window.

```
char menu_item = 0x00; //Initialize menu_item to 0.

Serial.println(menu); //display the menu
while (!Serial.available()) { //wait for serial input
  //do nothing
}
```

When we reach this point, the user has typed /something/ and pressed return. We read what they typed with Serial.read() and store it in menu_item.

```
menu_item = Serial.read();
```

After that, we get into the case statement that is the bulk of this function. Some of the options call one function. Others call a few functions. Some should probably be functions, but it's always a borderline case when a piece of code is only useful in one place whether to separate it out into a function or just leave it.

Options 1 and 2 are nearly identical. They call the named function. 1 resets the drive and waits for the drive to become ready, then breaks. 2 sleeps the drive and doesn't wait for anything. Then breaks.

In case you're not familiar with case statements, if you don't put a break at the end of the code for an option, case will execute every option that follows the first match. It's one of those annoying things about C. Also annoying: in C you begin a case statement with switch. Go figure.

```
switch (menu_item) {
  case 0x31 : {
      //0x31 is "1". Reset the drive, wait until ready.
      Serial.println("Resetting drive.");
      reset_drive();
      wait_for_drive_ready(1024);
      Serial.println("Drive is reset.");
      break;
    }
  case 0x32 : { //0x32 is "2" Sleep the drive.
      Serial.print("Sleeping Drive. ");
      Serial.println("Reset to wake up.");
      Serial.println("(Other functions will hang.)");
      sleep_drive();
      break;
    }
```

Option 3 identifies the drive. This is a three step process. After we tell the user what was selected, we call identify_drive(), which sends the identify command to the drive and transfers the sector buffer into our block_buffer[] array. Then we call dump_block_buffer to display the identity information in its raw form.

```
case 0x33 : { //0x33 is "3". Identify Drive.
    Serial.println("Identifying Drive");
    identify_drive(); //put ID info in the block buffer
    dump_block_buffer(); //dump the block buffer
    break;//Don't try the other cases.
  }
```

Option 4 finds the first non-empty block and dumps it, then asks the user if they want to continue. It begins by initializing a new uint32_t called sector to 0, (remember, in LBA blocks and sectors are essentially the same thing) and a byte called keep_going to 0x79, which is lowercase y. If you expect a loop to start next, you've got a good ear.

```
case 0x34 : {
    Serial.println("Seeking Non-empty Block/Sector.");
    uint32_t sector = 0;
    byte keep_going = 0x79; // lower case y
```

Here's the loop. While keep_going is "y", read the sector. If sector modulus 100 is zero (sector is an even hundred) tell the user we're checking this block, so they get a count of the sectors we're ignoring. Finding a sector that has stuff in it can take quite a while and we'd like them to know the sketch is still running.

```
    while (keep_going == 0x79) {
      read_write_drive_LBA_block(sector, false);

      if (!(sector % 100)) {
        Serial.println("Checking Block " +
                       (String)sector);
      }
```

If we find a block where `non_zero_read` is true, tell the user what block it is, and call `dump_block_buffer`. Then ask to continue. Wait forever for a response. When they press return, put whatever they typed into `keep_going`, increment sector, and go back to the beginning of this loop. If we fall out of the loop, break so we exit out of the case statement.

```
      if (non_zero_read) {
        Serial.print("Found something. Dumping sector ");
        Serial.print(sector);
        Serial.print("\n");
        dump_block_buffer();
        Serial.println("Shall I continue? (y/n)");
        while (!Serial.available()) {
```

```
            }
            keep_going = Serial.read();
          }
          sector++;
        }
        break; //Don't try the other cases.
      }
```

Option 5 lets the user seek a specific block, then write a string to that block. It then rereads the block and dumps it to the terminal. Because this sketch knows nothing at all about filesystems, this option basically corrupts the disk. It was in the junk box anyway, right? Option 5 begins by initializing a new string called input_string empty, and a new uint32_t called block to 0. Could I have reused sector? Yes.

```
    case 0x35 : {
        String input_string = "";
        uint32_t block = 0;
```

Next, we ask the user what block they would like written to. We also tell them how many sectors the drive has, and that we only have 28 bit addresses to work with. Newer versions of the ATA standard go to 48 bit LBA addresses, but I don't have access to those standards, and most of drives that big are SATA.

```
        Serial.println("What LBA block shall I write to? ");
        Serial.print("The drive has ");
        Serial.print (LBA_sectors);
        Serial.print(" LBA blocks, and 28 bit LBA addresses go to");
        Serial.println("268435456.");
        while (!Serial.available()) {
        }
        input_string = Serial.readString();
```

Call string2uint32_t on the string the user gave us, clear input string, and ask the user for their message to write on the disk. We'll store that in input_string, too.

```
        block = string2uint32_t(input_string);
        input_string = "";//clear input string.
        Serial.println("Writing to block " + (String)block +
                       ". Please type your message and" +
                       " press return.");
        while (!Serial.available()) {
          //do nothing
        }
        input_string = Serial.readString();
```

Next, we tell the user what they're writing and to where. Note that the \" marks are escaped so they'll show up in the Serial.println(). Once that's finished, we copy the string into block_buffer, and set any unused bytes of block_buffer to 0.

```
        Serial.println("Writing \"" + input_string +
                       "\" to Block:" + (String)block);

        for (int c = 0; c <= 512; c++) {
          if (c <= input_string.length()) {
            block_buffer[c] = input_string.charAt(c);
          } else block_buffer[c] = 0;
        }
```

Finally, we call read_write_drive_LBA_block with the block number and "true" to tell it that it is write enabled. This copies block_buffer[] to the drive. Technically, we copy it to the drive's sector buffer, but the drive writes to the physical disk(s) at the same time. Then wait 1024ms for the drive to signal that it's ready again.

```
        read_write_drive_LBA_block(block, true);
        wait_for_drive_ready(1024);
        Serial.println("Done writing. Reading block "
                       + (String)block);
```

Reread the block. We don't fake it by just playing back our own block_buffer[] array, we read the drive and get a fresh copy. It's the only way to be sure the write really happened. Once we read it, we dump it, clear input_string, and break.

```
        read_write_drive_LBA_block(block, false);
        dump_block_buffer();
        input_string = "";
        break;//Don't try the other cases.
      }
```

Option 6 is Quit. We sleep the drive, then go into an infinite loop. Only resetting the Cestino will restart the sketch now. Yes, there's a break. No, we'll never hit it.

```
    case 0x36 : {
        Serial.println("Sleeping Drive and Exiting.");
        sleep_drive();
        Serial.println("Halted. Reset Cestino to Restart.");
        while (true) {
        }
        break;
      }
  }
```

When any of the case statement options break, they drop out the bottom of the case and wind up here, where we clear menu_item, and fall out the bottom of the function. Because loop() is called and recalled forever, the Arduino core will handle getting us the next menu. And the next. And the next.

```
  menu_item = 0x00; //clear menu_item for the next go-round.
```

The Complete Sketch

The complete sketch, including voluminous comments, is here, as always.

```
//Hardware:
//-------------------------------------------------------------
// CONTROL_PORT
// Signal: /CS3FX /CS1FX DA2 DA1 DA0 /DIOW /DIOR /RESET
// Means: 3F6-3F7 1F0-1F7 (Drive Addr) (Write)(Read) (Reset)
// IDE PIN: 38       37    36 33   35   23      25     1
// Port Bit: 7        6     5  4    3    2       1     0

// LSB_PORT
// Signal:  DD7 DD6  DD5  DD4  DD3  DD2 DD1 DD0
// IDE Pin:   3   5    7    9   11   13  15  17
// Port Bit:  7   6    5    4    3    2   1   0

// MSB_PORT
// Signal: DD15 DD14 DD13 DD12 DD11 DD10 DD9 DD8
// IDE Pin:  18   16   14   12   10    8   6   4
// Port Bit:  7    6    5    4    3    2   1   0
//
// GROUND:
// IDE pins 2, 19, 22, 24, 26, 30, and 40 should be grounded.
// Yes really, ground them all.
//-------------------------------------------------------------

//Notes
//-------------------------------------------------------------
// Sector and LBA block are used interchangeably in this
// sketch, because when the drive is identifying itself,
// it's still in cylinder/head/sector mode. All other
// reads and writes deal with LBA blocks. Which some
// people also call LBA sectors.
```

```
//Preprocessor #defines
//-------------------------------------------------------------
// This sketch has a ton of preprocessor #defines.
// They use no run-time memory and make the code easier to
// follow.
//-------------------------------------------------------------

//Define what ports do what, and how to access their PORT,
//PIN, and DDR registers.
#define CONTROL_PORT PORTB
#define CONTROL_DDR DDRB
#define CONTROL_PINS PINB
#define LSB_DDR DDRA
#define LSB_PORT PORTA
#define LSB_PINS PINA
#define MSB_DDR DDRC
#define MSB_PORT PORTC
#define MSB_PINS PINC

// Define the two DDR modes for any port exc. control
#define DATA_READ 0b00000000
#define DATA_WRITE 0b11111111

// Define various control port register settings.
// Note that this assumes the control port will be wired
// as shown below, in 0bXXXXXXXX notation.
// bit 0 is on the RIGHT.
// /CS3FX,/CS1FX,DA2,DA1,DA0,/DIOW,/DIOR, and /RESET
#define REG_DEFAULT      0b11000111 //bogus register
#define REG_STAT_COM     0b10111111 //Status/Command Register
#define REG_ASTAT_DEVCOM 0b01110111 //Alt Status/Device Command
#define REG_ERR_FEAT     0b10000111 //Error/Features
#define REG_LBA0         0b10011111 //Low byte of LBA address
#define REG_LBA1         0b10100111 //Second byte LBA address
#define REG_LBA2         0b10101111 //Third byte LBA address
#define REG_LBA3         0b10110111 //Fourth byte LBA address
#define REG_DATA         0b10000111 //Data register
#define REG_HEAD REG_LBA3 //Same register, different modes.

// Most actions require a command to be sent
// over the LSB_PORT data lines once the
// register is selected.
// Those commands are defined here.
#define CMD_SLEEP 0x99 //Put the drive to sleep/spin down.
#define CMD_INIT 0x91 //Set drive to a known state.
#define CMD_IDENTIFY 0xEC //Get ID info -> drive sector buffer
```

```
#define CMD_READ_WITH_RETRY 0x20 //Read block to drive buffer.
#define CMD_WRITE_BUFFER 0xE8 //Write data to drive buffer
#define CMD_WRITE_WITH_RETRY 0x30 //Write drive buff. to disk.
#define CMD_LBA_MODE 0b01000000 //Set the drive to LBA mode.

// Status byte masks (Cribbed from GeneT's IDEFAT library.)
// I've annotated them in binary to make it clearer
// how they work. These are masks used with AND to pull
// one bit from a status register read.
// All status bits are active high. In order, they are:
// BSY DRDY DWF DSC DRQ CORR IDX ERR
#define BSY  0x80 //0b10000000 - Busy
#define RDY  0x40 //0b01000000 - Ready
#define DF   0x20 //0b00100000 - Drive Write Fault
#define DSC  0x10 //0b00010000 - Drive Seek Complete
#define DRQ  0x08 //0b00001000 - Data Request
#define CORR 0x04 //0b00000100 - Corrected Data
#define IDX  0x02 //0b00000010 - Index
#define ERR  0x01 //0b00000001 - Error

//Define modes of set_drive_hw_lines - Signals are
// /DIOW,/DIOR, and /RESET
#define HW_OFF   0b111 //Set the read-write lines all off.
#define HW_READ  0b101 //Set the read-write lines for reading.
#define HW_WRITE 0b011 //Set the read-write lines for writing.
#define HW_RESET 0b110 //Lower the reset line to reset.

// Define BIG_ENDIAN as true and LITTLE_ENDIAN as false.
// It makes the switch in transfer_sector_buffer clearer.
#define BIG_ENDIAN true
#define LITTLE_ENDIAN false

// DEBUG mode compiles in a lot of Serial.writeln messages.
// for debugging purposes. Can also be #defined for individual
// functions and #undef'd at the end of the function.
//#define DEBUG

//Code
//-------------------------------------------------------------
// Global Variables and functions. The meat of the sketch.
//-------------------------------------------------------------

//-------------------------------------------------------------
// GLOBAL VARIABLES
//-------------------------------------------------------------
```

```
byte block_buffer[512]; //One buffer for reading and writing.
int drive_cylinders = 0;
int drive_heads = 0;
int drive_sectors = 0;
uint32_t LBA_sectors = 0;
boolean non_zero_read = false;
//Last block buffer transfer empty?

//-------------------------------------------------------------
// Function get_drive_status_byte()
// - Select the REG_ASTAT_DEVCOM register and read it.
//-------------------------------------------------------------
byte get_drive_status_byte() {

#ifdef DEBUG
  Serial.print("get_drive_status_byte() called.");
#endif

 byte temp_control_port = CONTROL_PORT;
 byte temp_lsb = LSB_PORT;
 byte temp_lsb_ddr = LSB_DDR;
 //save port states

  byte status_byte = 0;

  CONTROL_PORT = REG_ASTAT_DEVCOM;
  //Use the alternate status register.

  LSB_DDR = DATA_READ; //set LSB port to read mode.
  LSB_PORT = 0b11111111; //Turn on pullup resistors - reading.
  set_drive_hw_lines(HW_READ); //Set the read signals
  status_byte = LSB_PINS; //Read LSB_PINS into status_byte.
  set_drive_hw_lines(HW_OFF); //Clear the read signals.

  CONTROL_PORT = temp_control_port;
  LSB_PORT = temp_lsb;
  LSB_DDR = temp_lsb_ddr;
  //restore port states

#ifdef DEBUG
  Serial.print(" Returning:");
  Serial.print(status_byte, BIN);
  Serial.print("\n");
#endif
  return (status_byte);
};
```

```
//-------------------------------------------------------------
// Function write_command_to_register(reg,command)
// Set CONTROL_PORT to reg, then write a command byte
// to it on LSB_PORT
//-------------------------------------------------------------
void write_command_to_register(byte reg, byte command) {
  byte temp = LSB_DDR; //Preserve LSB r/w state
  LSB_DDR = DATA_WRITE; //Set LSB's DDR
  CONTROL_PORT = reg; //Set CONTROL_PORT to register address.
  LSB_PORT = command; //Set LSB_PORT to the command byte.
  set_drive_hw_lines(HW_WRITE); //Strobe write/read lines.
  set_drive_hw_lines(HW_OFF);
  LSB_DDR = temp; //Restore the state of LSB's DDR.
}

//------------------------------------------------------------
// Function set_drive_hw_lines(byte status)
// These are the last three lines of CONTROL_PORT.
// We switch our read, write, and reset signals
// here, and they're all active low. So it's backwards.
//------------------------------------------------------------
void set_drive_hw_lines(byte status) {
#ifdef DEBUG
  Serial.print("set_drive_hw_lines() called with status:");
  Serial.print(status, DEC);
  Serial.print(" CONTROL_PORT was: ");
  Serial.print(CONTROL_PORT, BIN);
#endif

  byte temp = CONTROL_PORT;
  //preserve CONTROL_PORT's existing state.

  temp = temp & 0b11111000;
  //wipe the hw line bits

  temp = temp | status;
  //OR the remaining bits with the new setting (status)

  CONTROL_PORT = temp; //Set CONTROL_PORT to the new value.

#ifdef DEBUG
  Serial.print(" CONTROL_PORT now ");
  Serial.print(CONTROL_PORT, BIN);
  Serial.print("\n");
#endif
}
```

```
//-------------------------------------------------------------
// Function wait_for_drive_drq();
// Do nothing while checking to see if the DRQ line is high
// DRQ is data request. The drive signals it's ready to transfer
// a word of data (8 or 16 bits, in or out) with this signal.
//-------------------------------------------------------------
void wait_for_drive_drq() {

#ifdef DEBUG
  Serial.print("wait_for_drive_drq() called: ");
#endif
  byte status_byte = get_drive_status_byte();
  byte mask = DRQ | BSY; //check bits 3 and 7, DRQ and BSY.
  while (!((status_byte & mask) == DRQ)) {
    status_byte = get_drive_status_byte();
  }
  //if we don't get a DRQ and BSY,check again until we do.

#ifdef DEBUG
  Serial.println("DRQ");
#endif
}

//-------------------------------------------------------------
// Function wait_for_drive_not_busy()
// Do nothing while we wait for the drive to go not busy.
//-------------------------------------------------------------
void wait_for_drive_not_busy() {

#ifdef DEBUG
  Serial.print("wait_for_drive_not_busy() called: ");
#endif
  byte status_byte = get_drive_status_byte();
  byte mask = BSY;
  while (!((status_byte & mask) == 0)) {
    status_byte = get_drive_status_byte();
    //while the highest bit of the status byte is not 0,
    //get the status byte. Do it forever if need be.
  }
#ifdef DEBUG
  Serial.println("Busy Clear");
#endif
}
```

```
//-------------------------------------------------------------
// Function wait_for_drive_ready(int ms to wait)
// Do nothing while we wait up to ms_to_wait ms for the drive
// to go ready. It's a crude timeout mechanism, but it works.
//-------------------------------------------------------------
void wait_for_drive_ready(int ms_to_wait) {

#ifdef DEBUG
  Serial.print("wait_for_drive_ready() called: ");
#endif
  byte status_byte = get_drive_status_byte();
  byte mask = BSY | RDY;
  //look at the highest two bits of the status byte.

  int c = 0;
  while ((!((status_byte & mask) == RDY)) && (c <= ms_to_wait)) {
    status_byte = get_drive_status_byte();
    delay(1);
    c++;
  }
  //While the highest two bits of the status byte are not
  //equal to 0x40 (RDY), delay 1ms, increment c,
  //and do it again.

#ifdef DEBUG
  Serial.println("DRIVE READY");
#endif
}

//-------------------------------------------------------------
// string2uint32_t()
// - The String class includes no nice way to turn strings of
//   digits into a numeric value.
//   This function goes through the string from left to right.
//   For each position it advances to the right, it multiplies
//   the existing value by 10 and adds the value of the
//   character at that position minus the value of the
//   character '0'. When we reach the end of the string,
//   return temp.
//-------------------------------------------------------------
uint32_t string2uint32_t(String input) {
  uint32_t temp = 0;
  input.trim();
  for (int c = 0; c < input.length(); c++) {
    temp = temp * 10 + input.charAt(c) - '0';
  }
```

```
#ifdef DEBUG
  Serial.print("string2uint32_t called with a string of >" +
               input + "< Returned:");
  Serial.println(temp, DEC);
#endif

  return temp;
}

//------------------------------------------------------------
// Function transfer_sector_buffer(write,big_endian)
// - copy block_buffer to/from the drive.
//
// NOTE BENE: Drive info is stored big_endian
// but nearly all microcomputers store data
// in little_endian.
//------------------------------------------------------------
void transfer_sector_buffer(boolean write, boolean big_endian) {
#ifdef DEBUG
  Serial.println("transfer_sector_buffer called.");
#endif

  wait_for_drive_not_busy();
  non_zero_read = false; //set false before we start.
  CONTROL_PORT = REG_DATA; //select the REG_DATA register.

  if (write) {
    LSB_DDR = DATA_WRITE;
    MSB_DDR = DATA_WRITE;
    //set both LSB and MSB to write mode.
  }
  else {
    LSB_DDR = DATA_READ;
    MSB_DDR = DATA_READ;
    //Set both LSB and MSB to read mode.
  }

  for (int c = 0; c < 512; c += 2) {
    if (write) { //If we're writing...
      if (big_endian) { //and we're big endian
        MSB_PORT = block_buffer[c];
        LSB_PORT = block_buffer[c + 1];
      }
      else { //or we're little endian
       LSB_PORT = block_buffer[c];
       MSB_PORT = block_buffer[c + 1];
      }
```

```
      set_drive_hw_lines(HW_WRITE);//set write signals
    }
    //For big endian writing, put the cestino's buffer into the
    //drive's buffer in pairs, second value first, because
    //the Cestino's sector buffer is little endian.
    //Otherwise put the values into LSB and MSB in the order
    //they are in the Cestino's buffer.

    else {//if we're reading
      set_drive_hw_lines(HW_READ); //set read signals
      if (big_endian) { //if we're big endian
        block_buffer[c] = MSB_PINS;
        block_buffer[c + 1] = LSB_PINS;
        non_zero_read = MSB_PINS | LSB_PINS | non_zero_read;
      }
      else {
        block_buffer[c] = LSB_PINS;
        block_buffer[c + 1] = MSB_PINS;
        non_zero_read = MSB_PINS | LSB_PINS | non_zero_read;
      }
    }
    //For big endian reading, store the MSB byte first, then
    //store the LSB byte. Check to see if either MSB or LSB
    //had anything in them, or if non_zero_read was already
    //set true. If any of those are true, non_zero_read is
    //true. Probably this should be in a function return.

    set_drive_hw_lines(HW_OFF);//Turn the rw signals off.
  }

#ifdef DEBUG
  Serial.println("transfer_sector_buffer done.");
#endif
}

//-------------------------------------------------------------
// Function dump_block_buffer()
// - Pretty-print the block buffer.
//-------------------------------------------------------------
void dump_block_buffer() {
  Serial.println("Dumping Block Buffer");
  int c = 0;
  String hex_data = "";
  String human_readable_data = "";
  //We have some variables. Initialize them.
```

```
  for (int row = 0; row < 32; row++) {
    for (int col = 0; col < 16; col++) {
      //We're printing blocks of 512 bytes.
      //each is 2 characters + two spaces
      //in hex, plus 1 character in text.

      if (block_buffer[c] < 0x10) hex_data += "0";
      //Add leading zero for hex values below 0x10.

      hex_data += String(block_buffer[c], HEX);
      hex_data += " ";
      //Add block_buffer[c] as a hex string to hex_data.

      if (isprint(block_buffer[c])) {
        human_readable_data += (char)block_buffer[c];
      } else {
        human_readable_data += ".";
      }
      //If block_buffer[c] is printable, add it to
      //human_readable_data. Otherwise add a . for
      //a placeholder.
      c++;
    }

    Serial.println(hex_data + " | " + human_readable_data);
    hex_data = "";
    human_readable_data = "";
    //Combine the two strings in one Serial.println.
    //Then clear them.

  }
  Serial.print("Bytes dumped:");
  Serial.print(c);
  Serial.print("\n");
}
//-------------------------------------------------------------
// Function read_write_drive_LBA_block(block_num)
//
// -LBA mode only. Set the drive to LBA mode
// and put the contents in the drive's block buffer.
// Then read the drive's block buffer into the
// Cestino's block buffer. Block numbers are 28 bits,
// and little-endian.
//-------------------------------------------------------------
```

```
void read_write_drive_LBA_block(uint32_t block_num, bool write_enable) {
  union {
    uint32_t uint;
    byte byte_array[4];
  } block_num_union;
  //This union takes a uint32_t (4 byte unsigned int)
  //and represents it also as an array of 4 bytes. Very handy,
  //but watch for endian problems if you run this on a non
  //ATmega Arduino.

  block_num_union.uint = block_num;

#ifdef DEBUG
  Serial.print("read_drive_LBA_block called on block ");
  Serial.print(block_num);
  Serial.print("\n");
  Serial.println("third byte is:" + String(block_num_union.byte_array[3],
BIN));
#endif

  wait_for_drive_not_busy(); //wait for the drive to signal it's not busy.
  write_command_to_register(REG_LBA0, block_num_union.byte_array[0]);
  write_command_to_register(REG_LBA1, block_num_union.byte_array[1]);
  write_command_to_register(REG_LBA2, block_num_union.byte_array[2]);
  write_command_to_register(REG_LBA3, (block_num_union.byte_array[3] |
CMD_LBA_MODE));
  //Set all three-and-a-half LBA address bytes.
  //LBA3 (aka the head register) gets four bits of address,

  if (write_enable) {
    write_command_to_register(REG_STAT_COM, CMD_WRITE_WITH_RETRY);
  } else {
    write_command_to_register(REG_STAT_COM, CMD_READ_WITH_RETRY);
  }
  wait_for_drive_drq();
  transfer_sector_buffer(write_enable, LITTLE_ENDIAN);
  reset_drive();
  //Kludge to fix write problems when the ATA bus is noisy.

#ifdef DEBUG
  Serial.println("read_write_drive_LBA_block done.");
#endif
}
```

```
//------------------------------------------------------------
// Function reset_drive()
// Raises the reset line, then inits the drive to default mode.
//------------------------------------------------------------
void reset_drive() {
#ifdef DEBUG
  Serial.println("reset_drive() called.");
#endif

  set_drive_hw_lines(HW_RESET); //Lower the /Reset line.
  delay(1);
  set_drive_hw_lines(HW_OFF);

  write_command_to_register(REG_HEAD, CMD_INIT);
  //Init the drive

  wait_for_drive_ready(1024);
  //Wait for the drive to show ready again.

#ifdef DEBUG
  Serial.println("reset complete.");
#endif
}

//-------------------------------------------------------------
// Function sleep_drive()
//  - Set the control port to the status/command register.
//  Set it write, and send the sleep code over the data lines.
//-------------------------------------------------------------
void sleep_drive() {
#ifdef DEBUG
  Serial.println("sleep_drive() called.");
#endif

  write_command_to_register(REG_STAT_COM, CMD_SLEEP);
  //Send the sleep command to REG_STAT_COM register.
  //Most other functions except reset will fail to
  //work due to waiting on ready/not busy/drq signals.

#ifdef DEBUG
  Serial.println("ZZZ");
#endif
};
```

```
//--------------------------------------------------------------
// Function identify_drive()
//
// - Read the drive's identity block. The whole thing. Burp.
// NOTA BENE: drive info is big-endian, so we tell
// transfer_sector_buffer to read that way. Also, 32 bit
// values will be transferred as two sixteen bit words
// with the least significant word first, which is backwards.
// Fortunately there's only one, so we can fix it manually.
//--------------------------------------------------------------
void identify_drive() {
#ifdef DEBUG
  Serial.println("identify_drive() called");
#endif

  wait_for_drive_not_busy(); //Wait for bsy flag to be clear.
  write_command_to_register(REG_STAT_COM, CMD_IDENTIFY);
  transfer_sector_buffer(false, BIG_ENDIAN);
  //Send the identify command to the REG_STAT_COM register
  //Then transfer the drive's buffer to the Cestino's.

#ifdef DEBUG
  Serial.println("identify_drive() done.");
#endif
}

//--------------------------------------------------------------
// Function setup() - Arduino Boilerplate. Called once.
//--------------------------------------------------------------
void setup() {
  Serial.begin(115200);
  while (!Serial) {
  }
  CONTROL_DDR = DATA_WRITE;
  //Set up serial and set the control port to write mode.

#ifdef DEBUG
  Serial.println("DEBUG is set.");
#endif
  //If global debug is set, the user is about to
  // get innundated with debug text. Warn them...

  reset_drive(); //Any drive out there, reset it.
  Serial.print("Checking for a drive...");
  byte status_byte = get_drive_status_byte();
  //Get the status byte from any drive connected.
```

```
if (get_drive_status_byte() == 255) {

  Serial.println("Floating bus - Drive not detected.");
  while (true) {
    //loop forever.
  }
} //If no drive is detected, the bus floats high. Stop.

else
{
  Serial.println("Found a drive."); //Got valid drive
  identify_drive();
  //pull drive identity information into the block buffer.

  Serial.print("Model: ");
  for (int c = 54; c <= 74; c++) {
    Serial.print((char)block_buffer[c]);
  }

  Serial.print("\nSerial Number: ");
  for (int c = 20; c <= 30; c++) {
    Serial.print((char)block_buffer[c]);
  }

  Serial.print("\nFirmware Version: ");
  for (int c = 46; c <= 50; c++) {
    Serial.print((char)block_buffer[c]);
  }

  drive_cylinders = block_buffer[2] << 8;
  drive_cylinders += block_buffer[3];
  Serial.print("\nCylinders: ");
  Serial.print(drive_cylinders);
  drive_heads = block_buffer[6] << 8;
  drive_heads += block_buffer[7];
  Serial.print("\nHeads: ");
  Serial.print(drive_heads);
  drive_sectors = block_buffer[12] << 8;
  drive_sectors += block_buffer[13];
  Serial.print("\nSectors: ");
  Serial.print(drive_sectors);
  Serial.print("\nLBA Sectors: ");
  LBA_sectors = block_buffer[122];
  LBA_sectors = (LBA_sectors << 8) | block_buffer[123];
  LBA_sectors = (LBA_sectors << 8) | block_buffer[120];
  LBA_sectors = (LBA_sectors << 8) | block_buffer[121];
  Serial.print(LBA_sectors);
```

```
    //Why aren't we reading these bytes in order? They are
    //a 32 bit number in big-endian. Our endian converter
    //only understands 16 bit values. This is the only
    //32 bit field we have to deal with, so we grind it
    //manually.

    //Scrape the data in the block buffer for Model,
    //serial number, firmware version, etc.
    //Display those values.

  }
  Serial.println();
} //yay. Done with setup.

//-------------------------------------------------------------
// Function loop() - Arduino Boilerplate. Called repeatedly.
// Create the menu text with a big, hideous #define
// Reset the drive and wait until it's ready.
// Print the menu and wait for serial input.
// Process the menu entry and call one of our functions
//  Seek non-empty and write are more involved
// Since we're in loop(), repeat forever.
//-------------------------------------------------------------
void loop() {
#define menu "\n\n**** MENU ****\n\
1\tReset Drive\n\
2\tSleep Drive\n\
3\tIdentify Drive\n\
4\tSeek Non-Empty Block/Sector\n\
5\tWrite to Block/Sector\n\
6\tQuit\n"
  //This is the menu text, in one big, messy define.

  char menu_item = 0x00; //Initialize menu_item to 0.

  Serial.println(menu); //display the menu
  while (!Serial.available()) { //wait for serial input
    //do nothing
  }
  menu_item = Serial.read();
  //if we're here, we got serial input
  //We act on the input in the switch() below.
```

```
switch (menu_item) {
  case 0x31 : {
      //0x31 is "1". Reset the drive, wait until ready.
      Serial.println("Resetting drive.");
      reset_drive();
      wait_for_drive_ready(1024);
      Serial.println("Drive is reset.");
      break; //Don't try the other cases.
    }

  case 0x32 : { //0x32 is "2" Sleep the drive.
      Serial.print("Sleeping Drive. ");
      Serial.println("Reset to wake up.");
      Serial.println("(Other functions will hang.)");
      sleep_drive(); //sleep the drive.
      break;//Don't try the other cases.
    }

  case 0x33 : { //0x33 is "3". Identify Drive
      Serial.println("Identifying Drive");
      identify_drive(); //put ID info in the block buffer
      dump_block_buffer(); //dump the block buffer
      break;//Don't try the other cases.
    }

  case 0x34 : {
      // 0x34 is "4". Iterate through the sectors
      // until we find a non-empty one.
      Serial.println("Seeking Non-empty Block/Sector.");
      uint32_t sector = 0;
      byte keep_going = 0x79; // lower case y

      while (keep_going == 0x79) {
        // Keep going until the user stops typing "y".

        read_write_drive_LBA_block(sector, false);
        //Read the LBA Block into the block buffer.

        if (!(sector % 100)) {
           Serial.println("Checking Block " +
                          (String)sector);
        }
        //Every 100 blocks give some feedback
        //so the user knows the sketch is still running.
```

```
          if (non_zero_read) {
            //Is the block empty? No? print a message and
            // dump the block buffer.

            Serial.print("Found something. Dumping sector ");
            Serial.print(sector);
            Serial.print("\n");
            dump_block_buffer();

            Serial.println("Shall I continue? (y/n)");
            //Does the user want to go on?

            while (!Serial.available()) {
              //do nothing until we get serial input.
            }

            keep_going = Serial.read();
            //got serial, store it in continue.
            //The loop will evaluate it.
          }
          sector++;
        }
        break; //Don't try the other cases.
      }

    case 0x35 : {
        //0x35 is "5". Ask for a block number,
        //write to it, and dump the block back.

        String input_string = "";
        //This case asks for more than single chars.
        //Store here.

        uint32_t block = 0;
        // Block is the block number we will write to.

        Serial.println("What LBA block shall I write to? ");
        Serial.print("The drive has ");
        Serial.print (LBA_sectors);
        Serial.print(" LBA blocks, and 28 bit LBA addresses go to");
        Serial.println(" 268435456.");
        //28 bit LBA can reach 128GiB.
        // To go further means not using ATA-1.
```

```
while (!Serial.available()) {
  //do nothing while we wait for input.
}
input_string = Serial.readString();
////Store the serial input in input_string

block = string2uint32_t(input_string);
//turn input string into a uint32_t.

input_string = "";//clear input string.

Serial.println("Writing to block " + (String)block +
               ". Please type your message and" +
               " press return.");
while (!Serial.available()) {
  //do nothing
}
input_string = Serial.readString();
//Get another input string, this time the
//pithy text message to be written to the
//block the user selected. No post-processing
//of the string needed this time.

Serial.println("Writing \"" + input_string +
               "\" to Block:" + (String)block);
//Tell the user what we're writing. The \"
//marks are escaped quotes so we can print them.

for (int c = 0; c <= 512; c++) {
  if (c <= input_string.length()) {
    block_buffer[c] = input_string.charAt(c);
  } else block_buffer[c] = 0;
}
//Copy input_string into the block buffer.
//Any bytes not used (because the string
// is shorter) fill with 0s.
//reset_drive();
//The drive can get in the wrong mode to write.
//resetting beforehand makes sure it's ready for
//writing.

read_write_drive_LBA_block(block, true);
//write the contents of the block buffer
//to the drive's data buffer which writes to
//the media itself.
```

```
        wait_for_drive_ready(1024);
        Serial.println("Done writing. Reading block "
                       + (String)block);
        //wait up to 1024ms for the drive to finish writing.
        //We're only writing one block...

        read_write_drive_LBA_block(block, false);
        dump_block_buffer();
        //Read the block back into the block buffer and
        // dump the block buffer so the user can see.

        input_string = "";
        //clean up input_string.

        break;//Don't try the other cases.
      }

    case 0x36 : {
        Serial.println("Sleeping Drive and Exiting.");
        sleep_drive(); //put the drive in its sleep state
        Serial.println("Halted. Reset Cestino to Restart.");
        while (true) { //Do nothing forever.
        }
        break;
        //Don't try the other cases. Not that we'll ever
        //reach this code.
      }
  }
  menu_item = 0x00; //clear menu_item for the next go-round.
}
```

Output

This chapter wouldn't be complete without the output. Some sector dumps have been removed for brevity.

We find a drive when the sketch starts. That's good.

```
Checking for a drive...Found a drive.
Model: ST320014A
Serial Number: 5JZGC69W
Firmware Version: 3.07
Cylinders: 16383
Heads: 16
Sectors: 63
LBA Sectors: 39102336
```

Here's the menu.

```
**** MENU ****
1       Reset Drive
2       Sleep Drive
3       Identify Drive
4       Seek Non-Empty Block/Sector
5       Write to Block/Sector
6       Quit
```

Selected 1.

```
Resetting drive.
Drive is reset.

**** MENU ****
1       Reset Drive
2       Sleep Drive
3       Identify Drive
4       Seek Non-Empty Block/Sector
5       Write to Block/Sector
6       Quit
```

Selected 2.

```
Sleeping Drive. Reset to wake up.
(Other functions will hang.
```

We chose function 1 again.

```
Resetting drive.
Drive is reset.

**** MENU ****
1       Reset Drive
2       Sleep Drive
3       Identify Drive
4       Seek Non-Empty Block/Sector
5       Write to Block/Sector
6       Quit
```

We chose option 3. It dumps the block buffer once it's loaded.

```
Identifying Drive
Dumping Block Buffer
0c 5a 3f ff c8 37 00 10 00 00 00 00 00 3f 00 00 | .Z?..7.......?..
00 00 00 00 35 4a 5a 47 43 36 39 57 20 20 20 20 | ....5JZGC69W
20 20 20 20 20 20 20 20 00 00 10 00 00 04 33 2e |     ......3.
30 37 20 20 20 20 53 54 33 32 30 30 31 34 41 20 | 07  ST320014A
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 |
20 20 20 20 20 20 20 20 20 20 20 20 20 20 80 10 |       ..
00 00 2f 00 00 00 02 00 02 00 00 07 3f ff 00 10 | ../.........?...
00 3f fc 10 00 fb 00 10 a7 80 02 54 00 00 04 07 | .?.........T....
00 03 00 78 00 78 00 f0 00 78 00 00 00 00 00 00 | ...x.x...x......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 7e 00 00 34 6b 5b 01 40 03 34 69 1a 01 40 03 | .~..4k[.@.4i..@.
00 3f 00 00 00 00 00 00 ff fe 60 0b 80 80 00 00 | .?........'.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 01 a8 28 02 54 a7 80 02 54 2e 30 00 02 0c b7 | ...(.T...T.0....
02 10 00 00 3c 03 3c b4 ff ff 00 0d 00 00 08 01 | ....<.<.........
04 80 02 a0 01 02 00 00 00 3c 04 38 e8 08 bd 10 | .........<.8....
00 00 04 54 00 28 00 00 00 00 00 00 00 e0 00 0a | ...T.(..........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 df a5 | ................
Bytes dumped:512

**** MENU ****
1       Reset Drive
2       Sleep Drive
3       Identify Drive
4       Seek Non-Empty Block/Sector
5       Write to Block/Sector
6       Quit
```

We chose option 4. Most drives have something in sector 0. In PCs, this is where the boot sectors are usually located.

```
Seeking Non-empty Block/Sector.
Checking Block 0
Found something. Dumping sector 0
Dumping Block Buffer
fa b8 00 10 8e d0 bc 00 b0 b8 00 00 8e d8 8e c0 | ................
fb be 00 7c bf 00 06 b9 00 02 f3 a4 ea 21 06 00 | ...|.........!..
00 be be 07 38 04 75 0b 83 c6 10 81 fe fe 07 75 | ....8.u........u
f3 eb 16 b4 02 b0 01 bb 00 7c b2 80 8a 74 01 8b | .........|...t..
4c 02 cd 13 ea 00 7c 00 00 eb fe 00 00 00 00 00 | L.....|.........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 e0 2d 00 00 00 00 00 20 | .........-.....
21 00 0b fe ff ff 00 08 00 00 00 98 54 02 00 00 | !...........T...
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa | ..............U.
Bytes dumped:512
Shall I continue? (y/n)
```

We decided to continue...

```
Checking Block 100
```

Many more blocks checked

```
Found something. Dumping sector 21248
Dumping Block Buffer
54 68 69 73 20 69 73 20 61 6e 20 6f 6c 64 2c 20 | This is an old,
6f 6c 64 20 64 72 69 76 65 2e 20 49 20 77 6f 75 | old drive. I wou
6c 64 6e 27 74 20 70 75 74 20 64 61 74 61 20 49 | ldn't put data I
20 76 61 6c 75 65 64 20 6f 6e 20 69 74 2e 20 41 | valued on it. A
6c 73 6f 2c 20 69 74 27 73 20 22 6f 6e 6c 79 22 | lso, it's "only"
20 32 30 67 62 2e 0a 00 00 00 00 00 00 00 00 00 | 20gb..........
```

The block was empty below this file.

```
**** MENU ****
1       Reset Drive
2       Sleep Drive
3       Identify Drive
4       Seek Non-Empty Block/Sector
5       Write to Block/Sector
6       Quit
```

We chose option 5.

```
What LBA block shall I write to?
The drive has 39102336 LBA blocks, and 28 bit LBA addresses go to 369098751.
Writing to block 1. Please type your message and press return.
Writing "Writing in block one, just to see if it can be done." to Block:1
Done writing. Reading block 1
Dumping Block Buffer
57 72 69 74 69 6e 67 20 69 6e 20 62 6c 6f 63 6b | Writing in block
20 6f 6e 65 2c 20 6a 75 73 74 20 74 6f 20 73 65 | one, just to se
65 20 69 66 20 69 74 20 63 61 6e 20 62 65 20 64 | e if it can be d
6f 6e 65 2e 00 00 00 00 00 00 00 00 00 00 00 00 | one............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ...............
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
Bytes dumped:512
```

Yup, the block was written to, and filled out with zeros.

```
**** MENU ****
1       Reset Drive
2       Sleep Drive
3       Identify Drive
4       Seek Non-Empty Block/Sector
5       Write to Block/Sector
6       Quit
```

Finally, option 6.

```
Sleeping Drive and Exiting.
Halted. Reset Cestino to Restart.
```

Credit Where Credit Is Due

I am by no means the first person to connect a PATA drive to a microcontroller, and I've used a large number of sources to piece together what I needed to do it.

The PJRC website, http://www.pjrc.com/tech/8051/ide/index.html#lba, goes into great detail about how to connect one to an 8051 microcontroller with an 8255 parallel driver chip doing the duty we're using a pair of ATmega ports for. It was this site that convinced me to stick with LBA rather than trying to suss out cylinder/head/sector mode, which is different for each drive.

PJRC credits Wesley's Pic Pages, which it now also hosts: `http://www.pjrc.com/tech/8051/ide/wesley.html`. These pages also talk about interfacing a PATA drive (IDE) to a microcontroller using an 8255 to handle the 16 bit data bus from an 8 bit microcontroller. More important, they include a lot of documentation about the PATA standard, from a number of sources including a reverse engineering project.

As I mentioned in the sketch, I cribbed the status masking system in its entirety from GeneT's idefat library, available here: http://code.google.com/p/idefat-arduino/. You might want to take a copy in case Google code ever finishes going away.

Finally, I must credit ANSI, particularly Tim Dovan, the Director of Customer Service, for graciously letting me use their copyrighted information from this long-withdrawn standard. If you need a copy of the modern standard, it's reasonably priced and available here: http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+INCITS+397-2005+Package.

Further—Bigger, Faster, More Modern?

Where could you go with this project? Lots of places. The nearest goals are the most obvious: LBA48 to let you talk to larger drives. You could add filesystems as GeneT did in the idefat library.

SATA requires very high speeds and precise timing, and is probably out of reach of Arduino type projects permanently. I may be wrong. If you find an approach that works, I'd love to know.

Fortunately for Cestino projects, we don't really need hard drives at all. Any reasonable amount of data we might want to store with an 8 bit microcontroller can be stored on an SD/SDHC card using the hardware SPI system built into the ATmega1284P. The Arduino foundation appears to maintain a library for doing just this here: https://www.arduino.cc/en/Reference/SD. If you use this library, your pinouts won't be the same. The ATmega1284P's SPI interface is in PORT B, on pins PB4 through PB7. SPI is not the fastest mode for an SD/SDHC card to run in. Be aware that SD cards want 3.3 volts rather than 5. You'd need to do some level shifting or modify the Cestino to work at 3.3v (Change some of the fuse settings for power management and replace the full can oscillator with something 3.3v-compatible.)

CHAPTER 10

Time Out For a Quick Game

Once upon a time, when you said the words "role playing game" you did not mean a game played on a computer. They were played with pencil and paper and a series of distinctive dice, based on the Platonic solids: a 4-sided die on a tetrahedron, 6-sided on a cube, 8-sided on an octahedron, 12-sided on a dodecahedron, and 20-sided on an icosahedron. As gaming moved on, it was common to add a pair of 10-sided dice, on petagonal bipyramids, which aren't platonic solids, but do, conveniently, have 10 sides. These were used singly to roll values between 1 and 10, and in pairs to roll percentages, or percentile dice.

In those primitive times, we kept up with the gaming community by subscribing to a magazine. It was in the back of this magazine that we saw another device we all coveted but few of us bought: an electronic die roller.

Recently, while trying to dredge up images of this die roller, I stumbled across the expired patent for the device (it's in the Credit Where Credit Is Due section). In addition to the shape of the device and the circuit (which contains no microcontroller of any kind) it contained an in-depth analysis of how the device generated randominity. This project, shown in Figure 10-1, is a reimagining of that device, using the same method of random generation, on the Cestino.

© James R. Strickland 2016

J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_10

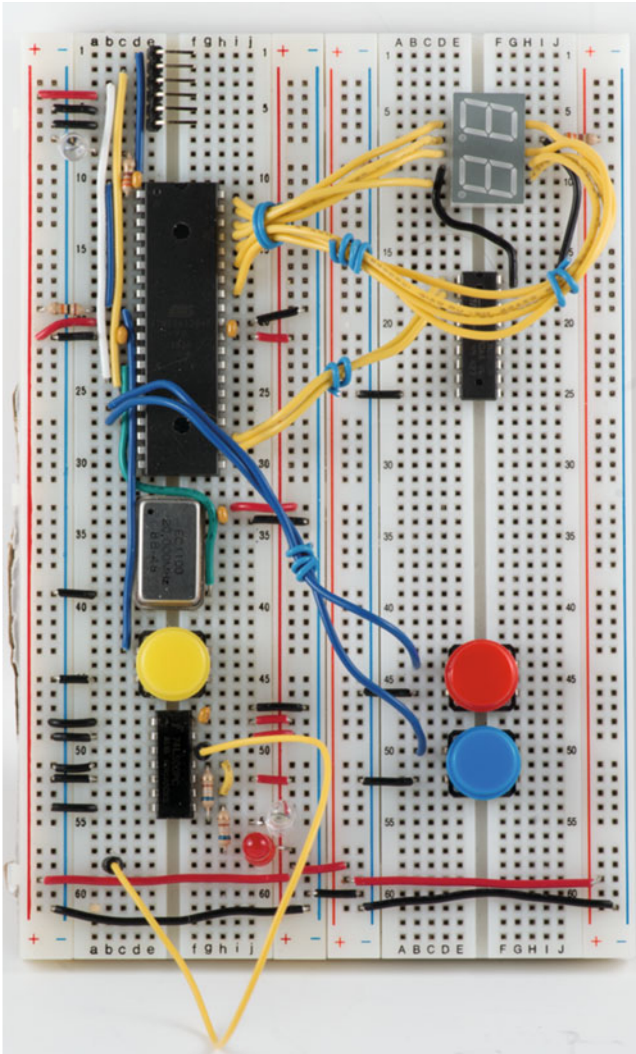

Figure 10-1. *Cestino Dice Device*

Because this project also deals, for the first time, in time-critical functionality, it's also a good vehicle to introduce interrupts and timers. Although we've largely ignored them until now, they are the bread and butter of ATmega and similar microcontrollers.

The Stuff You Need

Although this project has more parts than some of the previous ones, they're smaller, simpler, and among the most common parts out there.

New Parts

- 2 330Ω resistors. 1/4 watt will work, and they're what I used, but they get a warm to the touch. I suggest 1/2 watt resistors for this project if you have them.
- (Optional) 7 150Ω resistors, also ¼ watt. If your display gets dimmer or has dim segments and is hard to read, putting these in as shown in the schematic will fix it.

Used Parts

- 1 ULN2803 or similar NPN Darlington transistor array. These are a pretty standard item in the Arduino world. If you don't have one handy, a pair of TIP120 Darlington transistors will work without changing the rest of the circuit. If you haven't got those either, any NPN transistor capable of switching 80mA or more and appropriate base and emitter resistors will work, but you're on your own figuring out the resistor values and wiring. PNP devices will also work, and would actually be a better choice for common anode LED displays, but again, this will change the schematic.
- 1 LTD 4608JG two-digit, right-hand decimal, seven-segment LED display. This is what I used, and since LED displays' pinouts vary by manufacturer, if you want to use the sketch as written, this is the one you need. That said, there's nothing at all special about this display. Just about any dual-digit, right-hand decimal seven-segment LED display will work, but you will have to change the wiring of the display to suit the display you have. You can also use two single seven-segment, common anode, right-hand decimal LED displays. Just wire the segment and decimal lines together and again, your setup will differ from the schematic. You'll need the datasheet for whatever display you use.
- Two Tactile Switches, (buttons) momentary on. I used the same kind I used for the Cestino's reset button. Ideally, these will have different colored buttons or some other way to tell them apart, because they have different functions.

Electronic Dice

Computers in general are very bad at creating randominity when they're working correctly. The entire point of logic circuitry is repeatability: a given set of inputs result in a given output. Randominity flies in the face of this. There are many random generation algorithms. It's a heavily researched field, since encrypted data should be indistinguishable from random values. Indeed, a friend of mine who wrote software for slot machines once told me it takes a great deal of study and quite a lot of computing power to generate sufficiently random values for gambling.

So how did the $20 (about $60 or $80 in modern dollars) electronic dice device I saw in that magazine back in the bronze age of computers accomplish it? Here's how the circuit worked.

We talked about counters in Chapter 6. If you tied the 74LS92 to the 20MHz oscillator on the Cestino, you undoubtedly observed that the 74LS92 can count very, very fast. You'll recall from Chapter 7, the logic probe, that we humans can't even perceive LEDs changing from bright to dark or back in much less than 20ms. The dice device used both these factors to its advantage.

It was a simple beast. It contained a pair of TTL decade counters, one for ones and one for tens, each of which had outputs for zero through nine. Because a die can't roll zero, the ones counter was wired to show one higher than its actual count: its zero output is wired to LED 1, and so on.

When the ones counter's output nine switched on, instead of lighting an LED, it incremented the tens counter. The tens counter, by contrast, was wired to count from zero to nine. Left to run freely, the counters would count from zero to ninety-nine, lighting up one or two LEDs to indicate values from 1 to 00. (00 had a separate LED, and represented a 10s digit of 0.)

A six-position switch and two AND gates were used to select the die. For four-, six-, and eight-sided dice, the switch simply tied output four, six, and eight (which represented five, seven, and nine, as the ones counter is skewed by one) to the reset circuit for both counters. For 12-and 20-sided dice, it connected to the output of an AND gate. The 12-sided die's AND gate tied output 2 (representing 3) of the ones counter and output one (representing 10) of the tens counter together to represent 13. When the counters reached 13, the AND gate would reset them long before the LED could light. The 20-sided die worked the same way. Output zero of the ones digit (representing 1) and output 2 of the tens digit (representing 20) were wired to the other AND gate, and when the selector switch was set to its output, and those two outputs went high, the AND gate would reset the counters before the LED could light.

For percentiles, the reset circuit was grounded and both counters were permitted to run freely.

To actually roll the dice, you held a button down. It connected an oscillator to the counters, and this is where the randominity was generated. We don't think of our actions as random, but measured on the scale of a few hundred kHz, when our thumb actually comes off a button, and when the button's contacts stop bouncing (more on that later) is actually quite solidly random. So once you selected the die you wanted, you held the button down. All the LEDs up to your roll appeared to light simultaneously, even though no more than two were ever actually on, and when you let up, they stopped at your roll apparently instantly.

It's an elegant, all TTL design. It uses two decade counters, a dual AND gate for die selection, and a quad NAND gate, used with a resistor and a capacitor as an oscillator. The rest are switches, LEDs, resistors, and a battery. Because they used 4000 series CMOS TTL, the device could tolerate a wide range of battery voltages as the battery ran down. Because it was simple, it would have been fairly cheap to manufacture.

Apparently they didn't take the gaming world by storm. The patent, US4431189, was filed in 1981, and by 1987 it had expired for failure to pay maintenance fees. In 30-plus years I've been gaming, I've seen exactly one in use.

We're going to have the Cestino generate random numbers the same way, by measuring the button-down time some hundreds of thousands to millions of times a second. We're also going to exploit the low speed of human eyes and LEDs the same way as we drive our seven segment displays.

Driving 7 Segment Displays

So what exactly is a seven-segment display with a right-hand decimal and a common anode? It's seven LEDs shaped like segments of a number from zero to 0xf plus one that is on the right and very small, where it serves as a decimal point. Common anode means all the anodes (positive terminals) of the LEDs are tied together. You put voltage on the anode, ground the cathode pins for the segments you want on (as always, with dropping resistors, but we'll get to that) and you have a display. Put two of them together and tie the cathodes for each pair of segments in the same position, and you have a two-digit display, which is what the LTD4608JG is. You chose the digit you want by choosing an anode to apply power to, and you chose what segments to light by choosing which cathodes to ground. Note that the digits are numbered left to right, so the leftmost digit is digit one, and the rightmost digit is digit two.

The cannonical "correct" way to wire the seven-segment display is with one dropping resistor per segment. But this eats up a lot of breadboard real estate and takes a lot of extra wiring. Plus, two of our 330Ω resistors are permanently tied up in the logic probe circuit. If you've looked closely at the photo of the Cestino Dice Device (Figure 10-1), you'll see I took another tack.

Recall that Kirchhoff's first law is that the sum of the currents flowing into a node of a circuit equal to the sum of the currents flowing out. This means that whichever side of the LED we put the dropping resistor on, it will impose the same current limit and cause the same voltage drop. We can, as easily, put our dropping resistors on the anodes. Because all of our segments in each display digit share a common anode, this cuts our resistors to two. If we connected one of these resistors from + to - it would drop 5 volts and dissipate a bit over 15mA, and a wattage of about 75mW. Our resistors are rated at 250mW (0.25 Watt), so we're fine there. Does adding the LEDs in parallel decrease their resistance? Yes, but it's unimportant. The forward drop of each LED remains the same, about 1.5 volts, which means our dropping resistor limits the current to about 10mA. (The datasheet doesn't specify the forward drop. It's a lousy datasheet.)

It's tempting to calculate each segment's load and add them together, and then worry about the wattage of the resistor, but recall Kirhhoff's first law says it isn't so. The current is limited by the resistor and all the circuits tied to it share that current. This means that the LEDs get slightly dimmer, the more of them that are switched on, as they divide the 10mA available among all eight (worst case) LEDs.

This isn't standard industry practice, and there's a reason. Each LED in a seven-segment display has slightly different characteristics. Some may draw more current than others, or they may have slightly different voltage drops, or different internal resistance. As you add these variations together, you can get a display that is hard to read, or has some segments brighter than others. My LED display worked fine, but if you're having these problems, add a 150Ω resistor between each cathode and its respective PORT A pin.

So we know we can switch the current we need to light the seven segment LEDs, but what if we want more digits? What if we wanted to make a calculator? Can we really only drive four seven-segment displays with one ATmega1284P? Do we really have to use two full ports just for these two digits? Could we really only drive four seven-segment displays with an ATmega1284P, and that only if we didn't want it to do anything else?

The answer is no. We can drive an arbitrary number of digits with one port, plus one pin per digit. The technique is called time division multiplexing.

Time Division Multiplexing

From our experiments with the logic probe, we know that between the speed of an LED and the speed of human vision, we can't perceive off pulses on an LED of less than about 20ms, or a frequency of about 50Hz. We'll use 60Hz from here on. I am an American, and our wall power is 60Hz. If the LEDs cycle much slower than the overhead lights, it can make them appear to flicker at the difference between the two frequencies.

For those outside the Americas, the Caribbean, and parts (but not all) of Korea and Japan, your power is probably 50Hz, which fits the metric way of thinking better. You might need to slow the display timing in the sketch to suit your lights if it appears to flicker. You could also increase the display speed until the flicker goes away.

As long as we power each LED at least every 16.6ms, we won't see the difference. We know from those same experiments that 60Hz is glacial compared to the speed of transistor logic. That's how the digits in this project are driven. If the port can react at least once every 8.3 ms per digit (hint: it can) we can drive two digits with different values and have them appear to be lit at the same time. Three digits would need updating every 5.53ms, and so on.

As with all things, there's a cost. We have to control, with great precision, which display is active at any given time. Their common anodes give us an easy mechanism to do that. To talk to digit 1, on the left, we connect the common anode of that digit to the + bus (by way of the dropping resistor). To talk to digit 2, on the right, we disconnect digit 1 from the + bus and connect digit 2 to it. We can switch whichever cathode lines we want. Without power to its anode, digit 1 won't do anything. Again, as long as we get back to powering digit 1 in less than 16.6ms, we won't see the difference.

We could reduce the duty cycle further, powering each digit for less than half the time, either by adding more digits to the multiplexing system, or by making the duty cycle asymmetric—off more than it's on—and if we needed to conserve power, the extra complexity might be worth it. For this project, we'll stick with a 50 percent duty cycle, and

two digits. With the circuit we're going to build, we could run up to eight, on two ports. With a better design (an oscillator, a decade counter, and some buffers, perhaps) we could do many more with only one port. If we had an ATtiny, with only three or four GPIO pins available, we could multiplex the individual segments as well.

Interrupts

The big catch of multiplexing (muxing, for short) is that it's time sensitive. If our loop function gets busy doing something else, and the display switching doesn't happen on time, it will be visible. Worse, if the display switching and the port switching get out of sync, the digits could be transposed or superimposed over each other. We need a way to let the ATmega do the job we want of it, but make sure that every so often, it stops and updates the display. Fortunately, like nearly all microprocessors and microcontrollers ever made, the ATmega1284P has circuitry that, when activated, stops whatever it's doing, and jumps to another part of the code. When that part is done, it jumps back and picks up where it left off. It's called an interrupt.

The piece of code (a function, since we're using C/C++) called when the interrupt is activated is called the Interrupt Service Routine, or ISR.

If we were looking at assembly language, the address in program memory that the interrupt causes a jump to is called the interrupt vector. These are handled for us by the Arduino core.

The ATmega1284P has two types of interrupts: internal and external We'll deal with the external type first.

If you look at the pinout diagram in the ATmega1284P datasheet (I left them out of the one in Chapter 2) you'll see that all the pins of all the ports have a PCINT number. These are not the interrupts we're looking for. These are the PCINT system.

The PCINT interrupt system is designed to monitor whole ports at a time for any change to the pins, whether or not the pins are set up as outputs. Each port's PCINT system can call one fixed interrupt service routine (ISR)—the code that the interrupt sends the processor to. This is handy for things like the ATA Explorer (although we didn't use it) so that when the drive returns a byte of data, our code can go deal with it, regardless of what else it was doing at the time.

Although there are several libraries for Arduino that allow PCINT to be used in sketches, we won't be using the PCINT system in this project. We need finer control over what events cause an interrupt.

Instead, we will use the external interrupt system, called INTs. There are three INT pins: INT0, INT1, and INT2 (ATmega328 based Arduinos have only the first two). These are found on pins INT0/PD2 (Pin 16), INT1/PD3 (Pin 17) and INT2/PB2 (Pin 3) of the ATmega1284P. These lines are, in other microprocessors and microcontrollers, called IRQ lines—Interrupt ReQuest lines. File that away. You'll see IRQs again in chapter 11.

The ATmega1284P's three interrupt lines can be configured to activate when the pin is low—below 0.3 volts, when it is high—above 0.6 volts, when it rises from below 0.3 volts to above 0.6 volts, when it falls from above 0.6 volts to below 0.3 volts, or on any change at all, like the PCINT system. Each interrupt pin has its own interrupt vector, so it can call a specific interrupt service routine. The INT system allows us to know which pin /exactly/ changed and *what* exact change occurred before we even get to the interrupt service routine (ISR). This is its big advantage over the PCINT system.

Configuring Interrupts

The Arduino core makes configuring external interrupts very, very easy with the attachinterrupt() function. This function returns no data, and takes three parameters: the interrupt number, the Interrupt Service Routine's name, and the interrupt mode: LOW, CHANGE, RISING or FALLING. There is a HIGH mode, but it is only implemented on certain types of Arduino that don't use an ATmega microcontroller.

If you read about Arduino interrupts, you'll see they recommend using a function called digitalPinToInterrupt() to look up the interrupt number for a given pin. As of this writing, this function doesn't work on the Cestino, and calls to it won't compile.

Here's an example of attaching an interrupt, snipped from the sketch we'll be writing later in this chapter:

```
attachInterrupt(1, read_selector_isr, FALLING );
```

This command connects interrupt 1 to the function read_selector_isr, and configures the interrupt to occur when the voltage on that pin (PD3, aka Pin 17, if you wondered) falls from above 0.6 volts to below 0.3 volts.

Note especially that while read_selector_isr() is the correct name of the c function, it isn't called with any parameter parenthesis. ISRs are not allowed to take any parameters, nor may they return any data, and they are attached without their (empty) parameter field.

Interrupt Service Routines (ISRs)

Interrupt Service Routines are, in C and C++, functions that take no parameters and return no data. They must, above all, be brief. The longer they run, the more likely additional interrupts will be missed (global interrupts are disabled while an ISR is running) and while the ISR is running, the ATmega is not doing whatever job was interrupted. Short, fast ISRs are important. Incrementing a variable is a good job for an ISR. Triggering a hardware event, such as turning a pin on or off is a good job for an ISR. Reading and storing an incoming piece of data is the most common use.

Global variables that are changed by an interrupt service routine must be handled differently than normal. A normal variable exists in memory. A copy of it may also exist in the registers of the microprocessor. As long as the microprocessor is in the same section of code that the variable is in, the variable in memory can be updated when that section of code exits, or at some other interval determined by the compiler and/or the Arduino core. When the ATmega jumps to an interrupt vector and begins running a completely different piece of code, the registers and the variable in memory can get out of sync.

To avoid this problem, simply add a qualifier to the variable definition: volatile, as in `volatile byte myvariablename`. This tells the compiler to generate code such that the variable is /always/ accessed from memory. While somewhat slower than using registers

in the ATmega, this ensures that the copy in memory is up to date with any changes made to it by the ISR. Local variables in the ISR need not be declared volatile, as their context remains the same for their entire (brief) existence.

Interrupt Vectors

For INTs (external interrupts), the interrupt vectors are handled automatically by attachInterrupt, and (presumably) a pointer to our ISR is placed in the interrupt vector for us. This is incredibly handy. Sadly, the Arduino Foundation did not see fit to extend this to internal interrupts, like those used by timers, for reasons known only to themselves. Here's how interrupt vectors really work, so they're not confusing later.

When an interrupt happens, the ATmega stops executing whatever code it's executing, does some quick housekeeping so it can remember where it came from, then jumps to a specific pair of locations in memory. Those locations had better contain another jump instruction, and a location in memory that contains the code that will handle the interrupt.

Because internal interrupts (like those generated by the timer) are not handled by the attachInterrupt() function, the Arduino core sets them up to call specific function names, even though at the assembly language level, underneath the C we look at, internal and external interrupts are handled more or less the same.

The AVR/GCC compiler environment has #defined macros to handle *all* the interrupt vectors, for example: `TIMER3_COMPA_vect`, so that by using this name for the actual interrupt vector (which is 0x0040, in case you wondered) as a parameter to the ISR() macro, the vector would be set up correctly, and the compiler signals (which we haven't talked about) would be handled correctly.

This is still how timer/counter interrupts are still handled. You'll see this in the sketch where the selector interrupt service routine is set up with attachInterrupt and has a normal name, but the timer interrupt vector is set up and connected to its ISR with `ISR(TIMER3_COMPA_vect)`, which will make a lot more sense after you read the section on timer/counters.

If you wanted to, you could handle external interrupts (INTs) the same way we do internal interrupts, using `ISR(EXT_INT1_vect)`, changing the numeric part for whichever INT you actually wanted. You'd then have to set the interrupt mask to enable that external interrupt yourself too. You'll sometimes see it done this way this in older Arduino sketches.

Timer/Counters

If you think back to Chapter 6, where we talked about the 74xx92 counter, timers will make a whole lot more sense. Imagine you connected that counter, or one of the decade counters I mentioned earlier to the 20MHz TTL oscillator that drives your Cestino. You know that the counter generates some combination of signals every time its input goes positive. You know that it will go positive exactly 20,000,000 times a second. If you know the oscillator's speed and you count the pulses, you have a timer.

To set the timer, you have to have a place to store a bit pattern that you can check against the output of the counter, and some logic that does something if they match, or don't match.

With an extra place to store some control settings and some logic, you could use the same electronics as a counter for other events, too. Two functions out of one circuit. Awesome.

This is exactly how timers work in the ATmega1284P. If you guessed that the control settings and the bit pattern to match are stored in registers, you catch on fast. The Arduino world goes to great machinations to hide the timer registers and make them "easy to use" by creating a #define for each bit of each mask, and suggesting you logically combine them, but we're used to looking at eight and sixteen bit registers. We've done it for the last two chapters. Here's how the ATmega1284P really handles timers.

ATMega Timer/Counter Counting

There are three timer/counters in the ATmega1284P. (ATmega328 based Arduinos have only 2.) They come in two flavors, one 8 bit counter/timer, and two 16 bit counter/timers, which means exactly what it sounds like. The 8 bit timer/counter can count to 255 (0xFF) before resetting, and the 16 bit timers can go to 65535 (0xFFFF) before resetting.

Each timer/counter has two possible clock (the signal that triggers the timer) sources. One is the external clock source, which is an edge detector, that picks up the rising or falling edge of an external pulse, just like the INTs do. We won't be using that one here. The other source is internal, called the timer/counter prescaler. This is a 10 bit counter used as a divider, whose input is the system clock.

What does this mean? We can select the clock for our timer/counter to not be divided, and run at the full 20MHz of the system clock, or we can choose to have the clock divided by 8, 16, 32, 64, 128, 256, or 1024. This is set in the Timer Counter Control Register (TCCR), along with a lot of other parameters I'll get to. The value the timer has counted to is always stored in the Timer CouNTer register (TCNT) for that timer.

ATMega Timer/Counter Actions

Timer/counters would be pretty useful if they did just that, and we had to read the Timer Counter register and process the results in software, but like the commercials on late night TV always say "wait, there's more."

Timer/counters can make the comparison for us at hardware logic speeds, and they can trigger events in hardware. There are more registers for this, in case you hadn't already guessed.

Timer/counters on the ATmega have many possible functions. They can trigger an event when the TCNT register reaches its maximum value. They can trigger an event when the TCNT register reaches its minimum value. They can trigger when a specific value stored in the Output Compare Register is reached. They can also do combinations of these functions.

Timer/counters can switch an output pin on and off, very quickly, without program intervention. This, plus some configuration, allows them to generate clock signals (we'll talk about this more in Chapter 11) and Pulse Width Modulation (PWM) signals. In PWM, the duty cycle of a signal controls /how much/ current and voltage is available to down-stream electronics. PWM is essentially an analog output, with a couple of passive components. You can also use it to control servo motors.

Most important for our uses in this sketch, however, timer/counters can cause an interrupt in the ATmega, and the resulting jump to an interrupt handler.

What exactly the timer does and when, is configured in the TCCR register, and is broken up by modes. I'll cover all the registers and modes next.

Configuring Timer/Counters

We'll use the 16 bit timer 3 as our example. Eight bit timers will have smaller TCNT and Output Compare Registers (OCRs). For different timer/counters, you'd use a different number, like TCCR1A or TCCR2A, and so on. The numeric value determins which timer you're using.

TCCR3A—TCCR3C

There are really three 8 bit Timer Counter Control Registers per timer/counter, for a total of 24 bits. A lot of this space isn't used, but the engineers at Atmel wisely decided to leave some extra space for future capabilities.

TCCR3A contains these bits in the following order: COM3A1, COM3A0, COM3B1, COM3B0, unused, unused, WGM31, and WGM30.

TCCR3B contains these bits: ICNC3, ICES3, unused, WGM33, WGM32, CS32, CS31, and CS30.

TCCR3C contains these bits: FOC3A, FOC3B, and the rest are unused.

Clock Select Bits (CS30-CS32)

The Clock Select bits, CS30, CS31, and CS32 for Timer/Counter 3, allow us to select what triggers our counter to count. These three bits are the lowest three of TCCR3B.

0b000 for these three bits is no clock source. The counter is stopped.

0b001 uses the internal clock divided by 1. That is, with no prescaling. This means our counter will get to drink from the 20MHz firehose of the Cestino's TTL oscillator.

0b010 will divide the internal clock by 8. Our counter will be triggered (clocked) every eight pulses of the 20MHz oscillator.

0b011 will divide the internal clock by 64.

0b100 will divide the internal clock by 256

0b101 will divide the internal clock by 1024.

0b110 uses an external clock source on the appropriate pin, (note that it's different for different ATmega types) and it will clock on the falling edge, when the signal on that pin falls from above 0.6v to below 0.3v.

0b111 uses an external clock source on the appropriate pin, and it will clock on the rising edge, when the signal on that pin rises from below 0.3v to above 0.6v.

Waveform Generation Modes (WGM30-WGM33)

The four Waveform Generation Mode bits are located in the lowest two bits of TCCR3A and bits three and four of TCCR3B. I'm going to treat them like they're one contiguous bit field, with WGM30 at the low end and WGM33 at the high, but you'll need to break that bit field up and put the bits in the right registers when you use them.

0b0000 is normal mode. The timer counts from zero to 65535 and rolls over.

0b0001 to 0b0011 are phase correct PWM modes, from 8 bit to 10 bit resolution. This means the timer/counter counts more often to ensure that the resulting waveform's phasing is consistent.

0b0100 is Clear Timer on Compare (CTC) mode 4. When the timer hits the value stored in OCR3A, the timer (TCNT3H and TCNT3L) will be zeroed and a timer event will be triggered.

0b0101 to 0b1011 are various other PWM modes, with the focus on correct phasing, correct frequencies, or both. These use the timer/counter in different ways, depending on what parts of the resulting PWM signal are most important: high frequency, the phase-accuracy, frequency accuracy, or phase and frequency accuracy. Generally speaking, the phase accurate and phase and frequency accurate modes are good for motor control (according to the datasheet) and the fast modes are good for power regulation and digital to analog conversion, where higher frequencies mean smaller passive components. We won't be using PWM in this project.

0b1100 Clear Timer on Compare (CTC) mode 12 is the same as CTC mode 4 except that the ICR3 register is used for the comparison. That register can be set by input events. See IRC3H and ICR3L

0b1101 is reserved, so we can't use it.

0b1110 to 0b1111 are more fast PWM modes.

Compare Output Pin Mode Bits (COM3A0-1, COM3B0-1)

The Compare Output Mode bits are the highest four bits of TCCR3A, arranged like this:

COM3A1, COM3A0, COM3B1, COM3B0.

The output compare system of each counter has two channels, either of which can trigger an action. One counter can trigger an action when OCR3A is reached, and a different action when OCR3B is reached. (see OCR3AH and OCR3AL and OCR3BH and OCR3BL, below).

The COM3A and COM3B registers set up whether a pin is toggled by the timer/counter event, and if so, when and how. COM3A and COM3B correspond to pins OC3A and OC3B. These are also known as PB6 and PB7 (Pin 7 and Pin 8). If Timer 3 is configured to trigger a pin, the timer will temporarily override the settings on that port to do what we've told it.

Since the COM3A and COM3B bits do the same things for their respective channels, I'm going to treat them as a pair of bits, with COM3x1 the highest, and COM3x0 the lowest.

0b00 means the port operates normally. No outputs are connected.

0b01 means that whatever state the OC3x pin is in, switch it when the timer/counter reaches the value in OCR3x (where x is A or B).

0b10 means when the timer/counter reaches the value of OCR3x, turn the pin off. If we're counting down, turn it on.

0b11 means when the timer/counter reaches the value of OCR3x, turn the pin on. If we're counting down, turn it off.

We don't actually use the compare output pin modes in this project, although it would have saved some code and processing time if we had. File them away, though. We use compare output pin modes a lot in Chapter 11.

Input Capture Bits (ICNC, ICES)

ICNC3 and ICES3 are bits 7 and 6 of TCCR3B. They control aspects of the input capture system.

ICES3, Input Capture Edge Select determines whether the input capture pin for Timer/Counter 3 (ICP3, aka PB5, Pin 6) clocks the counter on the rising edge or the falling edge. If you want the timer/counter to clock when the pin transitions from below 0.3v to above 0.6v, use the rising edge, and set this bit to 1. If you want it to clock when the pin transitions from above 0.6v to below 0.3v, use the falling edge, and set this bit to 0.

ICNC3, The Input Capture Noise Canceler bit for timer/counter 3 turns the input capture noise canceler on and off. When it's on (1), the input pin is read three more times, and the timer/counter is clocked only if all four readings are the same.

Force Output Compare Bits (FOC3A and FOC3B)

The Force Output Compare bits, FOC3A and FOC3B, are the only interesting bits in TCCR3C, bits 7 and 6, respectively. When written to, these bits force whatever output compare action is configured (changing the state of a pin, usually.) This will not set the OCF3A or B flag (see the TIFR3 register), nor will it reset the counter if we're in CTC mode.

TCNT3H and TCNT3L

The TCNT3H and TCNT3L registers are two parts of a 16 bit register containing the timer/counter's actual value. In Arduino, they are merged into TCNT3 and read or written to as an integer.

A timer/counter need not start from zero or 65535. You can write whatever starting value you want into this register and count up or down from it.

OCR3AH and OCR3AL and OCR3BH and OCR3BL

OCR3AH and OCR3AL and OCR3BH and OCR3BL are the Output Compare Registers for the A and B channels. Each pair (H and L) is a 16 bit register, which you'd expect to match a 16 bit TCNT3 value. Like TCNT3, the high and low (H and L) pairs are merged in Arduino into a single register: OCR3A and OCR3B, and can be read and written as integers.

ICR3H and ICR3L

The ICR3H and ICR3L registers are both halves (high and low) of the 16 bit ICR3 Input Capture Register. When the Input Capture Pin (ICP3 aka PB5, Pin 6) is triggered, according to the rules set in TCCR3B, the value of TCNT3 is copied into ICR3. As with all 16 bit registers, ICR3H and ICR3L are merged in Arduino into the ICR3 register, and may be written to or read from as integers.

TIFR3

The TIFR3 (timer/counter interrupt flag register) is an 8 bit register that contains all the flags for Timer/Counter 3. A flag is simply a boolean value, represented by a single bit, that is set when the bit is set to one, and cleared or not set when the bit is set to zero. These flags can be thought of as the interrupt signals. If the TIMSK3 register has the corresponding bit set, they interrupt the ATmega. If it doesn't, they don't, but you can still read them. Functionally, the timer/counter system ANDs the TIFR3, and TIMSK3 registers and a global interrupt enable register to determine which, if any, interrupts to fire.

The bits in this register go like this: reserved, reserved, ICF3, reserved, reserved, OCF3B, OCF3A, TOV3.

The reserved bits will always be zero. We can ignore them.

ICF3 is set when ICP3 is triggered according to the rules set for it in TCCR3B. This means that if the pin's voltage rises, and the input capture system for Timer/Counter 3 has been set to capture the rising edge of that pin, when TCNT3 is copied to ICR3, the ICF3 flag will also be set.

OCF3B and OCF3A are set when an output compare match occurs on channel B or A (respectively).

TOV3 is the overflow flag. When the timer/counter overflows in either direction—high if it's counting up, low if it's counting down) TOV3 is set. TOV's exact behavior is set by the WGM bits in TCCR3A and TCCR3B.

TIMSK3

TIMSK3 is an 8 bit register containing a mask to determine what interrupts are activated, if any, and when, when a timer match or overflow happens. Functionally, the timer/counter system ANDs the TIFR3, and TIMSK3 registers and a global interrupt enable register to determine which, if any, interrupts to fire.

This is the register we'll use to enable an interrupt to call our interrupt service routine (ISR) to drive the LED displays. The bits in this register go like this: unused, unused, ICIE3, unused, unused, OCIE3B, OCIE3A, TOIE3.

ICIE3 is the input capture interrupt enable for timer/counter 3. If this bit is set to 1, and ICP3 (the input capture pin) goes high, an interrupt will be generated.

OCIE3B and OCIE3A are the output compare match interrupt enables for channels A and B. If either of these is set to 1, when OCRA3 or OCRB3 is matched (respectively) an interrupt will be generated.

TOIE3 is the overflow interrupt enable. When the counter overflows either high or low, depending on which direction it's counting, the TOV3 flag is set, and the counter is reset. If this bit is set to 1, an interrupt will also be generated.

Build the Dice Device

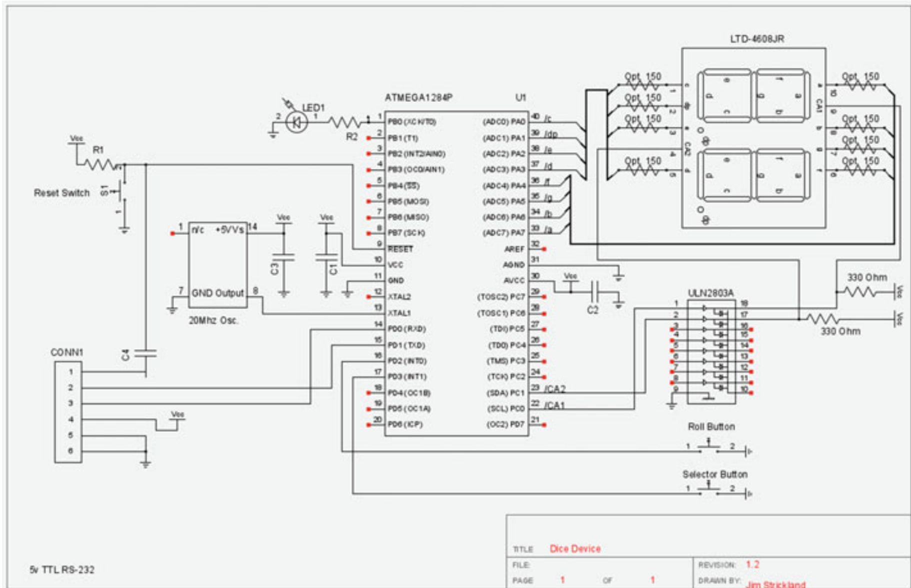

Figure 10-2. *Dice Device Schematic*

For all the complexity of using the hardware timers and interrupts, as you can see from the schematic in Figure 10-2, the actual circuit of the dice device is very simple. (That's the advantage of using the hardware timers and interrupts.) There's also a certain dumbness in this circuit, that I'll explain when we get to it.

There are four components besides the Cestino in this circuit: two buttons, the dual LED display, and a ULN2803. This last is an integrated circuit containing eight Darlington transistor channels, complete with built-in load and base resistors, as well as diode protection. It's a staple of the Arduino world, used to drive loads up to half an amp per channel, for electric motors (where the diode protection is a must), light bulbs, solenoids, and large strings of LEDs. Here's where the dumbness comes in: it's not really the right choice to drive a common anode LED display. The common anodes connect to the + bus of the Cestino. When one of the bases of the ULN2803A is raised high, the corresponding output pin is connected to a common ground. All the emitters of the Darlington transistors are tied to it. In order to use it in this circuit, what I've done is put the 330 Ohm dropping resistors between the anodes of the LED displays and the + bus, and then wired the display side of each resistor to one of the ULN2803 channels. When that channel of

the ULN2803 is activated, the common anode is shorted to ground. This works, but it means that when a display is turned off, more current is consumed and turned into heat by the dropping resistor, than when it's switched on. In a battery driven situation, this would be incredibly wasteful. If you have separate Darlington transistors, such as the TIP120, or any other power transistors rated at 15mA or more plus whatever resistors they need to switch at 5v, you could wire the emitter to the common anode (via the dropping resistor), and the collector to the + bus and only consume current to make light. The ULN2803A is not really the right IC for the job, but when you're making electronics from stuff in your junk box, sometimes you have to compromise. In fairness, we're making no effort whatever to conserve power with the Cestino either.

That said, let's go ahead and build it.

As you saw in Figure 10-1, I put the display at the top of the breadboard, followed by the ULN2803A, followed at some distance by the buttons. Make sure you can tell your buttons apart. The upper one will roll, and the lower one will select what die to roll.

We'll drive the LED display with port A, so wire pins 1, 2, 3, and 5 of the LED display (assuming you're using the LTD4608JG or JR—the red version) to PA0-PA3, and 6, 7, 8, and 10 to PA4-PA7. Yes, seven segment displays have only seven cathodes, but the decimal points have one each, too. If you find your display gets too dim, or the segments are not evenly lit, you can go back and add the resistors marked Opt. 150 (optional 150Ω) between PA0-8 and pins 1,2,3,5,6,7,8, and 10 on the LED display.

Wire a 330Ω resistor to pin 18 of the ULN2803, and from there to the + bus. Do the same with the second 330Ω resistor and pin 17. Then wire the junction between pin 18 and the resistor to pin 9 of the LTD4608JG LED display. This is the common anode line for digit 1, which is the leftmost, or most significant digit. That's how the datasheet for the LTD4608J series reads, and for clarity, I'll call them the same thing.

Connect the junction between the resistor and pin 17 of the ULN2803 to pin 4 of the LTD4608JG. This is the common anode for digit 2, the rightmost, least significant digit.

Now wire pin 1 of the ULN2803, which controls the common anode for digit 1, to PC0, (Pin22) and pin 2, which controls the common anode for digit 2 to PC1, (Pin 23).

The display is now wired.

If you are using a different LED display than the LTD-4608JG or JR, it is vital that you find a datasheet on it and read it. Your pinout will be different. Don't count on the segments being lettered the same either. If your display is a common cathode display (they're not as common) you will need to change the common anode driver circuit (the connections to the ULN2803) accordingly and change part of the sketch to reflect that your segments are active high instead of active low. You can do this. It took me maybe half an hour to convert the sketch and the circuit to this display from the one I used in the prototype. It's just lighting up LEDs. As long as they make light in the right place given the right signal, it'll work fine.

The roll button (the red one in Figure 10-1) is connected to pin 16. You might notice that this is INT0, and you may have heard that using INT0 messes up serial communications between Arduinos and the host computer. All true, but we're not using the interrupt on this pin. This pin will be poled by the loop() function, that is, loop() will read it over and over again (using digitalRead, no less.) Go ahead and wire PD2 (Pin 16) of the ATmega to one side of the roll button. Wire the other side of the roll button to the - bus.

The selector button will use an interrupt, INT1, in this case, which is connected to PD3 (Pin 17), so go ahead and connect one side of your button switch there and the other side to ground. If you're guessing we'll be turning on the pull-up resistors on both these pins, you're right, and normally you'd be jumping ahead, except that we're done with the wiring. Really, that's all there is to it.

The Sketch

The sketch, for all the complexity we've added using interrupts and hardware timer/counters isn't that complex either. Sometimes knowing why we do something is far more complicated than actually doing it. But the why is half of why we're doing it, right?

In any case, we begin with the usual precompiler definitions. First, we define our busses. SEG_BUS drives the displays and MUX_BUS drives the common anodes to allow us to multiplex, mux for short, the displays.

```
#define SEG_BUS PORTA
#define MUX_BUS PORTC
```

We define the port settings for the MUX_BUS next. Remember that the common anodes, because of how we had to wire them to use the ULN2803, are active *low*. So to turn /on/ the MSD, which is digit 1, we need to make sure that PC0 is *low* and PC1 is *HIGH*. We do this by setting the port to 0x2, in hex, which turns on the two bit, and turns all other bits off. LSD_COMMON works the same way, save that it turns on the ones bit and turns off all the others. Active low, as we've seen over and over again, makes everything backward.

```
#define MSD_COMMON 0x2
#define LSD_COMMON 0x1
```

Most of our displayed values will fit the usual hexidecimal digits, but because the seven segment (plus decimal) displays can display some values that aren't in hex, we define those here. We define them in decimal for clarity, but they'd be 0x10 and 0x11 in hex.

```
#define DECIMAL 16
#define BAR 17
```

Next we define which pin is the button pin (the roll button is connected here) and its interrupt (even though we don't use it) and the selector button pin, and its interrupt, which we do use.

```
#define BUTTON_PIN 16
#define BUTTON_PIN_INTERRUPT 0
#define SELECTOR 17
#define SELECTOR_INTERRUPT 1
```

Interrupt service routines have one particular annoying feature: they can neither take data nor return data. Since we're dealing with them in C as functions instead of C++ objects, we have to store their data in global variables. Some of the variables also need to be persistent, rather than recreated every time a function is called, to save compute time. Again, using an object would make this simpler.

The first global is seg_decode_array. It defines an array from zero to 17 that contains all the possible port settings to drive that particular value on one digit of the LED display. Remember that we defined 16 as BAR and 17 as DECIMAL, so they're the last two entries in the array. Because we're initializing the array at the same time we define it, and it's a constant, we don't have to specify an index number. With this specific LED display, and the segment order used in its datasheet, the order of the bits in this pattern are, from MSB to LSB:

A B G F D E Decimal Point C

```
const byte seg_decode_array[] = {
  0b00100010, //0
  0b10111110, //1
  0b00010011, //2
  0b00010110, //3
  0b10001110, //4
  0b01000110, //5
  0b01000010, //6
  0b00111110, //7
  0b00000010, //8
  0b00001110, //9
  0b00001010, //A
  0b11000010, //B
  0b01100011, //C
  0b10010010, //D
  0b01000011, //E
  0b01001011, //F
  0b11111101, //DECIMAL
  0b11011111  //BAR
};
```

Each 0 in these bit patterns corresponds to a lit LED segment, and each 1 corresponds to one that is left dark. Obviously, if your display is wired differently from the one in the schematic (note that the A B G assignments may also be different) these patterns will produce gibberish on your display, and you'll need to create your own.

The next global variables are for ISRs. The first of these is msecs, a count of miliseconds, which we use for debouncing the selector button. What does this mean? Well, consider that the ATmega can respond to an interrupt in microseconds to nanoseconds, and that nothing we do with our fingers happens in that time. Further, a switch's contacts are physical. They can bounce around and remake contact. Because the ATmega is so fast, we'll interrupt on every one of them. This makes a push-button selector pretty much useless. We could put a capacitor in series with the push button, so that the jitter is soaked up by the capacitor, but we can do the same thing in software and save a component. As you get deeper into electronics, especially if you're designing for manufacture, you'll find that saving a component by adding software is a constant theme. Ever wonder why nothing has a real power switch anymore?

You might note that milis is declared volatile. This tells the compiler that this variable is always to be read from RAM, rather than from a stored register. Globals modified by interrupt service routines should be volatile, since when a given variable is updated and by what routine (the ISR or a regular function) are difficult to know.

The other value here is a flag, a simple boolean to tell us whether we've rolled since the last time we selected. This is used in loop() and I'll cover it there.

```
volatile unsigned long msecs = 0;
boolean rolled_since_select = false;
```

Roll stores the value of the last die rolled. It's sent to the display system every time loop() is called when rolled_since_select is true.

Die does the same thing, except that it's set by the selector button ISR, so it's declared volatile.

D_select is the value of the die that is selected. It's modified by the selector button ISR, so it's volatile.

LSD and MSD are the variables that hold the *value* of each of the two digits in the display, from 0x0 to 0x11, remembering that 0x10 and 0x11 are bar and decimal, respectively. While they're read by the timer/counter interrupt ISR, they're not changed by it, so they need not be volatile.

Lsd_active is modified by the timer/counter ISR, so it is declared volatile. It's used to determine which digit is currently active.

```
byte roll = 1;
volatile byte die = 4;
volatile byte d_select = 0;
byte lsd = BAR;
byte msd = BAR;
volatile boolean lsd_active = true;
```

Next comes the read_selector interrupt service routine. When the selector button is pushed, and the interrupt sends the ATmega off on the interrupt vector to run some code there, this is the code that gets run.

The millis() function returns the number of milliseconds that have elapsed since the sketch started. We subtract msecs from it. If more than 250ms have elapsed since the last time the selector interrupt fired, we increment d_select, and set msecs to millis(). If it's less than 250ms, a quarter of a second, we're probably looking at button jitter, so we ignore it. Regardless, we set rolled_since_select false. We've just selected. Once all this is done, the ATmega goes back to what it was doing. Most likely executing code in loop().

```
void read_selector_isr() {
  if (millis() -msecs > 250) {
    d_select = d_select + 1;
    msecs = millis();
  }
  rolled_since_select = false;
}
```

What's next is the interrupt service routine for Timer/Counter 3. Note that, like I said at the end of the section about interrupts, it's defined using the ISR() macro, which is part of the AVR GCC compiler. It passes that macro a parameter to tell it which interrupt vector to use. The result of all this is that the interrupt vector points directly at the function below, without the benefit of attachInterrupt.

Once called, the ISR sets all the bits of SEG_BUS to 1s to turn them off (active low, remember?). It then clears the two bits of MUX_BUS that we're using by anding it with 0b11111100.

if lsd_active is true, we're writing to the least significant (rightmost) digit, aka digit two. We set MUX_BUS to LSD_COMMON to activate the correct common anode. We then set SEG_BUS to the value of seg_decode_array[lsd], which looks up the correct bit pattern for the value stored in lsd, and displays it.

if lsd_active is false, we are writing to the most significant digit, or the leftmost digit. This is different.

In some cases, if the most significant digit is 0, we don't want to display it. When using a dedicated driver IC for 7 segment displays (you knew they existed) this would be done with the ripple blanking pin. Here, we do it in software. The only time we want to display the leading zero is when we've rolled 100. In this case, MSD will equal 0xA. So we first test to see if msd is greater than zero, and if it isn't, we don't display anything. If it is greater than zero and it's equal to 0xA, or 10, we set it to zero. Since we've already passed the greater than zero test, that leading zero will be displayed. After that, we set mux_bus to MSD_COMMON to activate the correct digit, (active low, remember?) and look up the bit pattern for msd in seg_decode_array, the same as we did for the least significant digit.

```
ISR(TIMER3_COMPA_vect) {//toggle the LED pin
  SEG_BUS = 0b11111111;
  MUX_BUS = MUX_BUS & 0b11111100;

  if (lsd_active) {
    MUX_BUS = MUX_BUS | LSD_COMMON;
    SEG_BUS = seg_decode_array[lsd];
  }
  else { //we must be writing to the most significant digit.
```

```
    if ((msd != 0)) { //only display msd if it's nonzero.
      if (msd == 0xa) msd = 0;
      MUX_BUS = MUX_BUS | MSD_COMMON;
      SEG_BUS = seg_decode_array[msd];
    }
  }
  lsd_active = !lsd_active;
}
```

The next function is setup(). Normally a mild-mannered function where we initialize the serial console (which we don't do here, since we're not using it), this function is where we'll see all the machinations we do to use the external interrupt for the selector and the counter/timer and its interrupt. It's not that bad.

We start by setting our ports up. We set PORTA up to write on all pins. We set PORTC up to read on the top four pins and write on the bottom four pins. We don't use the other four pins of PORTC for anything, so their setting doesn't actually matter. PORTD is set to read, as you'd expect, and we OR the two bits where we have the buttons wired with 1s to switch on the pull-up resistors. These pins will be positive unless some low resistance connects them to the - bus. Like the roll or selector button.

```
void setup() {
  DDRA = 0b11111111;
  DDRC = 0b00001111;
  PORTC = 0b00001100;
  DDRD = 0b00000000;
  PORTD = PORTD | 0b00001100;
```

Here's where we set up both our timer/counter interrupt and our external INT interrupt. The first thing we do is turn interrupts off, so that we don't get interrupted while we're messing with the interrupts and interrupt vectors. It could happen, and the results could be unpredictable. How does this work? Remember in Configuring Timer/Counters where I mentioned that there's a global interrupt enable flag? This Arduino function sets and clears that flag.

```
  noInterrupts();
```

Next, we set TCCR3A to 0. We don't need any of its bits set. We're not using any compare output pin modes, and the two clock select bits it has aren't ones we need.

```
  TCCR3A=0;
```

TCCR3B is set to 0b00001101. This means the three Clock Source bits are set to 101, which gives us the prescaler dividing the system clock by 1024. Note that we've turned the timer/counter on, too, not that it can do anything with interrupts turned off. It is counting, however. We also set WGM32 on. Combined with the zero values in WGM30,

WGM31 (in TCCR3A), and WGM33 here, this gives our total WGM bit field a value of 0b0100, which is CTC mode 4. The timer/counter will count to the value in OCR3A and reset when it gets there. I've left the bit field definition comment in place for clarity.

```
// ICNC3 ICESN0 - WGM33 WGM32 CS32 CS31 CS30
TCCR3B=0b00001101;
```

Finally we set TCCR3C to zero. We don't want to force any compare-matches. This isn't really necessary.

```
TCCR3C=0;
```

Having set our timer/counter in motion by setting it up with a clock source, we need to tell it what to count to, and what to do when it gets there, so we set OCR3A (remember, it's a 16 bit register, so I set it with a 16 bit hex value. You don't have to use the leading zeros, but I think it makes things clearer.) Hex 00A3 is 163. if we divide a 20MHz clock by 1024, we get about 19.5kHz. If we divide that by 163, we get a bit under 120Hz. Because a given display is only updated on alternate interrupts, this gives us about 60Hz per display which should keep flicker from happening. If you're wondering whether this could have been done using timer 3's own output pins instead of writing the ISR, technically yes, although ripple blanking would have been a problem, and we wouldn't have learned how to write timer/counter ISRs.

```
OCR3A = 0x00A3;
```

Okay, our timer is running, it's counting from zero to 163 about 120 times a second. It's busily setting the OCF3A flag (bit) in the TIFR3 register, but even if global interrupts were on, it wouldn't do anything. Yet. Let's change the timer/counter interrupt mask to let OCF3A get through. The OCF3A flag corresponds to the OCIE3A interrupt enable bit in the mask, so we set TIMSK3 to 0b00000010, and set that output compare interrupt enable bit. Once again, I've left the comment with the bit pattern in place for clarity.

```
// - - ICIE3 - - OCIE3B OCIE3A TOIE3
TIMSK3 = 0b00000010;
```

Now the timer's hooked up. We've already set up its ISR, so once we turn interrupts back on, our display should switch back and forth between the two digits, so fast we can't see them.

We do have to hook up the interrupt for the selector button too, but because it's an external interrupt, and external interrupts are covered by the attachInterrupt mechanism in Arduino, we can set the interrupt, ISR, and mode up in one statement. SELECTOR_INTERRUPT is #defined at the beginning of the sketch, and read_selector_isr is the function we defined earlier to do the job. Note that we DON'T put the usual parens () when we name read_selector_isr. It looks wrong, but I'm sure the attachInterrupt function is creating a pointer to our function. When you do that, the parameters are in a separate pair of parens. In any case, that's the syntax and that's how the Arduino foundation said

it should be. When the button attached to SELECTOR_INTERRUPT's pin shorts that pin to ground, the falling edge (transition from above 0.6v to below 0.3v) will trigger the interrupt, which will jump to read_selector().

```
  attachInterrupt(SELECTOR_INTERRUPT, read_selector_isr, FALLING );
```

Finally, we turn interrupts on, and let the whole thing loose.

```
  interrupts(); //turn interrupts on.
}
```

The loop function, as you're undoubtedly tired of hearing by now, is called over and over again by the Arduino core. A lot of the time, we've stopped that from happening by putting an infinite loop at the end of our code, so we can control when loop runs, and what variables are reset.

I didn't do that here. I let loop() run over and over again. This is the other reason there are so many global variables.

The first thing we do is initialize a byte variable called display_value and set it to zero. Display_value is split into two *decimal* digits to make LSD and MSD later.

If rolled_since_select is true, we set display_value to roll, the last value rolled on the dice device. Otherwise we set it to die, the last die type selected.

```
void loop() {
  byte display_value = 0;

  if (rolled_since_select) {
    display_value = roll;
  }
  else {
    display_value = die;
  }
```

Next, we break display_value apart into its least and most significant *decimal* digits. We get the low digit by taking the modulus of display_value by 10. The high digit is set to the display value divided by ten. There is a huge assumption here. This mechanism depends on the fact that this is integer division. Anything less than zero is ignored. If you tried to mistreat real numbers this way, you'd get bizarre results. By the time we reach the bottom of this part of the sketch, either the last die selected or the last roll rolled will be set to display on the LED display. We don't have to touch the display itself. That's handled by the timer interrupt. It's a little strange when you're used to controlling when everything happens in a sketch to just say "that happens in the background." But it does.

```
  lsd = display_value % 10;
  msd = display_value / 10;
```

The other goings on in the background, whether the user has pushed the selector button or not, may have incremented the d_select variable. We declare a quick constant array with all the possible die_values in it, and then look up the die at d_select, and put

that value in die. If the user has pressed the selector button since the last time the roll button was pressed, rolled_since_select will be true. We don't need to do anything about that right now. Just set the variables and let loop() do the additional processing next time it comes around.

```
  const int die_value[] = {4, 6, 8, 10, 12, 20, 100, 18};
  die = die_value[d_select];

  if (d_select > 7) d_select = 0;
```

Enough with the spooky background stuff and letting loop pick up changes faster than we can see them. Here's the part of the sketch where we actually pole the roll button as fast as we can (once per cycle of loop()). As long as the roll button is held down, keep setting LSD and MSD to BAR and keep incrementing roll If roll goes higher than die, reset roll to 1. Also, keep setting rolled_since_select true. Over and over again. Don't do anything else until the user lets up on the button. Except, of course, that the timer interrupt for the display will go on interrupting. If we were really insistent on this loop running as fast as it can and not being interrupted, we could turn interrupts off. We're not that time sensitive here.

```
  if (!digitalRead(BUTTON_PIN)) {
    while (!digitalRead(BUTTON_PIN)) {
      lsd = BAR;
      msd = BAR;
      roll++;
      if (roll > die)roll = 1;
      rolled_since_select = true;
    }
```

Okay, we're out of the die rolling loop. The user has let up on the button, but we're still inside the original if statement that tested the button.

You might be wondering how we're going to roll 3d6. Just roll from 1 to 21 and add three? No, that gives you different odds. Having generated a random number, if die is 18, and if rolled_since_select is true, we seed the clib random function with roll, and add three calls to the random() function with a low value of one and a high value of 6 together and put them in roll. Those are the correct odds.

```
    if (die == 18 && rolled_since_select) {
      randomSeed(roll);
      roll = (int)random(1, 6) + (int)random(1, 6) + (int)random(1, 6);
    }
  }
}
```

And that's the end. The loop() program is called over and over again forever, so our newly generated roll will be displayed next time around, and of course, our timer/counter will go on interrupting loop() to update the display.

As always, the complete sketch with all the comments (and there are a lot of them) is included here.

The Complete Sketch

```
//Dice Device
//------------------------------------------------------------
// This sketch generates die rolls based on timing randominity.
// Essentially, this method of random generation assumes that
// human reflexes are not accurate to less than about a
// quarter of a second, and a sufficiently fast counter that
// rolls over will generate good random values when switched
// on and off by a human pushing a button.
//
// This sketch generates the rolls for a full set of
// polyhedral dice used in various role playing games,
// including a properly implimented 3d6 roll, and displays
// them to a multiplexed pair of 7 segment displays.
//------------------------------------------------------------

// Precompiler #defines
//------------------------------------------------------------
#define SEG_BUS PORTA
#define MUX_BUS PORTC
// Define the port to talk to the 7 segment display
// and the one that controls multiplexing via a uln2803

#define MSD_COMMON 0x2
#define LSD_COMMON 0x1
// Define the port settings for the multiplexer.

#define DECIMAL 16
#define BAR 17
// Define special characters as integer values.

#define BUTTON_PIN 16
#define BUTTON_PIN_INTERRUPT 0
#define SELECTOR 17
#define SELECTOR_INTERRUPT 1
// Define the pin and interrupt used by the roll button and
// the selector button.
```

```
//Global Variables
//-------------------------------------------------------------
const byte seg_decode_array[] = {
  //7 segment decoding byte array.
  //A B G F D E DP C
  0b00100010, //0
  0b10111110, //1
  0b00010011, //2
  0b00010110, //3
  0b10001110, //4
  0b01000110, //5
  0b01000010, //6
  0b00111110, //7
  0b00000010, //8
  0b00001110, //9
  0b00001010, //A
  0b11000010, //B
  0b01100011, //C
  0b10010010, //D
  0b01000011, //E
  0b01001011, //F
  0b11111101, //DECIMAL
  0b11011111  //BAR
};
volatile unsigned long msecs = 0; //stores time in ISRs for debouncing.
boolean rolled_since_select = false; //Have we rolled?
//If so, this is true. Used in loop()

byte roll = 1; //value of the last roll of the dice.

volatile byte die = 4; //Die, as in d4, d8, d16, etc.
volatile byte d_select = 0; //what die is selected.
volatile byte lsd = BAR; // leftmost digit's integer value.
volatile byte msd = BAR; // rightmost digit's integer value.

volatile boolean lsd_active = true; //Used by timer ISR
// to store whether we're writing to the least significant
// digit or the most significant digit.

//-------------------------------------------------------------
//read selector ISR
//
// This function is an interrupt service routine. It takes no
// parameters and returns no data.
// When the button connected to BUTTON_PIN is pressed, the
// interrupt executes this function. It checks to see if 250ms
```

```
// have elapsed since the last press (for debouncing) and if
// so, increments d_select by 1.
// d_select is actually processed in the loop() function.
//-------------------------------------------------------------
void read_selector_isr() {

  Serial.println("read_selector Fired");
  if (millis() -msecs > 250) {
    d_select = d_select + 1;
    msecs = millis();
  }
  rolled_since_select = false;
}

//-------------------------------------------------------------
// ISR Timer interrupt service routine)
//
// This function takes no parameters and returns no data.
// it selects which digit is enabled (lsb or msb) by clearing
// MUX_BUS and then setting to LSD_COMMON or MSD_COMMON, then
// sets SEG_BUS to the bit pattern for the value in LSD or
// MSD respectively.
//
// This function also impliments ripple blanking, where
// the MSD is not displayed if it contains zero /unless/
// MSD contains 0xa (10), which we only hit if we're selecting
// or rolling percentile dice.
//
// We're using timer 3 for the display multiplexer. ATmega328
// based Arduinos will need to use a different timer.
// This ISR is connected to the TIMER3_COMPA interrupt vector.
// This means that when the timer's value matches the contents
// of TIMER_3_COMPA, this ISR will be called.
//-------------------------------------------------------------
ISR(TIMER3_COMPA_vect) {//toggle the LED pin
  SEG_BUS = 0b11111111;
  MUX_BUS = MUX_BUS & 0b11111100;

  if (lsd_active) { //write to lsd

    MUX_BUS = MUX_BUS | LSD_COMMON;
    SEG_BUS = seg_decode_array[lsd];
  }
```

```
  else { //we must be writing to the most significant digit.
    if ((msd != 0)) { //only display msd if it's nonzero.
      if (msd == 0xa) msd = 0;
      // if msd is 0xa set it to zero after the zero test.
      // leading zero will show on d100 select and 100 rolls.

      MUX_BUS = MUX_BUS | MSD_COMMON;
      SEG_BUS = seg_decode_array[msd];
    }
  }
  lsd_active = !lsd_active;
}

//------------------------------------------------------------
// setup()
//
// This function is called once by the Arduino core. It returns
// no data and takes no parameters.
// It sets up the serial console, sets the ports into the
// necessary states, and sets up the segment decoding array.
// It then turns interrupts off, sets up the timer interrupt
// and attaches the external interrupt to the
// SELECTOR_INTERRUPT interrupt and to the read_selector
// function. After that, it re-enables interrupts.
//------------------------------------------------------------
void setup() {
  Serial.begin(115200);
  //Init the serial console.

  DDRA = 0b11111111;
  DDRC = 0b00001111;
  PORTC = 0b00001100;
  DDRD = 0b00000000;
  PORTD = PORTD | 0b00001100;
  //Set up the various ports. Note the pullup resistors
  //being set on port D.

  noInterrupts(); //turn interrupts off

  //Set up the timer/counter and timer/counter interrupt.
  //-----------------------------------------------------

  TCCR3A = 0; // Clear all pin control bits and
  TCCR3B = 0; // set clock select to 0. Stops the timer.
  TCNT3 = 0;  // Zero the timer count register
```

```
//TCCR3A:
// COM3a1 COM3a0 COM3b1 COM3b0 - - WGM31 WGM30
TCCR3A=0;
// All four COM3 bits remain at zero: the port
// operates normally. Likewise the WGM31 and 30
// bits remain at zero. We only turn on WGM32,
// for CTC mode 4, and that's in TCCRB3.

//TCCR3B
// ICNC3 ICESN0 - WGM33 WGM32 CS32 CS31 CS30
TCCR3B=0b00001101;
// turn on WGM32 for CTC mode 4
// and set the clock select bits to 101 for the
// 1024 prescaler.

//TCCR3C
// FOC3A, FOC3B, no other pins are used.
TCCR3C=0;
// Make sure FOC3A and B are cleared. We don't want to
// force a compare.

OCR3A = 0x00A3; //output compare register.
// 20mHz/1024 (the prescaler) is about 19.5kHz.
// Hex 00A3 is 163. 19.5kHz/163 is a bit over 120Hz.
// Bearing in mind that each digit is refreshed every other
// interrupt, that gives us about 60Hz per digit,
// which is enough to avoid flicker. Persistence of vision
// FTW.
// Yes, this could have been done with a timer/counter
// configuration using the OC3A and OC3B pins and no
// ISR at all, but then we wouldn't learn how to do
// timer ISRs.

//TMSK3
// - - ICIE3 - - OCIE3B OCIE3A TOIE3
TIMSK3 = 0b00000010;
// Set OCIE3A - enable an interrupt when output compare
// match A (TCNT3 = OCR3A) occurs.

//Set up the selector interrupt.
//---------------------------------------------------
```

```
  attachInterrupt(SELECTOR_INTERRUPT, read_selector_isr, FALLING );
  //read_selector interrupt - when the read_selector pin changes
  //from high to low, interrupt.

  interrupts(); //turn interrupts on.
}

//------------------------------------------------------------
// loop()
//
// Called over and over forever by the Arduino core.
// Loop sets the values of the globals lsd and msd for the
// display ISR to pick up,
// processes what the read_selector ISR has set in the
// d_select global variable,
//------------------------------------------------------------
void loop() {
  byte display_value = 0;

  if (rolled_since_select) {
    display_value = roll;
  }
  else {
    display_value = die;
  }

  lsd = display_value % 10;
  msd = display_value / 10;
  //Button is active LOW, so if the roll button is NOT pushed,
  //display either the selected die or the last roll.

  const int die_value[] = {4, 6, 8, 10, 12, 20, 100, 18};
  die = die_value[d_select];

  if (d_select > 7) d_select = 0;
// set die to whatever value was last selected by the
// select ISR with die_value[].

  if (!digitalRead(BUTTON_PIN)) {
    while (!digitalRead(BUTTON_PIN)) {
      lsd = BAR;
      msd = BAR;
      roll++;
      if (roll > die)roll = 1;
      rolled_since_select = true;
    }
```

```
// If the roll button is pressed, display bars in both digits
// and increment roll every time loop is called.Also set
// rolled_since_select to true.

    if (die == 18 && rolled_since_select) {
      randomSeed(roll);
      roll = (int)random(1, 6) + (int)random(1, 6) + (int)random(1, 6);
      //The D18 mode is really 3d6 for chargen.
      // Seed with roll and call random() 3 times
      //Cast random results as int since they are long
      // otherwise.
    }
  }
}
```

Credit where Credit is Due

The Dice Device drew inspiration from an electronic die roller from the early 1980s whose name I won't mention lest I stumble over a trademark. Nevertheless, I will mention that you can read its patent, look at the schematic, and understand the thing's marvelous simplicity here: https://www.google.com/patents/US4431189. I may yet have to build one for my own use with the original ICs, as they're unobtanium on the secondary market.

Understanding the timers and interrupts was something I had to do in spite of the Arduino foundation, rather than based on their documentation. I got most of it from the ATmega1284P datasheet, but I got the code working first based on several web tutorials. There are so many now that I can't put my finger on which one exactly. Nevertheless, this is a good set of tutorials on the subject: http://www.engblaze.com/microcontroller-tutorial-avr-and-arduino-timer-interrupts/

And of course, one must credit the late E. Gary Gygax for inventing the game, whose name is also a trademark, for which the dice were used, and without whose invention my life might have been more productive, but far less fun. https://en.wikipedia.org/wiki/Gary_Gygax

The Stand-Alone version

It wouldn't be difficult to convert this sketch and schematic to a standalone device, powered by a battery. you'd want to change how the common anodes are handled, and you might also want to use the hardware timer/counter and output pins without an interrupt to drive them. You might want to move the roll button to an interrupt, and set the fuses so that the ATmega sleeps any time it's not interrupted. (You'd need to make sure this doesn't turn the timer's output pins off.) Since we don't need many pins, you could do this with an ATmega328, the standard ATmega used for Arduino Uno and similar arduinos. There are a number of core/bootloader setups available to do just that, some of which use the internal oscillator as well. It would not be rocket science to build this device around an ATmega, where the only thing Arduino about it is software.

Likewise, it would not be difficult to copy the original device from the schematic listed in the expired patent, though if you're contemplating selling the finished devices you'd be well advised to make sure there aren't *other* patents you'd be infringing. Patent trolls are a way of life today, and the only industry I know of with more lawyers and less sense than the computer industry is the gaming industry.

The truth of the matter is, it would be far easier and likely far better received to write the dice device entirely in software for your favorite smartphone/tablet platform.

The same technique used here to drive a two-digit display could drive up to eight, although I'd seriously recommend using a UDN2981A, which is a high-side driver. If you wanted to go with more than eight digits, you could add a second such IC and use another port. At some point, it will make more sense to use something like an SPI or I^2C GPIO expander like the MCP23017 to do common anode switching. Although their serial interfaces will eventually limit your speed, you'll probably run out of board space for the displays long before that happens. Displays also come in more than two digit packages, but if you're looking at more than a few, consider a serial LCD panel instead. They may be more cost-effective.

CHAPTER 11

Z80 Explorer

Well, here we are. The last chapter. This one's the payoff, the big one, the microprocessor chapter. This is the chapter where we finally go inside a system not too different from the Cestino's ATmega1284P microcontroller and really see what goes on behind the curtain. We're going to build the Z80 explorer, shown in Figure 11-1.

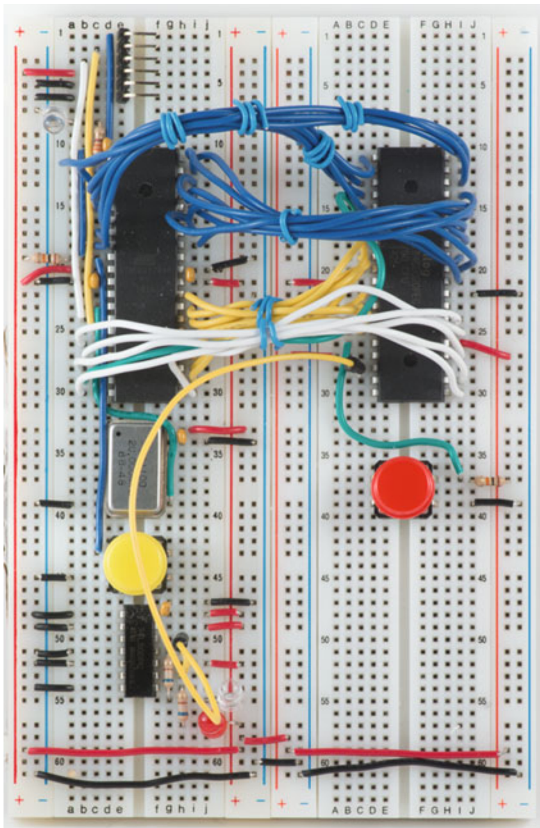

Figure 11-1. *The Z80 Explorer*

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1_11

In previous chapters, like Chapter 10, where I talked about interrupts at some length, you've had to take it on faith that I'm right. Certainly the code based on that understanding works. You've also had to take on faith that somewhere between the C++ Arduino sketch and the "bare metal" of the ATmega1284P, our intent was translated, reasonably, into machine instructions. It certainly seems to work, but there's a magic-y feel to the process, and I've said from the beginning that there is no magic in electronics and computers. (If you dig down far enough, there's quantum mechanics, but that's not actually magic either. It just seems that way.)

In this project, we're going to hook an ancient type of microprocessor, the Zilog Z80, to the Cestino, and program the Cestino to emulate all the other hardware that would normally go with the microprocessor. We're going to run the microprocessor's clock very slowly, so we can look into memory in real time. To make the Z80 work, we'll hand-assemble some programs (more on this later). You won't have to take on faith how interrupts work. You'll be able to see them go.

So let's get started. We have a lot of ground to cover.

The Stuff You Need

For as big as this project is, its part count is very low. There are three. Two of which you've used in other projects.

New Parts

- 1 10kΩ resistor.
- 1 0.1µF capacitor. Tantalum, like the ones we used to bypass the power supply on the ATmega and the oscillator in chapter 2, and the 74xx00 that is our logic probe in chapter 7. Ceramic disk caps will work, but tantalums are easier to find room for.

New or Used Parts

- One Z80 CPU. Sadly, not all Z80s will work. The oldest types, the NMOS types, won't accept a clock as slow as the one we'll be giving it. "Modern" Z80s in the Z8400 or Z84C00 series will do just fine.
- There's some question where the NEC clone, the µPD780 will. Its datasheet says it is effectively a static design, meaning it will accept a clock with a speed of zero, but that below a few hundred kHz is not guaranteed. I have one of these, and I think I developed the original prototype of this project with it, but as my µPD780 has since died, all I can say now is look up your specific IC's datasheet online, and look in the AC characteristics. If, as the Z8400/Z84C00 models from Zilog do, it lists no upper limit to the clock period, this means it will accept a static clock. They'll often mention a fully static design in the datasheet as well. You can also just plug your Z80 in and see if it works. It's junk, right?

- If you're ordering a new one (which might be the safest approach) any of the Zilog Z84C00 family, like the 20MHz Z84C0020, I used) will work. They differ only in their *maximum* clock frequency. As always, make sure it is the 40 pin through-hole DIP version, as the Z80 has been around a long time and comes in a lot of other packages.
- A tactile button. As with Chapter 10, this button needs to be a momentary switch, normally off. After that, use whatever button you like. Be advised that some tactile buttons (like mine) don't like to stay in the breadboard, and that they may switch adjacent terminals rather than opposite terminals like you'd expect. If they really give you trouble, you can solder them to a piece of protoboard, add some pin headers, make some connections, and use the pin headers to connect the switches to your breadboard. If there were any projects in this book after this one, I'd probably do that.

Z80 Microprocessor Anatomy

As microprocessors go, the Z80 is a simple beast. At their hearts, modern CPUs are nearly as simple, but they're complicated by multiple cores, vast, 64 bit busses, caches, complex memory read and write cycles, one or (frequently) more floating point units, vector units, caches, and so on. Nearly all these additions are designed to make the processor do more per clock cycle than the Z80 can. For a given speed in megahertz or gigahertz, a limitation of the very transistors from which the processor is made (among other things), a more efficient processor will get more done, simple as that. All the complexity does make them harder to understand. Even the ATmega1284P, heart of the Cestino, is more complex, despite being an eight bit processor itself. The addition of internal RAM, flash, and all the various I/O modules and functionality that are built in make the ATmega powerful and useful in the Cestino, but the complicate understanding it.

The Z80 is different. It is an early design, when simply making a microprocessor for a reasonable price was the great technical challenge. Its internal data bus is only eight bits, which we're already familiar with from the Cestino. It handles eight bits at a time, has no caches, no floating point units, no vector units, no nothing. All that's left is the *processor,* the bare minimum needed to be a functionally complete CPU.

You'll notice as we dig deep into the Z80's innards, especially if you've had formal programming training, that a lot of what you find here are classic computing data structures. Variables, stacks, pointers; they're all here. This is not an accident. A CPU is essentially a program itself, with various functions called by instructions with or without data. If you always thought of yourself as a software person like I did, this may be a bit of a shock. There's no there there. There's no intelligence in the CPU. It's a program written in transistors for running other programs. You already know everything you need to understand what goes on in the CPU.

Busses

We'll start with the busses. There are four of them in the Z80, three of which have direct connections to the outside world.

Memory Address

The memory address bus, despite being a 16 bit bus, is the simplest conceptually. It's a series of output lines not unlike a pair of the ATmega's ports in output mode. The state of the output lines together is controlled by logic in the CPU which connects them to one of several different registers, depending on what the CPU is doing. At the beginning of an instruction fetch cycle, the address bus is connected to the PC, or program counter register. This register will hold the address of the instruction being fetched. (More on that later in the Registers section.)

If an instruction requires data, as the cycle continues, the Program Counter increases by one for each byte the instruction requires. This is handled by the control logic, which is also discussed later.

If an instruction demands that the CPU read an unrelated area of memory the address pins can be connected to one of several other 16 bit registers, depending on which instruction is used.

Data

The data bus is even more like an ATmega port than the memory address bus. It's 8 bits wide, and it's bidirectional, which means you can read from it and write to it. The big difference is that you can do almost nothing "in the background" with the CPU without touching the data bus. The ATmega 1284P has an 8 bit data bus too, but it's not connected directly to the outside world. Our ports are, themselves, peripherals.

The Z80's data bus is where nearly all the communication with the outside world really happens. Any data the CPU sends to or gets from memory goes through here. Any data the CPU sends to a peripheral device goes through here. If you've noticed that many of the devices we've tinkered with in other chapters have wanted an 8 bit bus, it's because they were designed to be connected to the 8 bit data bus of an eight bit CPU. Even ATA, which was designed in the days of the sixteen bit ISA bus, is still controlled entirely with 8 bit commands.

Control

The control bus is kind of a hodge-podge of inputs and outputs, as will become (painfully) obvious when we connect it to the ATmega and write the sketch to control it. There are 12 control signals, only about half of which we'll actually use.

System Control

The system control registers are all outputs. They're there so the CPU can tell the rest of whatever system it's connected to what it's doing, and what it wants.

/M1

The /M1 signal tells the system that the CPU is in the instruction fetch part of a machine cycle. Whatever memory it's fetching from, it's fetching an instruction, as opposed to data. This can be important for timing purposes. /M1 is turned on once per machine cycle, as we only read one instruction per machine cycle.

/M1 could allow us to turn the Z80 into something like the ATmega, where instructions are read from one memory source, like Flash or EPROM, and data is read from RAM. We could connect some logic to the /M1 signal, probably combined with /MREQ, to chose which chip-select signal you switch on to turn RAM off and Flash on, for example. Don't worry. Our use for the /M1 signal in this project is very simple: tell the sketch when the Z80 is reading data or fetching an instruction.

/MREQ

/MREQ tells the system that the Z80 needs to use the memory system. Specifically, it says that the CPU has put a valid address on the Memory Address bus. /MREQ is pulsed active (low) several times in a machine cycle, and is active low.

/IORQ

/IORQ has two functions. First, it tells the system that the CPU is trying to transfer data to an IO device, and that the lower half of the address bus has a valid address for an IO device.

/IORQ is also generated during the M1 phase of the machine cycle when the M1 signal is active) during an interrupt. This is the CPU telling the system that if you're using a mode 2 interrupt, you can put the interrupt vector on the data bus and select where the interrupt takes you. This takes some external logic. Unsurprisingly Zilog had a series of companion ICs to the Z80 that provided timers that could interrupt the system (not that different from the ones on the ATmega) and interrupt controllers that could provide for many different interrupt-using devices and pass an address accordingly.

/RD and /WR

The /RD and /WR signals are active low, and tell the system that the CPU is trying to read or write (as the abbreviations suggest) to RAM. /RD may be pulsed active (low) up to twice per machine cycle, once for the instruction fetch and once for a data read, whereas /WR will only pulse on in the write phase of a machine cycle. These signals are used to put whatever RAM is connected to the CPU in the right state to receive data from the CPU (/WR) or transmit data to the CPU (/WR).

/RFSH

The /RFSH (refresh) signal takes some explaining. We're not using it at all in this project, so if you're not interested in the care and feeding of DRAM, you can probably skip down to the CPU Control signals later in this chapter.

You've undoubtedly heard of DRAM, (Dynamic RAM). It's the kind of ram your desktop computer uses. You might know that, unlike SRAM (Static RAM), dynamic RAM requires refreshing. Here's what that means.

If you look at Dennard's original 1966 patent, you can see that a cell of DRAM is essentially a FET and a capacitor to hold its state either conducting or not conducting. The patent also covered a way of addressing lots and lots of these cells by rows and columns, and the technique for manufacturing them on silicon wafers. The important point is that the FET will hold its settings only as long as the capacitors remain charged, and the capacitors are very, very small.

DRAM in those days was accessed by splitting the row and column address into two parts, allowing the DRAM chip to multiplex its address pins, and also allowing the DRAM to save a pin, since row 0 and column 0 were valid. Thus, a 16k DRAM IC like the 4116 (very common in Z80-based systems) could have seven address pins, and one data pin (each IC only held one bit. You needed eight of them for 16kb of RAM.) In 1980 they went for about U.S.$44 each. You wired them in parallel except that each one would be connected to a different line of the data bus to the CPU.

In order to access the DRAM, you pulled the /RAS (row address strobe) line low, sent the row address (the lower seven bits on the cpu's memory address bus) then pulled /CAS (column address strobe) low as well and sent the column address from the next seven bits up. If you cared to use page mode, you could pull /CAS low multiple times and send multiple column addresses.

Why am I telling you this? Because every time you pull /RAS low, that row of bits' capacitors were recharged. The row was refreshed. If you accessed all the bytes in your RAM array before the capacitors discharged (about 2ms on the 4116 datasheet I'm looking at, you wouldn't have to refresh it at all. In fact, the Apple II computer did exactly this, using its video circuitry to scan every row of DRAM bytes in the system. Steve Wozniak was and is a very clever man.

For most computers, perhaps lacking video circuitry at all (the earliest Z80 computers required a separate "dumb" terminal like an ADM3A or a DEC VT100), and certainly lacking Steve Wozniak, refreshing meant that at some point you had to stop the computer doing its business, generate row addresses and pull the /RAS signal low repeatedly to make sure that all the rows were refreshed before they ran out of time.

The Z80 provides the /RFSH line to tell the system (particularly memory) that the address bus's lower 7 bits are set to the next memory row in line to be refreshed, and it can now refresh that row while the CPU is decoding the instruction it just loaded and therefore doesn't need RAM at all. With only a little extra circuitry, the Z80 gave you dram refreshing for free.

This, more than any other single feature, is why the Z80 became so much more popular than the Intel 8080 whose instruction set it copied exactly. RAM was expensive, and logic wasn't cheap either. Having the CPU do this for you without making the CPU wait for RAM was a big deal.

DRAM refresh timing is a tricky business, and it still is today. When you read overclocking sites talking about CAS timing, they're talking about the latency between when the /CAS signal goes low and when the data shows up on the DATA signal of the IC. Also, on more modern DRAM ICs, if you pull the /CAS signal low before /RAS, the DRAM will generate the next row address that needs refreshing for itself. This is called /CAS before /RAS refreshing.

Now you know.

CPU Control

The CPU control signals of the Z80 are a mixture of one output and four inputs. With the exception of /HALT, these signals all control some aspect of the Z80's operation.

/HALT

/HALT is the oddball of the control group, in that it's an output. The CPU tells the rest of the system that it's halted through this signal, and the only thing that will get it to resume is an interrupt. If you add an external driver and an LED (the outputs on the Z80 can only sink a couple of mA) you could drive a HALT led from this signal. You could also plug the logic probe in there. We don't use the /HALT signal for anything in this project.

/WAIT

/WAIT is an input, and a very important one to know what it's doing. Wait tells the CPU to wait for IO. Back in the day, we used to talk about low/no wait-state memory. This meant that the RAM wouldn't tell the CPU to wait, via the /WAIT line. In other words, RAM could keep up with the CPU. We use /WAIT extensively because we will be simulating RAM with the ATmega using the ATmega's memory and a couple of interrupts to get the job done promptly. It's still nowhere near as fast as hardware RAM. Wait lets us keep the CPU from reading at the wrong time and getting really, really confused.

/INT

/INT is one of two interrupt signals on the Z80. Remember interrupts from the last chapter? We're going to be masters of interrupts by the end of this one, using them both on the Z80 and the ATmega. Whereas the ATmega has three INT pins, the Z80 has just two, of which this is the more flexible. Pulling this signal low causes the Z80 to stop what it's doing at the end of the machine cycle it's in and do one of several things, depending on how the interrupt is configured, which I'll cover at some length down in the programming section of this chapter. In any case, this is the pin that triggers it. Unlike the ATmega's interrupts, we get no choices whether this interrupt fires high or low or on the edges. It is active low. Like the ATmega, the /INT signal is disabled (masked) by default, but can be enabled from software.

We will use this signal, so it's wired to the button with a pull-up resistor, switched to the - bus.

/NMI

/NMI stands for Non-Maskable Interrupt. When /NMI is pulled low, the Z80 will interrupt at the end of the current instruction cycle and will restart and fetch its next instruction from address 0x0066. This works exactly the same way as the mode 1 interrupts we'll be using with the /INT line down in the program section, save that the address is different. I'll explain in detail there.

We don't use this signal, and it's very important to keep it out of mischief, so we pin it to the + bus.

/RESET

/RESET resets the CPU. The program counter goes to zero, all signals are reset to their default states, normal interrupts are disabled, and the interrupt mode is set back to 0. (More on that down below in the interrupt program). We're handling this signal from the ATmega.

Bus Control

/BUSRQ and /BUSACK

/BUSRQ and /BUSACK are control lines for the bus of the system. In fully fledged computer systems, often another device besides the CPU needs to talk to memory (usually) or another IO device. The CPU is slow. It takes several clock cycles for a single machine cycle, and if we don't *have* to use the CPU for something because we're just moving a large amount of data from device A to device B, it's possible to tell the CPU to pause and let the external device control the address bus, data bus, and the /MREQ, /IORQ, /RD and /WR signals so the external device can control the bus. /BUSRQ requests that this happen, and /BUSACK is the CPU's acknowledgement that it's waiting to be allowed to talk to the bus again. We don't use these signals, but if you do, you should know that /BUSRQ blocks the CPU's access to the RAM, so if you're using CPU refreshing, you won't be while the CPU doesn't control the bus.

Internal

The internal bus is the last one we'll talk about. As the name suggests it has *no* direct connections with the outside world. This is the bus we use when we move data from the data bus to a register, or from one register to another, or from a register to the ALU (arithmetic/logic unit). This is the bus that controls the data bus. The internal data bus on the Z80 is eight bits wide, and it's this bus which determines that the CPU is an eight bit processor. We'll use this bus a lot, but we don't get a lot of say what happens with it.

Registers

Registers are memory, if you like, inside the CPU. You can also think of them as variables.

We've talked about registers before discussing the port registers of the ATmega, and these registers are very much in that same vein. Most hold one byte each. Some are paired and can be used with their pair-mate as a sixteen bit register, or as two eight bit registers. Some, as they are primarily for holding memory addresses, must be used as sixteen bit registers. Some are modified automatically by the CPU as the ATmega's port registers are, and some are just a place to put data. Registers are at the heart of programming.

Program Counter

The Program Counter (PC) register is a sixteen bit register that keeps track of the address in memory we're at in this machine cycle. The program counter generates the address on the address bus when the Z80 requests memory access. The program counter is what gets changed when an interrupt happens, reset when the Z80 is reset, and so on. Memory holds instructions and parameters to those instructions. The program counter is what ensures that those instructions get read and executed in the order they were put in memory.

You can't read the PC register directly, although you can set it with a JP (jump) command. If you've ever programmed in BASIC, and JP sounds like GOTO to you? Well, it is, exactly.

Accumulator and Flag Registers

The accumulator (A) register is where eight bit math goes. To do math on a Z80, you load the accumulator, then load another register with another value, then issue an instruction to add that other register. The accumulator is where the result lands.

Closely tied to the accumulator register is the Flag (F) register. The various bits of this register have different values. In the usual 0b order, from bit seven to bit zero, these are: S, Z,(not used), H, (not used), P/V, N, and C.

S: Sign Flag

The Sign flag is equal to the most significant bit in the Accumulator. Because the Z80's ALU assumes signed integers, this bit will be the sign bit.

If there's one thing that is a lot more complicated at this level of computing than with higher level languages like the C/C++ we've been using for sketches, it's math. It gets worse when you try to do floating point math, as we will in the sketch for this project (but not in assembly on the Z80.) For CPUs with no floating point unit, the floating point math functions are part of the program. Floating point math is expensive in compute time on systems with no FPU.

Z: Zero Flag

The Zero flag is set if the byte in the Accumulator is Zero as the result of a calculation. It's set if the math unit is doing a compare and the two values compared are equal. It's also set (one) if a bit is being tested in a register if the bit tested is zero, and reset (zero) if the bit tested is one.

H: Half Carry Flag

The Half Carry flag is used by the Decimal Adjust Accumulator instruction to correct the decimal point when dealing with packed BCD notation digits.

That's great, what does it mean?

Simply this: Because computers are very often in the business of dealing with decimal numbers even though in the most literal sense they can't, many programs take advantage of the fact that 0x0 to 0x9 can represent decimal digits as well, so each byte of memory can store two digits of decimal represented numbers, zero to 99, as we did in the Dice Device's displays, if not in memory. This wastes half the permutations of a byte, but for decimal operations it simplifies things. The half-carry flag is set when bits get borrowed from the high digit to the low digit and from the low digit to the high digit.

P/V: Parity/Overflow Flag

The Parity/Overflow Flag has a number of different functions. If your math operation generates a value greater than 127 or less than -128, you've overflowed your 8 bit Accumulator, so this flag will be set so you can tell.

The flag is also used to determine the parity of the byte in the Accumulator if you've done logical operations on it or rotated it. Parity is odd (P/V is zero) if there are an odd number of 1 bits in the byte, and it's even (P/V is 1) if there are an even number. Parity is used, among other things, to verify that a byte has been sent or received correctly.

N: Add/Subtract Flag

The Add/Subtract flag is exactly what it sounds like. If it's set, it means we're doing a subtraction, which may be important for interpreting the results. There's more about this flag in the datasheet.

C: Carry Flag

The Carry flag is set when a math operation generates a value bigger than the one byte in the accumulator can handle, either through addition or subtraction. (Carry can also mean borrow if you're subtracting.) It's reset by any ADD or SUBtract that does not generate a carry/borrow, or any logical operation—AND, OR, or XOR. During the rotate instructions (RRCA, RRC, SRA, and SRL, if you wondered) it will hold the final bit at whichever end of the byte you're rotating toward when the final value is shifted out. If you rotate one of the registers, or the accumulator until it's empty in either direction, the Carry flag will hold the last bit.

Index Register

There are two index registers in the Z80, IX and IY. Both of them are 16 bit registers. This is for indirectly accessing memory.

Wondering what that means? It's simpler than it sounds. Moving 16 bit values around in an 8 bit computer is compute-expensive, so if all your data for the next few instructions are within 255 bytes of each other, you can set the address of the first byte in IX, and then access the rest of the bytes with an instruction that takes an 8 bit offset to that 16 bit address.

General Purpose

There are six general purpose registers in the Z80, in three pairs of two: BC, DE, and HL. These registers can be used individually as 8 bit registers (B,C,D,E,H, and L) or with their partners as 16 bit registers (BC, DE, and HL.) As you go through the list of instructions, you'll find that some instructions don't work with some registers. HL, for example, is clearly intended as a pointer to memory. (More about pointers a few sections down. We'll be dealing with them a lot in this project.) D is the register used for the offset when doing indirect addressing (index + offset - IX or IY+D). The IX+D pair is also most often used as a pointer.

There are quite a few instructions for accessing a given memory location at HL and remarkably few for other general purpose registers.

General purpose registers contain only what you put in them. They are variables, in the truest sense.

Stack Pointer

SP, the Stack Pointer. Remember how I mentioned pointers? The Stack Pointer is a 16 bit register dedicated to the Z80's stack.

The Z80's stack is called a "Software stack." This means that rather than have memory inside the Z80 for the stack, which is a standard part of microprocessor anatomy, the Z80's stack uses external memory. When you set the stack up, you point the stack pointer to the *highest* memory address it's allowed to use, and the Z80 stack will expand toward the *lowest* memory address. If it intersects with your data or your code, the stack system will happily overwrite both, just like you can overwrite the stack with data. One thing about machine language/assembly language: there's very little protection from yourself at this level. I'll mention this again later.

In any case, the stack is used by a number of different systems in the Z80, not least of which is the interrupt system and the call system. If you call another address in code, the first thing that happens is that the value of the PC (program counter) is pushed onto the stack. When your code issues the RET instruction, the call system pops the PC's value back off the stack, sets the PC to it, and away you go, assuming you haven't pushed anything else to the stack that you've not subsequently popped between the two.

Evil Twin Registers

If you've looked at the Z80 datasheet, you may have noticed a whole second set of the general purpose registers, and the accumulator and flag registers. They're real. You can swap the two sets of registers with a single instruction, and it's a very fast operation. If, for example, you have an interrupt handler (an ISR, in ATmega parlance) that uses the accumulator and HL, you can push the values of both onto the stack when the ISR is called, then pop them off when the ISR is done before it returns. Or you could execute one command, switch registers, and have a set of registers all to yourself for the ISR. When you're done, just switch back, call RETI (return from Interrupt) and let the CPU handle popping the PC register off the stack.

There's a lot of that kind of tweaky optimizing funcitonality in the Z80. It's a CISC (Complex Instruction Set) processor designed for assembly language programming, unlike the ATmega. It will reward hand-optimization like I just described far more than modern RISC instruction sets will, which instead use all those transistors to speed up *all* the instructions. Different philosophies, different times.

The ALU

The Z80's Arithmetic/Logic Unit (ALU) is a pretty limited beast. It can Add, Subtract, AND, OR, XOR, Compare, Increment and Decrement integers. That's it. Need multiplication and division? Write a program that does that. Need floating point math? Write a program that does that. Do you pay a *huge* penalty in performance using software instead of the ALU's built-in functions? Yes, absolutely. Zilog actually fixed the multiplication/division problem with the Z180, and while they, like their older Z80 brethren are cheap and plentiful, they aren't available in quantities of less than 120 in DIP format. If you were building a Z80-based system today, there are floating point units based on programmable DSPs, such as the Micromega uM-FPU64. You *might* also be able to use such an FPU for fast integer multiplication and division, although it would be worth figuring out whether the extra steps of communicating with these devices over SPI, I^{2}c, or UART (TTL level RS232) would be slower than having the Z80 multiply by adding over and over again.

The ALU of the Z80 breaks its operations into two groups: 8 bit math and 16 bit math.

Eight Bit Math

There are 17 8 bit math instructions on the Z80, but they can be grouped into eight groups: ADD, SUBtract, AND, OR, XOR, CP (compare), INCrement, and DECrement.

The ADD Family

The ADD family of instructions add various operands to the value stored in the Accumulator. This is how math is done on the Z80: values are stored in the Accumulator (A register) and modified mathematically.

So ADD A, r adds the contents of one of the eight bit general purpose registers to the value stored in the Accumulator.

ADD A, n adds the value n (an eight bit value) to the value stored in the accumulator.

ADD A, (HL) is tricky. In assembly, when a register or a memory address is enclosed in parens, it means that this is a pointer. HL, in this case, would be set to a sixteen bit address in memory, but (HL) means the *contents* of that memory cell, which will be an 8 bit number. This is the same with ADD A, (IX+D) and ADD A, (IY+D). IX and IY are index registers which will hold a 16 bit value, and the D register is used as the 8 bit offset from that address. The whole shebang is an address in memory, and is enclosed in parens, making it a pointer. What you *get* from the (IX+D) pointer will be the contents of that memory address, not the value of IX+D.

ADC is actually part of the ADD family, save that it is add with carry. ADC will add the value of the carry flag (remember the carry flag in the F (flag) register?), which is zero or one, plus whatever operand you give it to the value in the accumulator. If you actually are doing a software multiply, you'll probably need this. (Don't worry. We won't.)

Which Opcode is Which

We've seen the assembly instructions for the various adds, but one thing that the Z80 datasheet makes a little confusing is how those instructions translate into opcodes. When you look at ADD A,r; add the contents of register r to the Accumulator; there's no opcode listed. When you hand-assemble code, you have to put the opcode together yourself.

ADD A,r has seven permutations, depending on which register you want to add to A. To get the opcode you need, you have to figure out which register you're adding, then get the fixed bits of the command, and add the bits of the register you want to it.

Suppose you want to add general purpose eight bit register B to the Accumulator. That's the ADD A,r instruction. If you look on the table in the datasheet, you'll see that the B object code is 0b000.

The ADD A,r instruction's value is 0b10000___, where the last three bits are the object code of the register you want. If we fill 000 in for the last three bits, we get 0b10000000, or 0x80. This is the opcode for ADD A,B.

If you want C instead of B, no problem. C's object code is 0b001. We put 001 in for the last three bits of the opcode, and get 0b10000001, or 0x81. And so on. The three bit object codes for all the 8 bit registers you're allowed to touch stay the same for every 8 bit instruction. For reasons known only to themselves, Zilog's engineers did not assign an object code for pointers like (HL) or (IX+D). They just list the opcodes directly in those instructions.

The SUB Family

The SUB and SBC families has the same members as the ADD family, but they're grouped together differently. SUB is listed as SUB s, where the s operand can be any register, a number, or a register used as a pointer. The object codes are the usual suspects too.

Compare

CP, compare, is a little different. It compares the value in the accumulator with the value of the other operand. Sure, the other operand can be all the usual suspects—explicit values, other registers, pointers to memory like (HL) or (IX+D), but it's a weirdo in how it returns its results. The results are not set in the Accumulator. They come up in the flags attribute. If the compare is true, the Z (zero) flag is set to 1. If it's not, the Zero flag is set to zero. Compare also sets the H flag the P/V flag, the N flag, and the C flag as appropriate. The Z flag is the most useful, though.

Increment and Decrement

A lot of times what you really want to add or subtract to a given register is one. Consider the PC (Program Counter) The Z80 *increments* this register at least once every machine cycle, and often two or three times. If you guessed that the Z80 probably uses the very same hardware for that as for the INC instruction, you're probably right. INC is much faster than setting a register to one and adding it to another register.

DEC does the same thing in the other direction. Where INC adds one to the register or memory location, DEC subtracts one, with all the same advantages.

INC and DEC do not use the accumulator unless told to. You can increment or decrement any eight bit register you're allowed to touch (including the Accumulator) as well as values in memory referenced by the usual pointers.

The usual system of opcode building where you add the three bits for the given object code into the rest of the instruction is also used, save that the object code goes in the middle of the instruction. Like this.

If you want to INC C, you look up INC r, and discover that the opcode is 0b00___100. The object code for C is 001, so you put those three bits, right to left, in the missing slots, and get 0b00001100, or 0x0C (or simply 0xC, but it's easier to match up with a full byte using both hex digits.) If you want to increment the Accumulator (INC A), you apply 111 to the opcode and get 0b00111100, or 0x3C.

Pointers like (HL), (IX+D) and (IY+D) have their own opcodes, as is often the case.

INC and DEC set the usual flags.

If you're doing a counting loop, it should be obvious there are two ways to do it. Set the value of a register, INC it, and compare it, useful if you're using the register as part of an address (as in indirect memory addressing using (IX+D) where you're incrementing D) or set your register to the maximum value of the loop and DEC it until the Zero flag is set. When we do some simple programs for the Z80, we'll do both.

Sixteen Bit Math

The 16 bit Arithmetic group is similar to the 8 bit group, save that while there are fewer registers it can access (as we have to use pairs of registers instead of single 8 bit registers), more of them can have math performed on them. You can't use the accumulator at all, since it's an 8 bit register, and the F (flags) register is not directly user settable. Sixteen bit Arithmetic instructions *do* use the flags in the F register pretty much the same way their 8 bit counterparts do.

The other thing to note is that the object codes for 16 bit registers are only two bits long.

There are four of the usual families of math instructions in the 16 bit world: ADD/ADC, SUB/SBC, INC, and DEC. All of them are designed to deal with *addresses* or other 16 bit values in registers. None of them deal in pointers like (HL), as the value stored at any given address in memory must be 8 bits long.

ADD/ADC/SBC

The ADD/ADC and SBC families of instructions access specific registers.

ADD HL,ss works pretty much like ADD A, s, save that HL is just a general purpose register (usually used as a pointer to memory), and ss denotes another general purpose 16 bit register. You build the opcode the usual way, too.

If you want to add the value of a register to HL, for example, you look up the opcode for 16 bit ADD HL,ss, and discover it's 0b00__1001. Looking up the object code of the register pair HL, it turns out to be 0b10, so as usual we assemble the opcode to be 0b00101001, or 0x29.

If we place 0x29, 0x10, and 0x00 in three bytes of memory, in that order we've instructed the Z80 to add 0x0010 (decimal 16) to whatever is in HL already. You saw that, right? Little-endian means our lowest significant byte comes *first* in RAM, so 0x10 before 0x00 comes out equalling 0x0010. If you wonder why I insist on putting the leading zeros on hex values, here's where it pays off in clarity.

Always make sure you know what value is in a register (or the Accumulator for that matter) *before* you use it in math. Also, if you are programming within an operating system, the operating system may well be using any given register, so it behooves you to either store that register and restore it when you're done, or know what registers you're allowed to use.

The 16 bit ADD family also includes the ability to add to the IX and IY pointers, but not to ADC them.

ADC works pretty much like ADD, except that it carries.

The 16 bit SBC family is even more limited. There are instructions to subtract any given pair of 16 bit registers from HL, and all 16 bit subtractions use the carry flag. The 16 bit SBC is also a rare 16 bit instruction: there are two bytes in a row used to indicate it.

So if you want to subtract the value of BC from HL, you look up the opcode. The first byte of the opcode is 0xED. The second byte is the one you have to build. The skeleton is 0b01__0010. Register pair BC's object code is 00, so the second byte becomes 0b01000010, or 0x42. To subtract a given number from whatever is in HL, using BC, you'd first have to load that number into BC, then put the opcodes 0xED,0x42 into memory in that order. There's no other way to do sixteen bit subtraction, and you will be using the carry flag, whether you like it or not.

Why is 16 bit subtraction so limited? The Z80 was designed when you didn't get many transistors on a given IC. The original Z80 had only 8500 transistors *total,* whereas the CPU in my monster mac has 1.4 billion transistors. Every function they added to the Z80 cost transistors, and if an instruction wasn't used often, like 16 bit subtraction (in an 8 bit processor) it didn't make the cut.

INC/DEC

For all that the ADD/ADC and SBC arithmetic family is limited in the Z80, the INC and DEC families are full-featured. You can increment or decrement any general purpose register, as well as IX and IY.

INC ss, the increment instruction for any general purpose register, is a single byte instruction, with the usual object codes for 16 bit registers.

As always, the IX and IY registers have their own opcodes. In this case they are sixteen bit opcodes.

Logic, Rotation, and Bit Twiddling

There are other functions of the ALU. It deals, as the name suggests, with logic, also with bit rotation (which is similar to but not the same as bit shifting) and setting, resetting, and testing single bits.

Logic: AND, OR, and XOR

Boolean Logic

AND, OR, and XOR all work the same way. Set the Accumulator to the value of itself AND, OR, or XOR the s operand, which can be any general purpose register, an explicit value, or the usual pointers. Once again, for some permutations, you build the opcode yourself by putting the object code together with the prefix just as you do with ADD, and other opcodes are simply given outright.

Rotation

The RL and RR families of instructions do rotate left, rotate left with carry, rotate right, and rotate right with carry on a variety of registers.

The Z80's rotate instructions are quite different from the Arduino bit shift operators. They only shift the byte by one bit per call, and unlike the Arduino bit shift, the Z80 rotates *do* wrap bits back around to the other end of the byte. They also interact with the carry flag, depending on which rotate you call.

RLCA (0x07), rotates the byte in the Accumulator (register A) left one bit. Bit 7, the leftmost bit, is stored in the carry flag and is also set on bit 0 of the accumulator.

RRCA (0x0F) does the same thing in the other direction. All the bits of the accumulator are rotated right one bit, and the right most (lowest) bit of the byte goes to the carry flag and also is placed in bit 7.

RLA (0x17) rotates all the bits in A left. Whatever was in the carry flag moves into the rightmost (lowest) bit of A, and whatever was in the leftmost (highest) bit of A is copied into the carry bit.

RRA (0x0F) rotates all the bits of the accumulator right one bit. The carry flag is copied into the leftmost (highest) bit of A, and the rightmost (lowest) bit of A is then copied into the carry flag.

The rest of the RL family of instructions work the same way on different registers. They're two-byte (sixteen bit) instructions, so they will take longer to execute. Some of them also act on sixteen bit registers.

Bit Twiddling: Set, Reset, and Test

The BIT instruction tests one bit in a given register. If it's zero, the Z flag is set, otherwise the Z flag is reset to zero.

The BIT instruction is a two byte instruction. The first byte is 0xCB. The second byte contains a prefix code, 0b01, three bits to select which bit you want to look at, where 0b001 would be bit 1, second from the right, and 0b110 would be bit 6, second from the left. The remaining three bits select the usual object codes for registers. So if you want to see if bit 3 of register D is set, you'd use the first byte, 0xCB, and your second byte would be 01 for the prefix, 011 for bit 3, and 010 for register D. Put together it would be 0b01011010, or 0x5A. Then you'd check the flag register to see if the zero bit is set. If it is, your bit was set to zero.

There are versions of BIT to test memory locations with the usual array of memory pointers, too.

SET and RESET work exactly the same way, with the same types of parametersZ80 EXPLORER:ALU:.

Instruction Decoding and Control Logic

In order to do its job, the Z80 has to take instructions in and carry them out. If you wondered previously how it gets from ADD A,0x12 to actually doing that, you've already seen part of the answer.

The Z80 doesn't know anything about ADD A, 0x12. Add A is a mnemonic for humans. In fact, when Zilog first knocked the Z80's instruction set off from Intel's 8080, the first thing they did was *change* all the mnemonics to escape Intel's copyright. The two CPUs remained compatible, however, because the *opcodes* didn't change.

You've already seen some opcodes. You've built some, if you followed along as we talked about the ALU and its instructions. There are lots more, and no, I'm not going to get into them all. They're all in the Z80's mostly excellent datasheet, although as we've seen some construction is required for a lot of them.

Opcodes are what the CPU *does* understand. It may not know anything about ADD A,0x12 but put 0xC6 in a location in memory and 0x12 in the next location down, and the Z80 knows exactly what to do.

Unless told otherwise, the Z80 assumes anything it comes across in memory is an instruction. So when it hits 0xC6, the instruction decoder looks up that number, reads in the next value for data (0x12). The control logic connects the accumulator to the ALU, and passes 0x12 on the internal bus to the ALU, then triggers the ALU's add function. That's how it happens. It sounds like calling functions from what amounts to a menu because it *is* very much like calling functions from a menu. CPUs are like programs, as I said.

Putting It All Together: Operations

Let's walk through an entire instruction cycle on the Z80, soup to nuts, so we know how the thing works. This is discussed at some length in the datasheet, so some terms need defining.

A clock cycle or T (time) cycle is one full cycle of the clock, that is, one positive pulse, and one zero pulse.

A machine cycle is a section of the instruction cycle, a given phase of the work to be done. Each machine cycle can take from three to six clock (T) cycles.

An instruction cycle is the entire process from the time an instruction is read in as an opcode until the time the CPU is ready for the next opcode.

We'll look at an LD A, load the Accumulator, with a value of 0x10.

LD r,n is one of those opcodes like ADD A,r that we have to construct. We look up the opcode skeleton and find it's 0b00___110. We look up the Accumulator's object code, and it's 0b111 Put the two together and we have 0b00111110, or 0x3E. So our instruction and data, one after the other, will be 0x3E, 0x10.

The first step of an instruction cycle is to read the the opcode from memory. The CPU puts the Program Counter's value onto the memory bus, pulls /MREQ low, pulls /RD low, and pulls /M1 low.

/MREQ and /RD tell the memory system it's needed, that there is a valid address on its address bus, and that it will be read rather than written to. /M1 tells the system this is an instruction fetch, machine cycle 1, the start of an instruction cycle. The CPU stays in this state for three clock (T) cycles while it waits for the value on its data lines to settle. The opcode is read during the third clock (T) cycle. The CPU has taken up 0x3E. It will spend clock cycles 4, 5, and 6 decoding the instruction and, presumably, the control logic will spend that time moving data on the internal bus and connecting the right sections of the CPU together to actually carry out the instruction.

Meanwhile, for the rest of the M1 cycle, /M1 is high, /RD is high, and /MREQ goes high briefly (for one clock cycle) then goes low again, along with the /RFSH signal, to tell memory that not only are we not using it presently, but to go refresh the row at the bottom of the memory address bus. If needed. Be thankful we're not dealing with DRAM in this project.

Note that the M1 cycle, like all machine cycles, can be stretched if the /WAIT signal is low. This is so that slow memory (like what we'll be using) has time to do its job before the CPU starts asking for another address.

The LD r,n instruction is listed as requiring two M cycles. One to fetch the instruction, one to read the data from memory, so we have another M cycle to go through. Since we're reading data, this will be a memory read cycle.

The control logic is set by the decoded instruction, 0x3E, to increment the Program Counter (PC) so the next value can be read as data and *not* as an opcode later. As with the opcode fetch cycle, the memory read cycle begins with the memory address being set on the address bus, /MREQ going low, and /RD going low. /M1 does not go low, since this is the M2 cycle.

/MREQ and /RD stay low for the second half of the first clock cycle, all the way through the second, and halfway through the third. The data from the memory bus is actually read roughly at the start of the third clock cycle. Like the opcode fetch M1 cycle, the memory read cycle can be lengthened by pulling the /WAIT signal low.

The CPU now has the data it needs. The /MREQ and /RD signals go high, and no refresh activity happens.

The control logic has set the multiplexor so it's looking at the correct register (A, the accumulator) and it has data on the internal data bus to put there, so the value 0x10 is set in the accumulator.

And we're done.

LD A,n is a pretty simple instruction. There can be a lot of other M cycles involved: interrupt cycles, bus arbitration cycles, resets, and so on, but you can see from this example how things are broken down, how the opcode gets read and executed, and what happens behind the curtain.

Objects and Classes, Revisited

Let's pause a moment and pull back from the microscopic detail level we've been looking at in the guts of the Z80 and talk about objects and classes. We haven't done much object oriented C++ programming in this book, and my personal opinion is that when it's not needed, it adds complexity without a purpose.

Our sketches are starting to get complicated enough that we have a reason to use objects in this one. Even in the Dice Device, last chapter, you might have noticed that there were an awful lot of global variables, and at times it made the sketch messy and hard to read, and it sometimes got difficult to tell what code exactly accessed a given global variable.

Objects, by contrast, group the code with the variables. If a variable (or a function) is set private, it cannot be touched from outside the object. Only the object's member functions can touch it.

If a variable or a function is set public, then it *can* be touched from outside the object.

An object also has a constructor and a destructor. If we don't put one of each in explicitly, C++ will put a generic one in for us. Most of the time, that's fine. These are listed as member functions of the same name as the object for the constructor, and the same name again with a tilde (~) in front of it, for the destructor.

Here's an example:

```
class cpu {
  private:
    uint8_t a = 0;
    const uint8_t M1_MASK = 0b00010000;
  public:
    cpu() { // constructor
      Serial.println("CPU: Object created.");
      a = 0x51;
    }
    boolean M1() { //member function
      return (a & M1_MASK);
    }
   uint8_t my_uint=0;
};
```

This class doesn't do much, and it's not very useful. If it looks like I might have cut it down from a class in the sketch, you have a good eye.

This class is named cpu. There are two private variables: a uint8_t (unsigned eight bit integer) called a, and a constant uint8_t called M1_MASK, both of which are set as soon as they're declared. But watch out, there's a catch there that I'll cover very shortly.

Both those variables are private. The rest of the code can't touch them, or even see them. If you try, your sketch won't compile.

However, because the M1() function is a member of the class, it *can* touch those variables, AND them together, and return the result.

You should also notice the constructor. It throws a Serial.println() message and resets a to 51.

When exactly does that occur?

Good question. Declaring a class, by itself, does *nothing*. It defines only what an object of that class would have in it, if it existed. To actually *make* an object of that class (it's called instantiating that object), we have to declare one, just like a variable.

Here's the declaration that makes an object of the class cpu called Z80.

```
cpu Z80; // declare our CPU object.
```

You do it just like `uint8_t a;`

The object Z80, is of the class cpu, and the moment we declared it, the constructor ran. Notice that while we called the constructor, this particular one takes no parameters, so we don't put the parameters field in the call. If our constructor took parameters, for example:

```
    cpu(String a_string) { // constructor
      Serial.println("CPU: Object created." + a_string);
    }
```

we'd be obliged to call it like this:

```
  cpu Z80("hello");
```

To call other member functions, only the public ones, you use this syntax:

```
mybool=Z80.M1();
```

This tells C++ we want to call the M1() member function of the object Z80, and we want the results in the variable my_bool. Unlike constructors and destructors, calling member functions is done with standard function call syntax.

Want to access the public variable my_uint? You do it the same way.

```
Z80.my_uint=4;
```

Want to access the private variable a?

You can't.

So now that we've covered member functions and variables, what about that destructor function? When does it get called?

In this case, never. Z80 is a global object, declared at run time, and this sketch, like all sketches, has no graceful exit. When you're done, you unplug the Cestino.

There are times we'll want to make objects go away and thus fire their destructors, but to get into those, we need to talk about pointers.

Pointers

Pointers are one of those things that confuse new programmers endlessly. We've already seen them in their simplest form while we were discussing the guts of the Z80, and that may be the easiest way to understand them. So, a little review.

I said that the HL register in the Z80 was often used as a pointer to memory. This means that HL is often set to a memory address. You almost never want to access that address itself. What you really want is what's in memory *at* that address. In assembly language, this is shown like this. (HL). (HL) refers not to the memory address stored in HL, but to the value stored at that location *in* memory. With me so far? Good.

C and C++, like most higher level languages, also have pointers, and they're incredibly useful, but the syntax for getting at them is awful.

This is a C++ pointer's declaration: `uint8_t* a_pointer;`

Instead of declaring it as a regular uint8_t variable like we normally would, we declared it as a pointer. It's declared, but doesn't point to anything yet.

Pointer declaration syntax has undergone some semantic drift between C and C++. In C, pointers are traditionally declared `uint8_t *a;` that is, a is a pointer to a uint8_t. In C++ they are traditionally declared `uint8_t* a;` It doesn't matter which you use in C++. What does matter is when you try to declare more than one pointer at a time. Each pointer must have its own dereferencing asterisk. Eg: `uint8_t *a, *b;` If you declared `uint8_t* a,b;` you'll find that b is a uint8_t and not a pointer at all. It's better and much safer to put pointer declarations on individual lines.

Cannonically, in C++, you have to use the new command before your pointer points to anything. Eg: `mem_sim = new uint8_t;` The new command allocates memory for whatever type you put after it returns the address of the memory space it's allocated for you and the result is put in your pointer variable.

Still with me?

Okay. Here's where C++ causes major headaches for everyone sooner or later. In C++, you can ask for the address of any existing variable with the & character. Here's an example:

```
uint8_t *b;
uint8_t c;
b=&c;
```

See what I've done there? I've set the pointer b to equal the address of the uint8_t variable c. This works. This is fine, except if c is inside a function and b is not, when the function terminates, the memory that used to be the variable c is automatically deallocated. If I try to access b, the results are... undefined. That's never good. Worse, sometimes it will work because nothing else has allocated the memory yet.

In C++, you use pointers when a function needs to change one of the variables passed in to it: a variable that is neither a global nor (in the Arduino's case) a register, so the function can't see the variable at all. Cannonically, functions can't do this. However, if one of the parameters of your function is a pointer, your function can dereference the pointer with a *, modify the value stored in that location of memory, and go on its merry way.

Here's an example:

```
void my_function(uint8_t* pointer_to_c) {
  ++*pointer_to_c;
}

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);

  uint8_t *b;
  uint8_t c;
  b = &c;
  c = 10;

  Serial.println(c);
  my_function(&c);
  Serial.println(c);
  my_function(b);
  Serial.println(c);

}
```

Our function takes a pointer to a uint8_t called pointer_to_c. When we call it, we ask C++ to give us whatever pointer_to_c points to (hence the astarisk at the beginning of the pointer name) and increment *that*.

The first thing that happens in setup() is that We create a pointer to a uint8_t called b. Then we created a uint8_t called c. We set b to the address of c, and set c to 10.

Next, we Serial.println c. This should give us the 10 we put in it.

Next, we call the function with the *address* of c (hence the & sign in front of c). The function runs, and we Serial.println c again. Because our function takes a pointer as a parameter, and we've passed the address that would be in that pointer directly, the function works as advertised, looks up the memory address stored in pointer_to_c, and increments that uint8_t. When we Serial.println c again, we'll find that c is 11.

Next, we call my_function again, but this time with b which, you will recall, is a pointer to c. Notice well that there is no asterisk around b in this call. We're passing the value of b, which is a memory address, into my_function(), which expects a memory address. My_function() takes that address into its own pointer variable, pointer_to_c, dereferences it with a preceding asterisk, so C++ knows to look up that memory address and add 1 to the value stored there.

Nothing stops you from adding to the address stored in b. In fact, this is fairly common practice in C and C++, and a way of life in assembly language. Be careful, though. You can wind up with pointers pointed at the wrong thing very, very easily doing pointer arithmetic. Programs that mishandle pointers will compile and run, right up until the pointer is accessed, when the program will crash, access the wrong thing, or any number of other odious failures. Make sure you know what you're doing.

In case you wondered, here are the actual results from the pointer demo code.

```
10
11
12
```

Okay. I think we're up to speed on basic pointers in C++. Here's the rub. Pointers can point to objects, too.

To declare a pointer to the class we talked about in objects and classes, we'd do this:

```
cpu* Z80 = NULL;
```

The NULL should be a big clue. We have declared a *pointer* to an object, but it does not yet point to anything. In fact, it points to NULL, which is a convenient value we can test for, and also keeps the pointer out of trouble. Z80 has not been instantiated, and the constructor has not fired.

Let's make all that happen.

```
Z80= new cpu();
```

Now there actually is an object. *Now* the constructor has fired. Now Z80 points to an object in memory that we can work with.

Now is it time to talk about destructors in classes? Yes it is.

If we want to get rid of the object cpu() and fire its destructor function, we use the command "delete". Like this:

```
delete Z80;
Z80=NULL;
```

I try to always follow a delete up with setting the pointer to NULL.

Can you guess what happens if you forget to delete the object before you set its pointer to NULL?

Nothing. The object still exists, somewhere in memory, but our pointer doesn't point to it anymore. Nor does anything else. It will sit there wasting memory until the system reboots. This is a problem for pointers to regular variables too. When you hear about memory leaks in software, frequently sloppy pointer handling is the reason.

If you delete a pointer pointing to NULL, what happens? Nothing. When I talk about setting your pointers to a safe value, this is what I'm talking about. If you happened to delete a pointer that pointed to some random location in memory, that location is getting freed, whatever was in it. Not a good idea. Again, sloppy pointer handling causes an awful lot of the software problems in the world.

We will be using pointers and classes like the ones we've just talked about in the sketch, as well as in the programs we'll run on the Z80. I'll try to keep it simple and straightforward.

Function Prototypes

Up until now, when we've used functions, we've been careful to define them before they were called. This way, the GCC compiler knows that when you call function repeat(), it returns a String object, for example.

For smaller sketches (and programs in general) our approach works fine. When functions begin to call each other, however, the programming gets complicated. When some of the functions involved are members of a C++ class, your work gets worse. We'll be doing exactly that in this sketch. We'll be creating functions thate are members of a C++ class.

Fortunately C, and by extension C++, lets you create a function prototype. This prototype looks like a function declaration, except that it has no code, only the types and variable names. Here's an example.

```
String repeat(int number, char character);
```

This is the prototype for the function repeat, which takes an integer and a character parameter and returns a String object.

The compiler assumes (correctly) that we'll declare the actual function someplace else, anywhere else, like this:

```
String repeat(int number, char character) {
  String temp = "";
  for (int c = 0; c < number; c++) {
    temp = temp + String(character);
  }
  return temp;
}
```

Notice that the first line of the function has exactly the same variables, types, and variable order as the function prototype. If they don't, the sketch won't compile.

Function prototypes are a good idea generally. They completely free you from sorting out what functions call what and what order they're declared in. In fact, classic C programming usually has the prototypes first, and the functions *after* main() (the equivalent of loop() more or less,) after all the other code. It's such a good idea that up until Arduino 1.6.6, the Arduino IDE built your function prototypes for you. They reasoned that it can be a little tricky for new programmers to make sure that a function's prototype and its declaration are always the same.

For reasons known only to themselves, the Arduino maintainers broke this functionality in Arduino 1.6.6 and above, so moving forward, as protection from IDE chaos, it's probably a good idea to make your own.

Build the Z80 Explorer

For all the complexity of the sketch, the build is pretty straightforward. As you can see in Figure 11-2, we're using all four ports of the ATmega. Ports A and B are the address bus's least significant bits (LSB) and most significant bits (MSB). Port C is the data port. These can be changed around fairly easily in the sketch if you want to. The port that absolutely must remain the same is port D, which is wired to 6 of the 14 signals in the Z80's control bus. It's essential that this port be wired as shown, because we're using OC1A (PD5, aka pin 19) to generate the Z80's clock, and we're using INT1 (PD3, aka pin 17) and INT0 (PD2, aka pin 16) as interrupt signals for the ATmega to process the Z80's memory requests.

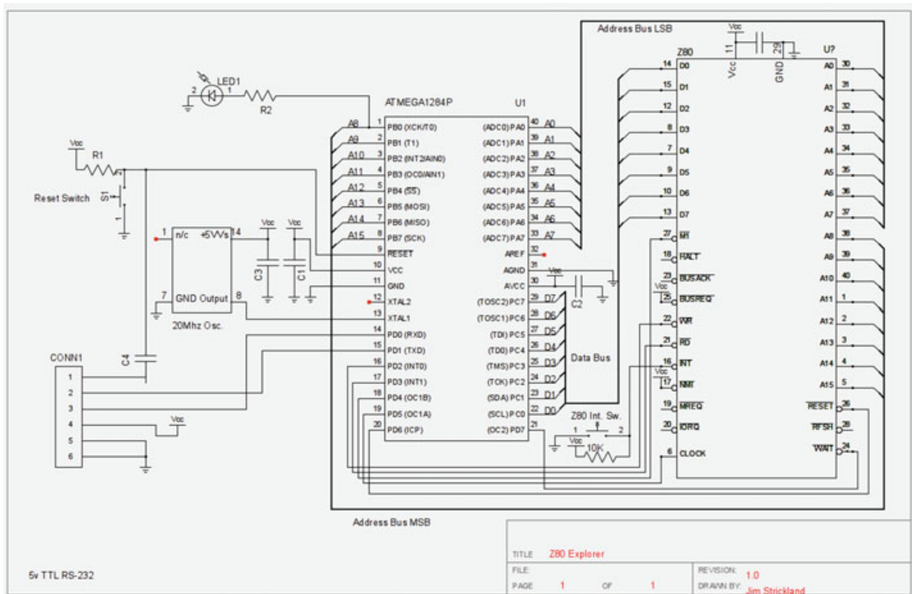

Figure 11-2. The Z80 Explorer Schematic

Install the Z80

Like the ATmega1284P, the Z80 is a 40 pin DIP IC, so I recommend installing it starting with its pin 1 on line 11 of your breadboard, oriented the same way as the ATmega. It makes for a neater looking breadboard and keeps the connecting wires shorter. For high speed signals, shorter is better.

Power Circuits

First, in case you haven't been doing this all along, unplug the Cestino from USB power.

Wire the Z80's pin 11 (the pins on the schematic in Figure 11-2 are *not* in order) and pin 29 to the + and - busses, respectively, then wire the 0.1µF capacitor from pin 11 to the - bus. Nothing you haven't seen here before. We're giving the Z80 a bypass cap of its own to keep noise out of its power supply.

Data Bus

Let's go ahead and wire up the data bus first. Why? Because all the data bus pins are on the left side of the Z80 as we've mounted it, and they all connect to port C, which is on the bottom right of the ATmega. This means that our data bus wires will be under many of the control bus wires, so its easiest to do them now.

The data bus pins are, sadly, all over the left side of the Z80 and not in any particular order. In *signal* order, that is, D0-D7, they are Z80 pins 14, 15, 12, 8, 7, 9, 10, 13. Wire these pins to ATmega port C, from PC0 (ATmega pin 22) to PC7 (ATmega pin 29) in that order. As usual, I hanked these wires up once I had them in place.

Control Bus

The Z80 has a lot of control signals. Of these, we use six, so let's start by connecting the input signals we're not using to the + bus to keep them out of trouble. Connect pins 17 and 25 on the Z80 to the + bus. These are /NMI, nonmaskable interrupt (Z80 pin 17), and /BUSRQ which requests the Z80 stop talking to the memory bus so other devices can talk to it. (Z80 pin 25).

The Clock

Ordinarily, the Z80's clock would be like the ATmega's, driven by an external oscillator. For this project, however, we'll be using an ATmega timer to generate the clock signal for the Z80, so we can control the Z80's processing speed from the sketch. We dealt with timers in chapter ten, but this time we'll use the timer's output pin to generate the Z80's clock directly, instead of interrupting the ATmega. Theoretically, this gives us up to a 20MHz clock, equal to the Cestino's own system clock (but isolated from it by the ATmega's internal electronics.). Realistically we won't be running the Z80's clock anywhere near that fast, even if you do have the 20MHz version of the Z80.

Connect pin 6 on the Z80, /CLK, to the OC1A pin on the ATmega's pin 19.

The Interrupt Button

One of the things I wanted to show with the Z80 explorer is how interrupts really work, what really happens, and what precisely an interrupt vector is. It can be a tough concept, but once you see it happen, it's completely clear. To do that, we need to be able to generate an interrupt on the Z80.

The Z80 has two interrupt lines, really, the /INT signal, and the /NMI signal. We pinned /NMI to the + bus to keep it from ever being activated. /NMI, as the name Non-Maskable Interrupt suggests, can't be turned off. If we have a button problem or an accidental push of the interrupt button during a program we don't want interrupts in, /NMI can send the Z80 off on a non-existent interrupt vector that goes nowhere, and from which we can't recover.

/INT, by contrast, is much better behaved. We'll wire it to a tactile button.

Once you get the button in your breadboard, wire one side of its switch to the - bus, and wire the 10kΩ resistor from the other side to the + bus. By now you know that we can safely short the + bus to the - bus through a 10kΩ resistor.

The 10kΩ resistor is a pullup for the interrupt circuit, so wire from the junction of the switch and the 10kΩ resistor to pin 16 of the Z80, the /INT pin. When you push the button, it will lower the voltage on the /INT circuit to zero, and since /INT is active low, this will send the Z80 an interrupt signal. Until we turn interrupts on, the button won't do anything, and that's fine.

Memory Read/Write Signals

We'll be using the ATmega to simulate RAM for the Z80. This will be slow. In order to make it as fast as possible, we'll do it with a pair of ISRs. This is why /RD and /WR need to be tied to two of our three INT pins on the ATmega.

Connect the /RD signal on the Z80 to INT1 on the ATmega (Z80 pin 21 to ATmega pin 17). Connect the Z80's /WR (pin 22) to the ATmega's INT0 (pin 16).

Reset and Wait Signals

Our simulated memory will be slow, as I said. To make sure the Z80 stays in sync with such a slow memory, we'll need the /WAIT line. /WAIT is on Z80 pin 24. Connect it to PD7 (ATmega pin 21, over on the other side of the ATmega from the rest of port D.)

We'll also need to be able to reset the Z80 (a lot). We could wire the Z80 to the ATmega's reset circuit, but this would mean we'd have to restart the sketch on the ATmega every time we wanted the Z80 to reset, and that the Z80 might well get going, asking for memory, before the sketch is ready for it. Instead, we'll wire the /RESET signal on the Z80 (pin 26) to PD6 on the ATmega (pin 20).

The Z80's reset must be held down for at least three clock cycles to ensure a reset. The datasheet doesn't specify why.

When you're done with the control bus signals, go ahead and hank those wires up, too.

Memory Address Bus

One more bus to go. This, by now, is standard stuff for us. Divide the 16 bit memory address bus into an 8 bit LSB and MSB, and wire each of those into the corresponding pins of an ATmega port.

Starting with the Least Significant Bits, wire A0-A7 on the Z80 (pins 30-37 in order) to PA0-PA7 on the ATmega (pins 40-33, in that order). It's nice when pins and their signals come in order like this. It doesn't happen most of the time.

For the MSB, we'll need pins A8-15 on the Z80. These are, in order, 38, 39, 40, 1, 2, 3, 4, and 5. Wire these to PB0-PB7, pins 1-8 in that order, on the ATmega.

That's it. That's all the wiring there is. I suggest hanking the memory address bus lines in two groups, one for each port on the ATmega. That's how I did it.

The Sketch

If you drink coffee, now would be a good time. This sketch is a big one, with a lot going on. It's got a timer with an output, two interrupts, three ISRs (although we only use one or two at a time), three classes, a handful of utility functions, and three, count 'em three global variables, two of which are pointers. It's not really as bad as it sounds. Let's dive in.

The Plan

This sketch has nine menu options divided into three groups.

We can reset the Z80, set the clock speed, and stop the clock in the Z80 Commands group.

We can hook the Free Run ISR up, or disconnect it in the Free Run ISR Commands Group.

The Memory Simulation and Z80 Programming group is the big one, where we can set up simulated memory, enter programs into it, run programs by attaching the mem_sim_ISRs, and dump the simulated memory to the Serial Monitor.

Clock Setting

Clock setting will configure timer/counter 1 to generate a specific, square-wave frequency and output it to OC1A without further intervention from the ATmega. Because the Z80 will begin running right then, and won't stop as long as the clock is running, we'll also provide a way to turn the clock off.

Free Running

Free running is an easy way to test a microprocessor to see if its most basic functionality is working: is it trying to read instructions. To free run, all you really need are the microprocessor, a clock, and some wire, although to see what's going on requires a bit more.

From our discussions in Z80 internals, you know that the Z80, once it restarts, goes into an M1 instruction fetch cycle. It pulls /MREQ, /RD and /M1 down, puts an address on the address bus, and reads whatever is on the data bus. Free-running essentially puts a fixed value on the data bus, an instruction called NOP, whose opcode is, conveniently, 0x00.

A NOP causes the Z80 to do nothing at all. The Z80 fetches the instruction, increments the PC (Program Counter), and goes into its next fetch.

If we wire the data bus so that all the data lines are grounded, every time the Z80 does a read, it gets a NOP, and advances, advances the program counter, and tries again. It will carry on until it hits the end of memory, and when it overflows, it will go back to zero and start again.

It's a great way to make sure your memory bus, data bus, and CPU are working.

You can do this without a sketch at all, except to generate the clock, but you'd have to wire a bunch of LEDs up to watch the data bus.

What we'll do is generate the clock, connect an ISR to the /RD line (as instruction fetches are all memory reads). That ISR will always return 0x00 to the data lines, and it will output the memory address requested to the serial monitor.

We'll also provide a way to disconnect the free_run_ISR without stopping the clock.

Memory Simulator

Memory simulation works pretty much the same way as free running, except that we'll use an 8k array to simulate memory, and an ISR connected to the ATmega's interrupts where the /RD and /WR signals come in.

Memory Reads

When the /RD signal goes active (low), INT1 will fire, and the mem_read_ISR will get called.

It will get the value on the memory bus by combining the inputs of port A and port B, then go to the cell of the memory simulator array corresponding to the address from the Z80, and return the contents of that cell to the Z80 on the data line.

The mem_read_ISR will also print what address was requested, what data was returned and, after reading the status of the /M1 signal, whether we're in an instruction fetch or a data read cycle.

Memory Writes

When the /WR signal goes active (low) INT 0 will fire, and the mem_write_ISR will be called.

It will get the value on the Z80's memory address bus by combining the input values of port a and port b, then go to that address in the memory simulator array. It will then copy data from the Z80's data bus to the memory simulator array, then print the address requested and what data was written there on the serial monitor.

Editor

The big advantage of having simulated memory is that the ATmega can control what's *in* memory. Since the Z80 will run happily with only 8k of RAM (or less) we'll be writing a simple way to input bytes to the memory array and review those contents. Is it a slow way to write programs? Yes. Is it better than toggling them in with toggle switches on the address and data lines and reading them back on arrays of LEDs the way they did in the mid 1970s? Much.

Dumper

It'd be nice to be able to see what our programs look like in memory, and more importantly, what's in memory after they've run. While this dumping is included in the editor, we'll also connect the same function up so we can call it outside the editor.

The Code

So that's the plan. Let's get started. As usual, we begin with preprocessor defines.

Preprocessor Defines

```
#define MAX_CLOCK 20000000
#define MEM_SIZE 0x2000 // 8192 Bytes
#define RESET_MS 5000 // Z80 reset: hold /RESET low this long.
```

MAX_CLOCK is the maximum speed the a timer on our ATmega can be set to. This is always equal to the clock feeding the ATmega, which in this case is 20MHz. This means you'll be able to generate a clock signal up to 20MHz. If you have an 8MHz or 4MHz Z80 on the end of that, bad things may happen to it.

MEM_SIZE is 0x2000, or 8192 Bytes. That's 8k. That's about 600 dollars worth (U.S.) in 1980. It's half the ATmega's available RAM. With careful tweaking, you can probably make this bigger, but none of the example programs use more than a tiny fraction of this space. Setting it to 0xFF would probably work, but I haven't tried it.

RESET_MS is the number of milliseconds to hold /RESET low. I have this set for 5000, or five seconds. At the slowest clock speed available, 1Hz, this is more than enough. 3000 would probably work, but if your reset isn't long enough your results can be very odd.

Function Prototypes

We talked about function prototypes already. Here are the prototypes for the utility functions, which we'll discuss at length when we get done with all the classes.

```
String repeat(int number, char character);
String get_input_string();
uint32_t string2uint32_t(String input);
uint16_t hex_string2uint16_t(String input);
void menu();
```

Classes

The next part of the sketch is the declaration of the three classes we use in this sketch. As I'll repeat several times, since we haven't used objects much, remember that no object of a given class exists until one is instantiated, and that isn't part of the class declaration. Okay? Okay.

cpu

The CPU class controls PORTD and PIND, the output and input registers of port d. It uses this to set and clear various control signals on the Z80, and to read the status of others. The only control signal it does not control is /CLK, the clock.

The CPU class also provides access to read the Z80's address bus, and to read or set the Z80's data bus. Remember that no objects of this type will exist until we declare one, which is much further down in the sketch. We'll start by declaring the class.

```
class cpu {
```

Private Variables and Functions

The first thing in the class are the private variables and functions. They don't have to be first. That's just where I like to put them. The unsigned 8 bit integer saved_control_port is used to store the output setting of the control port (port D) on the ATmega. Could we just look at it directly, since it's a global register? Yes, but if we're going to use an object to control the CPU, then we should *always* use the object. So we need a place to store our current ATmega output status so we can restore it later.

```
  private:
    uint8_t saved_control_port = CTRL_DEFAULT;
```

Since we're using this object and only this object to talk to the CPU, it didn't make a lot of sense to use a lot of global defines, so instead I set these masks as constant uint8_ts. The first one's a mask, to be ANDed with CONTROL_PINS (an alias for PIND we'll set later) to return the /M1 signal's status. The other three are values that can be set on

CONTROL_PORT, the alias for PORTD. CTRL_DEFAULT holds /WAIT and /RESET high, and sets everything else low. Since most of the other signals are ATmega inputs (Z80 outputs), this matters not at all. It also ensures that all the ATmega inputs' pullup resistors aren't set. I'm kind of sloppy with them with the other states, but since we're only in those states briefly and we don't read any other signals in them, it doesn't cause problems.

```
const uint8_t M1_MASK = 0b00010000; //Read the /M1 signal
const uint8_t RESET = 0b10111111; // Set the /RESET signal
const uint8_t WAIT = 0b01111111; // Set the /WAIT signal
const uint8_t CTRL_DEFAULT = 0b11000000; // default Z80 state
```

Next are variables that look like pointers and work kind of like pointers, but aren't really pointers. These volatile uint8_t&s are reference variables. They are literally aliases, in this case for PORTD, PIND, PORTC, DDRC, and PINC. They could have been done with global #defines, but I chose not to. Note that these reference variables are private.

The prefix *volatile*, you'll recall from chapter 10, is necessary any time a variable is accessed outside the current flow of code, like when an ISR access them.

```
volatile uint8_t& CONTROL_PORT = PORTD;
volatile uint8_t& CONTROL_PINS = PIND;
volatile uint8_t& DATA_PORT = PORTC;
volatile uint8_t& DATA_DDR = DDRC;
volatile uint8_t& DATA_PINS = PINC;
```

Public Variables and Functions

Next come the public variables and member functions. Note the constructor, that gets fired when the CPU class is instantiated. It sets up all the ports this class uses. Note that I've used the real port registers. My reasoning for this is that for DDRA, DDRB, and DDRD, they are only set here, only accessed in this one function, and so have no aliases set. Using the alias in this particular function seemed confusing. Once the DDRs are set, we do use the aliases to set CONTROL_PORT and DATA_PORT to their default values. That's all this constructor does.

```
  public:
    cpu() { // constructor
      Serial.println("CPU: Object created.");
      DDRA = 0b00000000; // Address LSB
      DDRB = 0b00000000; // Address MSB
      DDRC = 0b11111111; // Data Port;
      DDRD = 0b11100000; // control_port
      DATA_PORT = 0b00000000;
      CONTROL_PORT = CTRL_DEFAULT;
    }
```

The first member function of the cpu class is M1(). It returns the status of the /M1 flag as a boolean. Like the flag, what it returns is active LOW.

```
boolean M1() {
  return (CONTROL_PINS & M1_MASK);
}
```

The mode member functions, mode_default(), mode_wait(), save_mode(), restore_mode() either change the output settings of CONTROL_PORT, aka PORTD, to change the operating mode of the Z80, or save and/or restore the current status of CONTROL_PORT. Note that mode_save() and mode_restore() do no sanity checking whatsoever, so use with caution.

The mode_default() function sets the Z80 in normal mode, ready to run a program, and mode_wait() pulls the /WAIT line low, making it stay in the current instruction cycle. Should reset() (below) have been mode_reset()? Probably.

```
void mode_default() {
  CONTROL_PORT = CTRL_DEFAULT;
}
void mode_wait() {
  CONTROL_PORT = CONTROL_PORT & WAIT;
}
void save_mode() {
  saved_control_port = CONTROL_PORT;
}
void restore_mode() {
  CONTROL_PORT = saved_control_port;
}
```

Next, we set a public pair of reference variables to point to PINA and PINB, where our addresses will be visible. We use this functionality so often it didn't seem prudent to abstract it in a function.

```
volatile uint8_t& addr_msb = PINB;
volatile uint8_t& addr_lsb = PINA;
```

The data_out() and data_in() functions set the DDR of the data port (DATA_DDR) to either input or output so the Z80 can send or receive data from the ATmega, and then either returns the value of the data bus (data_out()) or sets the data port to the value of the data passed to the function (data_in()).

These functions are a little confusingly named. They're all about setting the ATmega's ports, but they're named (since this is the cpu class) from the Z80's perspective. So when the Z80 is outputting (data_out()), the ATmega is inputting, and vice versa.

```
uint8_t data_out() {
  DATA_DDR = 0b00000000;
  return DATA_PINS;
}
```

```
    void data_in(uint8_t data) {
      DATA_DDR = 0b11111111;
      DATA_PORT = data;
    }
```

Here's the reset function. You call it, and the Z80's reset line gets held low for RESET_MS milliseconds. It also helpfully sends a message to the serial monitor.

```
    void reset(void) {
      Serial.println("CPU: Resetting...");
      CONTROL_PORT = CONTROL_PORT & RESET;
      delay(RESET_MS);
      Serial.println("CPU: Done Resetting.");
      CONTROL_PORT = CTRL_DEFAULT;
    }

Here's the end of the cpu class.
};
```

clock_gen

Next up is the clock_gen class. It's a considerably simpler animal. It has only two member functions, a constructor and a destructor. The constructor takes a 32 bit unsigned integer parameter and uses it to compute the settings for the clock frequency the user requests. The destructor turns the clock off.

Once again, we're controlling global registers by wrapping a class around them. Not strictly necessary, but it sure simplifies the rest of the sketch.

Private Variables

We start with a couple private arrays, prescale_values and prescale_bits, which are a lookup table. You look up the value you want (1024, 256, 64, 8, or 1) and use the index of the array you found it at to look up the bit setting to set timer/counter 1's clock source bits to that prescaler.

```
class clock_gen {
  private:
    int prescale_values[5] = {1024, 256, 64, 8, 1};
    int prescale_bits[5] = {0b101, 0b100, 0b011, 0b010, 0b001};
```

Public Variables and Functions

The first public function of this class is the destructor, ~clock_gen(). It's fired when we delete an object of the clock_gen class. It sets timer/counter 1's registers to zero, stopping the clock generator completely and disconnecting OCP1 from it. It also prints a helpful message to the serial monitor.

```
  public:
    // Destructor
    ~clock_gen() {
      TCCR1A = 0;
      TCCR1B = 0;
      OCR1A = 0;
      TCNT1 = 0;
      Serial.println("Clock Generator: Object Deleted.");
    }
```

Next is the constructor, clock_gen(uint32_t desired_frequency). Given the desired frequency (in decimal Hz, always, so 2MHz is 2000000), it computes the best fit between prescaler and counter match value to get as close as possible to the frequency requested. Sometimes that's not very close. The Cestino's clock runs at 20MHz. That means if 20,000,000 doesn't divide evenly by your desired frequency, you may be over or under a little bit. Fortunately, we don't need absolute precision.

First, we Serial.println() a message to the serial monitor, then declare and initialize (in most cases) a fistful of variables. Notably, we set counter_value, a float, to the value of MAX_CLOCK/desired_frequency. If the desired frequency does not divide evenly into the Cestino's clock (20MHz, normally) this will have a decimal component, and our resulting clock will not be exactly on the frequency.

```
    // Constructor
    clock_gen(uint32_t desired_frequency) {
      Serial.println("Clock: Object Created.");
      float counter_value = MAX_CLOCK / desired_frequency;
      float lowest_inaccuracy = 1.0;
      float current_steps = 0;
      int prescaler = 1;
      byte prescaler_config_bits;
      long int match;
```

Next, we step through the possible values of the prescaler (in prescale_values[]) and calculate which one produces the smallest decimal component when we divide counter_value by it. The smaller the decimal component, the closer the clock will be to the frequency we asked for.

Once we find the prescaler value, we set prescaler with that value, then set match to counter_value divided by the prescaler we chose. If match came out zero (usually because something rounded to there) we set match to counter_value.

```
      for (int c = 0; c <= 4; c++) {
        current_steps = counter_value / prescale_values[c];
        if ((current_steps - round(current_steps) < lowest_inaccuracy)\
            && (current_steps <= 65535)) {
          lowest_inaccuracy = current_steps - ((int)current_steps);
```

```
            prescaler = prescale_values[c];
            match = round(current_steps);
            prescaler_config_bits = prescale_bits[c];
            if (match == 0) match = round(counter_value);
          }
        }
```

Next, we tell the user what values we generated.

```
        Serial.print("We want to count to ");
        Serial.println(counter_value, 2);
        Serial.print("For a clock speed of ");
        Serial.println(desired_frequency, DEC);
        Serial.print("I chose a prescaler of ");
        Serial.println(prescaler, DEC);
        Serial.print("And a match of ");
        Serial.println(match, DEC);
        Serial.print("We'll count to ");
        Serial.println((long int)prescaler * match, DEC);
```

Finally, we set the clock registers, activate the timer/counter, and tell the user that we've done so.

```
        TCCR1A = 0b01000000;
        TCCR1B = (0b00001000 | prescaler_config_bits);
        TCNT1 = 0;
        OCR1A = match;
        Serial.println("Clock Generator: Running");
    }
So ends the clock_gen class.
};
```

memory_simulator

The memory simulator class isn't conceptually hard. Create an array of size MEM_SIZE, then, when given an index value for the array, either set or retrieve the value stored there. That's really all it does, except that it also has the m_dump() member function that pretty-prints the contents of the memory array in 256 byte pages, and an editor for putting values into the memory array.

```
class memory_simulator {
```

Private Variables and Functions

We start by declaring two private variables, halt and mem_array[MEM_SIZE]. Halt stores the value of a halt instruction for this particular CPU (the Z80), and mem_array[] is the array we'll use to simulate memory.

```
  private:
    volatile uint8_t halt = 0x76;
    volatile uint8_t mem_array[MEM_SIZE];
```

Next, we have some private functions for m_dump and m_edit. They Serial.println() messages when called.

```
    void dump_page_header() {
      Serial.println("\nAddress\t 0  1  2  3  4  5  6  7  8  9" +
                     String("  a  b  c  d  e  f   Data (text)"));
      Serial.println(repeat(75, '-'));
    }
    void m_edit_instructions() {
      Serial.println("\n\t*** Mem-Sim Line Editor ***");
      Serial.print("Enter an address and data in HEX ");
      Serial.println("eg: 0x0000,0x76");
      Serial.println("\"exit\" to quit\n\"dump\" to view memory\n");
    }
```

Public Variables and Functions

The public member variables are a volatile boolean called m_write_enable. If this is false, m_seek_write() will *say* it's written to memory, but it won't actually do it. This is a kludge to protect simulated memory from being written to during Z80 resets.

```
  public:
    volatile boolean m_write_enable = true;
```

The constructor of memory_simulator() zeros out the array. Some programming environments do that for you. The Arduino environment isn't one of them. That memory can contain literally anything. If you're curious, some time do a dump memory command without initializing the memory. Mine shows a lot of machine code and text strings from this sketch.

Once the array is wiped, tell the user how much memory is available in an old school "bytes free" message.

```
memory_simulator() { // constructor
  Serial.println("Memory: Initializing...");
  for (int c = 0; c < MEM_SIZE; c++) {
    mem_array[c] = 0;
  }

  Serial.println("Memory: " + String(MEM_SIZE, DEC) + \
               " (0x" + String(MEM_SIZE, HEX) + ") bytes free.");
}
```

Is there no destructor? Technically there is, but it's the one C++ assigns for us. That's fine, in this case. We're not setting any ATmega registers that need to be cleared in this class.

The next member functions are m_seek_read() and m_seek_write(). These functions, as the name suggest, grind an array index value from the two address bytes we can read from the cpu class, look that value up in the array, and either read the value or write it. Once again, to keep track of what is reading and what is writing, remember that this is from the memory_simulator class's perspective, and memory simulator is pretending to be RAM. A memory write is when the Z80 sends data to the simulator. A memory read is when the Z80 reads data from the simulator.

```
volatile uint8_t m_seek_read(volatile uint8_t address_msb, volatile
uint8_t address_lsb) {
```

You've seen me do this next bit half a dozen times now. Here, we're doing it backward. Put two uint8_ts into the union, get one uint16_t out. If you're changing code, make sure you put the bytes in in the correct order for a little endian CPU like the Z80. Does your ear tell you I might have gotten that wrong once during the development of this project? As usual, your ear is good.

```
  volatile union {
    volatile uint16_t sixteen_bit_address;
    volatile uint8_t byte_array[2];
  } addr_byte_union;
  addr_byte_union.byte_array[0] = address_lsb;
  addr_byte_union.byte_array[1] = address_msb;
```

Here, we sanity check the address that was requested. If it's too big, we go ahead and return a value anyway, but we return the value in the private variable halt. Otherwise we return the value in mem_array at the 16 bit unit address we got from the union earlier.

```
    if (addr_byte_union.sixteen_bit_address >= MEM_SIZE) {
      return halt;
    } else {
      return mem_array[addr_byte_union.sixteen_bit_address];
    }
  }
```

The member function m_seek_write() works pretty much the same way, except that it takes three parameters, adding a volatile uint8_t for data, and returns nothing. It also checks to see if m_write_enable is set, and if it's not, only pretends to do the write. Other than that it looks like the same code as m_seek_read() because it is the same code as m_seek_read(). If a write is requested to an invalid address, m_seek_write() silently fails.

```
  void m_seek_write(volatile uint8_t address_msb,
  \volatile uint8_t address_lsb, volatile uint8_t data) {
    if (m_write_enable) {
      volatile union {
        volatile uint16_t sixteen_bit_address;
        volatile uint8_t byte_array[2];
      } addr_byte_union;
      addr_byte_union.byte_array[0] = address_lsb;
      addr_byte_union.byte_array[1] = address_msb;
      if (addr_byte_union.sixteen_bit_address >= MEM_SIZE) {
      } else {
        mem_array[addr_byte_union.sixteen_bit_address] = data;
      }
    }
  }
```

The next member function, and one of the two really enormous functions in this class, is m_dump. If it looks a lot like the pretty printer from chapter 9, ATA Explorer? That's because it's a modified version of that same code. You've seen it before, and you know how it works, so my comments will be sparse on this function.

```
  void m_dump() {
    Serial.println("Dumping Simulated Memory");
    int c = 0;
    String line_start_address = "0x";
    uint16_t row_address = 0;
    String hex_data = "";
    String human_readable_data = "";
```

The pretty printer in chapter 9 did not have a header. This one does. Here's where we call the private function dump_page_header(). After that, we start the main loop, one for every row of 16 addresses. With 8192 addresses total, that would be 512 rows.

For each row we generate the start address of the row (something else the pretty-printer in chapter 9 didn't do) then loop through 16 cells of the memory array (instead of 32 as we did in Chapter 9), all the while building a pair of output strings, hex_data and human_readable_data. When we finish a row, we combine the two strings, plus our third string with the start address of the row, and Serial print them.

512 rows is far too much to output in one gulp, so there's a simple paging mechanism that's also new. If c, which is incremented for every cell of the array % 256=0 (c mod 256 =0 for 256 bytes per page) we stop, ask the user if they want to continue, and if they do re-print the header and carry on. If not, we set row to MEM_SIZE/16, which is its exit value, and when the main loop comes around again, it exits.

```
dump_page_header();
for (int row = 0; row < (MEM_SIZE / 16); row++) {
  row_address = 16 * row;
  if (row_address < 0x1000)line_start_address += "0";
  if (row_address < 0x0100)line_start_address += "0";
  if (row_address < 0x0010)line_start_address += "0";
  line_start_address += String(row_address, HEX) + "|";

  for (int col = 0; col < 16; col++) {
    if (mem_array[c] < 0x10) hex_data += "0";

    hex_data += String(mem_array[c], HEX);
    hex_data += " ";

    if (isprint(mem_array[c])) {
      human_readable_data += (char)mem_array[c];
    } else {
      human_readable_data += ".";
    }

    c++;
  }
  Serial.println(line_start_address + " " + hex_data + \
                " | " + human_readable_data);
  hex_data = "";
  human_readable_data = "";
  line_start_address = "0x";
```

```
      if (!(c % 256)) {
        Serial.println("Continue (y/n)");
        if (get_input_string() == "n") {
          row = MEM_SIZE / 16;
        } else {
          dump_page_header();
        }
      }
    }
  }
```

The m_edit() member function, the last one in this class (remember, we're in the memory_simulator class) is another fairly simple function. It takes a text string, processes into either a pair of hex values or y (to continue) or dump (to display memory). It does not keep track of what page of memory you're on and intelligently dump that page, nor can it save data you input anywhere but in simulated memory. By the time I was done writing the sample programs for the Z80, I sorely wished it did both, but it gets the job done. This function makes extensive use of the String object in Arduino, which isn't particularly well documented, and is certainly not part of standard C++.

The function starts out initializing input_string to "y" which is probably unnecessary. It has a 16 bit unsigned integer for an address and an eight bit unsigned integer for data. It also has an integer called comma_index to store the location of the comma in the pair of hexadecimal numbers. Finally, it declares a String object called temp.

```
  void m_edit() {
    String input_string = "y";
    uint16_t address;
    uint8_t  data;
    int comma_index = 0;
    String temp;
```

Next, m_edit calls the m_edit_instructions() private function to print its instructions. As with m_dump this happens more than once in m_edit, so it made sense to use a private function for it. Once that's done we start a do loop. I haven't used those much. Essentially they're a while loop upside down, so the action is always done at least once. Then we call the external function get_input_string(). This is a function we'll cover in the Utility Function section. It is a combination of two Serial object functions: it waits for serial data to become available (forever if need be) and reads that String in and returns it.

```
    m_edit_instructions();
    do {// repeat loop until the 'while' is satisfied.
      input_string = get_input_string();
```

If the input string is “dump” we call m_dump(). When we return from m_dump(), call m_edit_instructions() again.

```
if (input_string == "dump") {
  m_dump();
  m_edit_instructions();
}
```

If input_string starts with the character “0x”—a proper hex value for the address—then it’s okay to try and find the comma in the string, and set temp to the first half of the string, from the beginning to the comma. Once we call temp.trim(), another String object member function, to remove any excess spaces, we can call the external function “hex_string2uint16_t()” with the string. This, too, is in the Utility Functions section. Set address to the result.

```
if (input_string.startsWith("0x", 0)) {
  comma_index = input_string.indexOf(",");

  temp = input_string.substring(0, comma_index);
  temp.trim();

  address = hex_string2uint16_t(temp);
```

Having found our address, we set temp to the substring function of the String object input_string, starting at comma_index+1 and going to the end. (We don’t have to tell it to go to the end. The fact that we haven’t included an end value makes it do that by default.) Call temp.trim() again, and feed the result to hex_string2uint16_t again, cast the result to a uint8_t, and store that in data. Data will always be an eight bit byte, but hex_string2uint16_t builds eight bit bytes on its way to building sixteen bit bytes, so the results will be correct.

Once that’s done, we tell the user what data we got and where we’re going to put it in simulated memory. Then we do it.

```
  temp = input_string.substring(comma_index + 1);
  temp.trim();
  data = (uint8_t)hex_string2uint16_t(temp);

  Serial.println(">  Addr: 0x" + String(address, HEX) + \
               " Data: 0x" + String(data, HEX) + " [OK]");

  mem_array[address] = data;
}
```

Finally, we see if the user typed "exit". If not, the do while loop test passes, and we go back to the beginning of the loop.

```
      } while (input_string != "exit");
    }
};
```

That's the end of the memory_simulator class, and all the classes in this sketch. Remember that while we've declared three classes, no objects of these classes exist, and we can't use their variables or their member functions until one does. I know. I'm repeating it. It bears repeating.

Fortunately, that happens next.

Global Variables

There are three global objects (variables are a class of object in C++) in this sketch. Two of them are ourclock, a pointer to a clock_gen object, and mem_sim, a pointer to a memory_simulator object. These objects *still* won't be instantiated yet.

There is also Z80, which is declared as a cpu object. This object is instantiated right there, the constructor fires, and we can use the functions in it. It instantiates here because we never actually turn it off.

```
clock_gen* ourclock = NULL;
memory_simulator* mem_sim = NULL;
cpu Z80;
```

Utility Functions

The utility functions are part of no class, but they may be called from inside classes, other utility functions, setup() or loop(). They're mostly tiny utilities, save one, menu(), which is quite large.

repeat()

We need to make a lot of lines of dashes and whatnot in the menu for this sketch. Repeat makes that much simpler. You pass it a single character in a char variable, give it a number, and it returns a String object with that character repeated that many times.

```
String repeat(int number, char character) {
  String temp = "";
  for (int c = 0; c < number; c++) {
    temp = temp + String(character);
  }
  return temp;()
}
```

get_input_string()

This sketch does a lot of input. Rather than implement the usual wait-forever-for-serial loop followed by a Serial-readString() over and over again, I put them in a function. This function takes no parameters and returns a String object with the string the user typed. If the user never sends any data through the serial monitor, this function will happily loop forever.

```
String get_input_string() {
  while (!Serial.available()) {
  }
  return (Serial.readString());
}
```

string2uint32_t()

This function may look very familiar. It's copied from the sketch in Chapter 9. It takes a String object with a number typed in decimal in it and turns it into a uint32_t, a 32 bit unsigned integer. String objects actually have this functionality, it turns out, but it's practically undocumented, and there's no indication how large an integer their function will handle. Rather than poke through the Arduino core code, we'll use the same function we used in Chapter 9.

This function processes the string from left to right. For each digit it encounters, it multiplies the uint32_t temp by ten, then computes the numeric value of the digit by subtracting the character number of zero from it, and adds the result to temp. When it reaches the end of the string, it returns temp.

```
uint32_t string2uint32_t(String input) {
  uint32_t temp = 0;
  input.trim();
  for (int c = 0; c < input.length(); c++) {
    temp = temp * 10 + input.charAt(c) - '0';
  }
  return temp;()
}
```

hex_string2uint16_t()

Not only does this sketch need to process strings into decimal values, it needs to process strings into hex values. Whereas strin2uint32_t is a classic algorithm, hex_string2uint16_t is a somewhat more complicated, modified version of that algorithm that I put together myself.

We start out with temp=0, just as in the last function, and at the beginning of the loop, we multiply temp by 16.

```
uint16_t hex_string2uint16_t(String input) {
  uint16_t temp = 0;
  char tempchar;
  input.trim();
  for (int c = 2; c < input.length(); c++) {
    temp = (temp * 0x10);
```

Here's where things get complicated. Alphabetic characters are not as neatly arranged in numeric codes as numbers are, so we have to tinker more. We set tempchar to the character at position c of the String object input. If the character is lower on the ASCII chart than ":", it's a number from zero to nine. Add the character code minus the character code for zero to temp just as in the last function.

```
    tempchar = input.charAt(c);
    if (tempchar < ':') temp += (tempchar - '0');
```

If tempchar is between "@" and "G", it's a capital letter between A and F. Subtract the character code for "7" from the character code in tempchar and add the result to temp.

```
    if ((tempchar > '@') && (tempchar < 'G')) temp += (tempchar - '7');
```

if tempchar is between "`" and "g", it's a lower case letter between a and f. Subtract the character code of "W" from the character code value of tempchar and add the result to temp.

```
    if ((tempchar > '`') && (tempchar < 'g')) temp += (tempchar - 'W');
  }
```

Loop through the whole number like this. Once we're done, return the value of temp.

```
  return temp;()

}
```

menu()

The menu function works exactly the same way the menu in loop() worked in Chapter 9, so I won't cover the mechanics of the switch/case structure again. You already know. There's quite a bit going on in this menu, though.

We start out clearing input_string and displaying the menu. Note the extensive use of repeat().

```
void menu() {
  String input_string = "";
  Serial.println("\n" + repeat(15, ' ') + "*** Menu ***");
  Serial.println(repeat(42, '-'));
  Serial.println("Z80 Operations Commands");
  Serial.println(repeat(42, '-'));
  Serial.println("(1) Reset Z80");
  Serial.println("(2) Set Clock Speed");
  Serial.println("(3) Stop Clock");
  Serial.println(repeat(42, '-'));
  Serial.println("Free Run ISR Commands");
  Serial.println(repeat(42, '-'));
  Serial.println("(4) Attach Free Run ISR to INT1");
  Serial.println("(5) Detach Free Run ISR from INT1");
  Serial.println(repeat(42, '-'));
  Serial.println("Memory Simulation and Z80 Programming");
  Serial.println(repeat(42, '-'));
  Serial.println("(6) Initialize Simulated Memory");
  Serial.println("(7) Enter Program Into Simulated Memory");
  Serial.println("(8) Run Program (Attach Mem_Sim ISRs)");
  Serial.println("(9) Dump Simulated Memory");
  Serial.println(repeat(42, '-'));
```

Next, we print the status of the clock and whether simulated memory is available. Both of these checks are done the same way: see if the pointer is NULL. We initialized the pointers to NULL when we declared them in the global variables section, and when menu itself deletes these objects it resets the pointers to NULL, so it's a safe assumption that when the pointers are not null, there really is something out there that they point to.

Thus, if ourclock is not NULL, the address in ourclock must point to a valid clock_gen object. Since clock_gen's constructor demands a value to set the clock to, then sets the clock and starts it, the clock will be running if there's a clock_gen object.

Likewise, if there's something other than NULL in the mem_sim pointer, there will be a memory_simulator object at that address.

So we tell the user these things, and generate another dotted line.

```
  Serial.print("Clock is ");
  if (ourclock != NULL) {
    Serial.println("Running.");
  } else {
    Serial.println("Stopped.");
  }
```

```
Serial.print("Memory is ");
if (mem_sim != NULL) {
  Serial.println("Available.");
} else {
  Serial.println("Not Available.");
}
Serial.println(repeat(42, '-'));
```

Next, we call get_input_string(). When it returns, we pass the result into string2uint32_t, cast that result into an int, and use that result of the cast to switch to the correct case. You already know all about switches and case statements.

Menu Option 1

Option 1 is reset. We call the reset() member function of the object Z80. Remember that Z80 was declared as an object of the cpu class, and if you recall, the cpu class has a function called reset(). This is how object/member function calling is done. There are other ways to do it, but this is the most clear as far as I'm concerned.

```
switch ((int)string2uint32_t(get_input_string())) {
  case 1: Z80.reset();
    break;
```

Menu Option 2

Option 2 sets the clock speed. That's what the menu item says, at least. And sure enough, we set input_string equal to the results of calling get_input_string() like you'd expect. What this option really does, however, is this:

Check to make sure there's not already a clock_gen object that ourclock points to by deleting it and pointing it at NULL.

Call new and pass() the clock_gen class as an argument. The clock_gen constructor requires a numeric value of Hertz as a parameter. We have the String object the user typed in with that value as a text string, we call string2uint32_t on input_string, and pass the result as the parameter to the clock_gen constructor.

Congratulations, the clock_gen object ourclock finally exists, and the timer/counter is configured and the clock is running. But we need to reset the Z80.

```
  case 2:
    Serial.println("Menu: Enter Z80 Clock Speed (1 - 20000000Hz)");
    input_string = get_input_string();
    delete ourclock;
    ourclock = NULL; // clear the pointer. Otherwise things get confused.
    ourclock = new clock_gen(string2uint32_t(input_string));
```

Before we can reset the Z80, we should really disable writing to whatever memory_simulator object might happen to be available. We don't know whether one is or not, and we don't actually care. If mem_sim points to null, nothing will happen, and nothing needs to happen. So we just blindly call `mem_sim.m_write_enable=false`, right?

Well, no. Remember that mem_sim isn't an object of the memory_simulator class, it's a *pointer* to an object of the memory_simulator class. To access mem_sim's member variable m_write_enable, we have to dereference it with a ->.

Once we've disabled writes to any memory_simulator mem_sim may or may not point to, we can go ahead and call Z80.reset(). Once the reset comes back, we set `mem_sim->m_write_enable back to true.`

```
  mem_sim->m_write_enable=false;
  Z80.reset();
  mem_sim->m_write_enable=true;
  break;
```

Menu Option 3

Option 3 stops the clock. The only way to do this is to delete it. That's what option 3 does: call delete on the pointer ourclock, and then set the pointer's address to NULL. Does this seem unnecessary? It's not. I've seem some very strange behavior out of the clock without that step.

```
case 3:
  delete ourclock;
  ourclock = NULL; // Clear the pointer, otherwise things get confused.
  break;()
```

Menu Option 4

Option 4 hooks the free_run_ISR (which we'll talk about below) up to INT1, pretty much the way we did it in chapter 10, using noInterrupts, and attachInterrupt. The only difference is we make sure there's not already an ISR connected to INT1 by calling detachInterrupt(1) (for INT1), and then call attachInterrupt and re-enable interrupts.

```
case 4:
  Serial.println("Menu: Attaching free_run_ISR to INT1");
  noInterrupts();
  detachInterrupt(1);
  attachInterrupt(1, free_run_ISR, FALLING);
  interrupts();
  Z80.reset();
  break;
```

Menu Option 5

Option 5 detaches the free_run_ISR from INT1, and resets the Z80.

```
case 5:
  Serial.println("Menu: Detatching free_run_ISR from INT1");
  noInterrupts();
  detachInterrupt(1);
  interrupts();
  Z80.reset();
  break;
```

Menu Option 6

Option 6 does two critical things for our memory_simulator object. First, it deletes any that already exist, then sets the mem_sim pointer to NULL for safety. It then calls new on the object type memory_simulator() and sets the mem_sim pointer to that address. Congratulations. Achievement unlocked. We have a memory_simulator class object instantiated, its constructor has fired, and the pointer mem_sim now points to it.

```
case 6:
  delete mem_sim;
  mem_sim = NULL;
  mem_sim = new memory_simulator();
  break;()
```

Menu Option 7

Option 7 is to edit the contents of the memory_simulator object mem_sim points to. It calls mem_sim->m_edit(), which is the m_edit() member function of the memory_simulator object that mem_sim points to. There's no safety net here. I really have no idea what happens if you try to edit a nonexistent memory simulator.

I can't imagine it's good.

```
case 7:
  mem_sim->m_edit();
  break;
```

Menu Option 8

Option 8 hooks mem_read_ISR and mem_write_ISR up to INT1 and INT0, respectively. It works just exactly like attaching the free_run_ISR to INT1 did, except that there are two ISRs instead of one. We'll talk about those ISRs later, I promise.

```
    case 8:
      Serial.println("Menu: Attaching mem_read_ISR to INT1.");
      Serial.println("Menu: Attaching mem_write_ISR to INT0.");
      Serial.println("Menu: Any program therein should run.");
      noInterrupts();
      detachInterrupt(1);
      detachInterrupt(0);
      mem_sim->m_write_enable=false;
      attachInterrupt(1, mem_read_ISR, FALLING);
      attachInterrupt(0, mem_write_ISR, FALLING);
      mem_sim->m_write_enable=true;
      interrupts();
      Z80.reset();
      break;
```

Menu Option 9

Option 9, the last option, calls the m_dump() member function of the memory_simulator object that mem_sim points to. This starts to sound like one of those chain folk stories that ends "Piggy won't go over the style, and I sha'ant get home tonight." Pointers always wind up sounding that way. If you're curious, look up linked lists some time.

```
    case 9:
      mem_sim->m_dump();
      break;

  } // end of case.
} // end of menu function().
```

Free Run ISR

The free_run_ISR is attached to INT1 on the falling edge (since /RD is active low). When triggered, it sets the Z80 into wait mode, thus setting /WAIT active (low), then it reports the address requested and sends a 0x00 (NOP) on the Z80's data bus. Since NOP does nothing, the Z80 then goes on to the next address, and the next, and so on.

We begin by initializing a couple of variables and preserving the Z80's current mode.

```
void free_run_ISR() {
  byte temp;
  String output = "";
  Z80.save_mode();
```

We then call mode_wait() from the cpu class object Z80. This makes the Z80 wait until the ISR can service its memory request. The memory request is serviced immediately afterwards, with a call to data_in() on the same object, with a parameter of 0x0, the opcode for NOP.

```
  Z80.mode_wait();
  Z80.data_in(0x0);
```

After that, we build the output string with both bytes of the Z80's address bus, Z80. addr_msb and Z80.addr_lsb. Remember, those are alternate names for PINB and PINA, declared in the cpu class.

```
  output += String("free_run_ISR: Address: 0x");
  if (Z80.addr_msb < 0x10) output += String("0");
  output = output + String(Z80.addr_msb, HEX);
  if (Z80.addr_lsb < 0x10) output += String("0");
  output += String(Z80.addr_lsb, HEX);
  Serial.println(output);
```

Then we restore the Z80 to whatever mode it was in before, releasing /WAIT by setting it high.

```
  Z80.restore_mode();
}
```

That wasn't too bad. The good news is the next two ISRs work pretty much the same way, except that they access the memory_simulator object.

Memory Read ISR

Like free_run_ISR, mem_read_ISR is attached to INT1 on the falling edge. (Obviously you can't have both free_run_ISR and mem_read_ISR attached at the same time.) Like free_run_ISR, mem_read_ISR reads the address bus of the Z80 and returns a value to its data bus, but this ISR uses the m_seek_read() function of the memory_simulator class object pointed to by mem_sim to look up the value stored in the memory simulator at the address the Z80 asked for. It sends that value to the Z80's data bus, and also prints what it's doing for the user.

One other thing: this ISR also watches the /M1 signal, and tells the user whether the Z80 is in an M1 machine cycle, which is an instruction fetch. If /M1 isn't low, what we're doing is a data fetch. This turns out to be very useful for debugging assembly programs.

We begin by initializing a pair of variables, saving the Z80's existing control bus state, and setting the Z80 in wait mode. This works exactly the same way it did in free_run_ISR().

```
void mem_read_ISR() {
  uint8_t tempdata = 0;
  String output = "";

  Z80.save_mode();

  Z80.mode_wait();
```

Next, we load tempdata with the data stored at the address the Z80 requested. We do this by calling mem_sim->m_seek_read with the MSB and LSB of the Z80's address bus. Yes, the order looks wrong. It gets reversed again in m_seek_read so we're in the right order for a little endian CPU like the Z80.

```
  tempdata = mem_sim->m_seek_read(Z80.addr_msb, Z80.addr_lsb);
```

Next, we start building the output string. We begin with the results of testing the /M1 line.

```
  if (!Z80.M1()) {
    output += String("mem_read_ISR: Z80 Fetched Address: 0x");
  } else {
    output += String("mem_read_ISR: Z80 Read Address: 0x");
  }
```

Now we send the Z80 tempdata, which, you will recall, contains the value stored in the memory simulator at the address requested by the Z80. After that, we build the string to tell the user what just happened, clear the /WAIT signal, and we're done. Return control back to wherever it came from. ISRs should be as short as possible.

```
  Z80.data_in(tempdata);
  if (Z80.addr_msb < 0x10) output += String("0");
  output += String(Z80.addr_msb, HEX);
  if (Z80.addr_lsb < 0x10) output += String("0");
  output += String(Z80.addr_lsb, HEX);
  output += String("\tData: 0x");
  if (tempdata < 0x10) output += String("0");
  output += String(tempdata, HEX);
  Serial.println(output);
  Z80.restore_mode();
}
```

Memory Write ISR

The mem_write_ISR works almost exactly like the mem_read_ISR. If the code looks the same, it's because started out as a copy of the mem_read_ISR. There are differences.

The mem_write_ISR is attached to the falling edge of INT0, which is connected to the /WR line. It calls m_seek_write() instead of m_seek_read() on the memory simulator object. And, it calls Z80.data_out() instead of Z80.data_in() mem_write_ISR does not pay attention to the /M1 signal. Writes do not occur in M1 cycles.

We begin, as usual, setting up the output String object, and the uint8_t tempdata. We save the Z80's control bus mode, and activate the Z80's /WAIT signal so the mem_write_ISR is guaranteed time to service the request.

```
void mem_write_ISR() {
  String output = "";
  uint8_t tempdata = 0;

  Z80.save_mode();
  Z80.mode_wait();
```

We then copy Z80.data_out, the data the Z80 has placed on its data bus, into tempdata, and immediately call mem_sim->m_seek_write with the address bytes and tempdata to write the data to the memory simulator. Then we begin generating the output string for the user.

```
  tempdata = Z80.data_out();
  mem_sim->m_seek_write(Z80.addr_msb, Z80.addr_lsb, tempdata);
  output += String("mem_write_ISR: Z80 Wrote Address: 0x");
```

The Arduino HEX output system leaves the leading zeros off. This isn't wrong, but it's harder to keep track that this is a *byte* value, or a two-byte word value. What we do next is, if a byte (addr_msb, addr_lsb or tempdata) is less than 0x10, add a leading zero to it in the output string.

```
  if (Z80.addr_msb < 0x10) output += String("0");
  output += String(Z80.addr_msb, HEX);
  if (Z80.addr_lsb < 0x10) output += String("0");
  output += String(Z80.addr_lsb, HEX);
  output += String("\tData: 0x");
  if (tempdata < 0x10) output += String("0");
  output += String(tempdata, HEX);
```

Finally, we Serial.println the output string, restore the Z80's state, and we're done.

```
  Serial.println(output);
 Z80.restore_mode();
}
```

Setup()

The setup() function does practically nothing in this sketch. It sets up the serial monitor at 115,200 baud. That's it.

```
void setup() {
  Serial.begin(115200);
}
```

Loop()

The loop() function does even less. It calls menu(). That's all. Sure, the menu could have been in loop() instead of its own function, as we've done in all the previous chapters, but there's a certain amount of debate whether loop() or main() in more normal programming environments should do anything other than call functions. I chose to move the menu out to its own function this time to demonstrate that.

```
void loop() {
  menu();
}
```

And we're done with the sketch. As always, the full code, replete with comments, is next.

The Full Code

```
// Hardware
//-------------------------------------------------------------
// Z80 Signals Connected to ATmega1284P Port D:
//---------------------------------------------
// Signal: /WAIT /RESET /CLK       /M1 /RD /WR (-) (-)
// class: cpu    cpu    clock_gen  cpu cpu cpu (serial)
//
// All Other Z80 Signals
//----------------------
// /MREQ /IORQ /RFSH /HALT /INT   /NMI /BUSRQ
// n/c   n/c   n/c   n/c   button n/c  n/c
//-------------------------------------------------------------
// Z80 Address + Busses
// Signal:      A0-A7   A8-A15 D0-D7
// ATmega Port: PORTA   PORTB  PORTC
// Class:       cpu     cpu    cpu
//-------------------------------------------------------------
```

```
//-------------------------------------------------------------
// Preprocessor #defines
//-------------------------------------------------------------
#define MAX_CLOCK 20000000 // Maximum clock speed
#define MEM_SIZE 0x2000 // 8192 Bytes
#define RESET_MS 5000 // Z80 reset: hold /RESET low this long.

//-------------------------------------------------------------
// Function Prototypes
// Since some member functions of our classes call utility
// functions, it behooves us to declare function prototypes.
// Arduino 1.6.5 does this for us, but 1.6.7 seems to break
// that behavior.
//-------------------------------------------------------------
String repeat(int number, char character);
String get_input_string();
uint32_t string2uint32_t(String input);
uint16_t hex_string2uint16_t(String input);
void menu();

//-------------------------------------------------------------
// Classes
//
// Since we're using software running on the ATmega1284P to
// simulate hardware, it makes sense to break the code up
// into objects along the same lines.
//-------------------------------------------------------------

//-------------------------------------------------------------
// cpu class:
//
// An object of this class reads and writes the various pins
// of PORT D to control the Z80, read its status signals,
// provide access to its address bus, and read from or write
// to its data bus.
//-------------------------------------------------------------
class cpu {
  private:
    // These are private variables, constants, and (potentially)
    // member functions. These can't be touched by code outside
    // this class.
```

```
    uint8_t saved_control_port = CTRL_DEFAULT;
    // We often need to preserve the state of the
    // control port before we change it, and we may not
    // know exactly what it is. It's stored in this private
    // variable.

    const uint8_t M1_MASK = 0b00010000; //Read the /M1 signal
    const uint8_t RESET = 0b10111111; // Set the /RESET signal
    const uint8_t WAIT = 0b01111111; // Set the /WAIT signal
    // These consts define various bit values we need to
    // set or read the named status signals of the Z80
    // with PORT D.

    const uint8_t CTRL_DEFAULT = 0b11000000; // default Z80 state
    // This is the default state for PORTD and by extension, the Z80.
    // Hold /WAIT and /RESET high. /CLK is under timer control.
    // /WR, /MREQ and /RD are all inputs from the ATmega's view.

    volatile uint8_t& CONTROL_PORT = PORTD;
    volatile uint8_t& CONTROL_PINS = PIND;
    // These are reference variables. They are run-time
    // aliases for PORTD and PIND. Note the & sign.
    // That denotes them as references. Note that they are private.

    volatile uint8_t& DATA_PORT = PORTC;
    volatile uint8_t& DATA_DDR = DDRC;
    volatile uint8_t& DATA_PINS = PINC;
    // Reference variables for our data port, PORTC.  These
    // are private and strictly for programming convenience
    // within this class.

  public:
    cpu() { // constructor
      Serial.println("CPU: Object created.");
      DDRA = 0b00000000; // Address LSB
      DDRB = 0b00000000; // Address MSB
      DDRC = 0b11111111; // Data Port;
      DDRD = 0b11100000; // control_port
      // Initialize the DDRs for all ports. Since only PORTC
      // has a DDR reference varable, I'm using the register
      // names for everything for consistency.
```

```
  DATA_PORT = 0b00000000;
  CONTROL_PORT = CTRL_DEFAULT;
  // Initialize the values of DATA_PORT and CONTROL_PORT.
}
// This is the constructor of this class. It is called when
// an object of this class is instantiated. The constructor
// sets ports A and B up to read the low and high memory
// address bytes, and port C up (initially) to write data
// to the Z80, but this port can also read data sent from
// the Z80 to the system.

boolean M1() {

  return (CONTROL_PINS & M1_MASK);
}
// This function returns the status of the Z80's /M1 signal.
// It does a logical AND of CONTROL_PINS and the M1_MASK.
// If any pins in that AND turn up true, the function will
// return true. Because all Z80 signals are active low,
// this will mean /M1 is inactive.

void mode_default() {
  CONTROL_PORT = CTRL_DEFAULT;
}
// Set the CONTROL_PORT to its default. Put the Z80 in
// non-halted, non-waited mode.  Run mode, really.

void mode_wait() {
  CONTROL_PORT = CONTROL_PORT & WAIT;
}
// Lower the Z80's /WAIT signal so the Z80 will
// not try and read until our ISRs are ready.
// Even though the ATmega1284P is a universe
// faster than we're running the Z80, our
// memory is simulated in software. It's
// really, really slow.

void save_mode() {
  saved_control_port = CONTROL_PORT;
}
// Preserve the current value of CONTROL_PORT in the
// private saved_control_port variable.
```

```
    void restore_mode() {
      CONTROL_PORT = saved_control_port;
    }
    // set CONTROL_PORT to the value in saved_control_port.
    // Note that we don't check that a save_mode was done
    // first, so use with caution.

    volatile uint8_t& addr_msb = PINB;
    volatile uint8_t& addr_lsb = PINA;
    // These two references give our sketch
    // a consistent name for the address bytes.
    // That way we can access them without
    // adding any more instructions.

    uint8_t data_out() {
      DATA_DDR = 0b00000000;
      return DATA_PINS;
    }
    // Set the DATA_PORT to read mode, and read the
    // Z80's data port for data coming FROM the Z80.
    // Return that data.
    // ATmega is READING. Z80 is WRITING.

    void data_in(uint8_t data) {
      DATA_DDR = 0b11111111;
      DATA_PORT = data;
    }
    // Make sure the data port is in WRITE mode, and
    // set it to the value of data.
    // ATmega is WRITING, Z80 is READING.

    void reset(void) {
      Serial.println("CPU: Resetting...");
      CONTROL_PORT = CONTROL_PORT & RESET;
      delay(RESET_MS);
      Serial.println("CPU: Done Resetting.");
      CONTROL_PORT = CTRL_DEFAULT;
    }
    // Lower the Z80's /RESET signal for RESET_MS milliseconds.
    // In order to reset properly the Z80 needs its /reset
    // signal held low for multiple clock cycles. Given that
    // our clock's minimum speed is 1Hz, 5 seconds seems like
    // a safe value. This could be dynamic based on the clock
    // speed, but we're not in that much of a hurry.
};
```

```
//-------------------------------------------------------------
// clock_gen class
//
// This class configures timer/counter 1 to output a square
// wave signal on OCP1, from 1Hz to MAX_CLOCK MHz, depending
// on how we configure it. There are only two member functions,
// a constructor and the destructor. When an object in this
// class is instantiated, it's done with a text string
// parameter containing the text value the user wants
// the clock set to.
// The constructor handles the rest: generates the prescaler
// value and the match value, and tells the user what it's
// done. Note that because the Cestino has a 20MHz system
// clock, its maximum clock resolution is 50ns. Some
// speeds will be approximations since they don't divide
// evenly by 50ns.
//-------------------------------------------------------------
class clock_gen {
  private:
    int prescale_values[5] = {1024, 256, 64, 8, 1};
    int prescale_bits[5] = {0b101, 0b100, 0b011, 0b010, 0b001};
    // lookup tables for prescale bit fields and their values.

  public:
    // Destructor
    ~clock_gen() {
      TCCR1A = 0;
      TCCR1B = 0;
      OCR1A = 0;
      TCNT1 = 0;
      Serial.println("Clock Generator: Object Deleted.");
    }
    // The destructor is called when an object is deleted.
    // In this case, it clears all the timer registers, which
    // turns the timer off.

    // Constructor
    clock_gen(uint32_t desired_frequency) {
      Serial.println("Clock: Object Created.");
      float counter_value = MAX_CLOCK / desired_frequency;
      float lowest_inaccuracy = 1.0;
      float current_steps = 0;
      int prescaler = 1;
      byte prescaler_config_bits;
      long int match;
```

```
      for (int c = 0; c <= 4; c++) {
        current_steps = counter_value / prescale_values[c];
        if ((current_steps - round(current_steps) < lowest_inaccuracy)\
            && (current_steps <= 65535)) {
          lowest_inaccuracy = current_steps - ((int)current_steps);
          prescaler = prescale_values[c];
          match = round(current_steps);
          prescaler_config_bits = prescale_bits[c];
          if (match == 0) match = round(counter_value);
        }
      }
      // Find the highest prescaler value that will both allow the
      // clock to generate the value the user wanted, with the
      // lowest inaccuracy of the resulting clock speed.
      // We do this by determining the number of steps the
      // maximum clock speed available (20MHz on the Cestino)
      // will occur for each step of our clock's output. Then we
      // iterrate from highest to lowest prescaler values to find the
      // one that divides most evenly into the number of steps.
      // We set our match value to the number of steps we want
      // divided by the prescaler.

      Serial.print("We want to count to ");
      Serial.println(counter_value, 2);
      Serial.print("For a clock speed of ");
      Serial.println(desired_frequency, DEC);
      Serial.print("I chose a prescaler of ");
      Serial.println(prescaler, DEC);
      Serial.print("And a match of ");
      Serial.println(match, DEC);
      Serial.print("We'll count to ");
      Serial.println((long int)prescaler * match, DEC);
      // tell the user what values we generated.

      TCCR1A = 0b01000000;
      TCCR1B = (0b00001000 | prescaler_config_bits);
      TCNT1 = 0;
      OCR1A = match;
      Serial.println("Clock Generator: Running");
      // set the timer/counter registers.
    }
};
```

```
//-------------------------------------------------------------
// memory_simulator class
// This class contains the array we're using as simulated ram
// for the Z80, and functions to access it.
//-------------------------------------------------------------
class memory_simulator {
  private:
    volatile uint8_t halt = 0x76;
    volatile uint8_t mem_array[MEM_SIZE];
    // declare a variable to hold the halt instruction
    // for the Z80, and declare the array we're using
    // for simulated memory. These are private variables and
    // cannot be touched directly by outside code.

    void dump_page_header() {
      Serial.println("\nAddress\t 0  1  2  3  4  5  6  7  8  9" +
                    String("  a  b  c  d  e  f   Data (text)"));
      Serial.println(repeat(75, '-'));
    }
    //Print the header for the dump pretty printer.

    void m_edit_instructions() {
      Serial.println("\n\t*** Mem-Sim Line Editor ***");
      Serial.print("Enter an address and data in HEX ");
      Serial.println("eg: 0x0000,0x76");
      Serial.println("\"exit\" to quit\n\"dump\" to view memory\n");
      // Print the handy instructions.
    }

  public:
    volatile boolean m_write_enable = true;
    // If m_write_enable is false, m_seek_write() will say it
    // wrote, but it won't actually do it. A kludge used to
    // protect simulated memory during Z80 resets.

    memory_simulator() { // constructor
      Serial.println("Memory: Initializing...");
      for (int c = 0; c < MEM_SIZE; c++) {
        mem_array[c] = 0;
      }// Wipe the entire array. There's no telling what was
      // in that memory before we declared the array.

      Serial.println("Memory: " + String(MEM_SIZE, DEC) + \
                    " (0x" + String(MEM_SIZE, HEX) + ") bytes free.");
    }
    // tell the user how much memory we have.
```

```
// The constructor of memory_simulator wipes the memory
// array, which is declared on instantiation.after that,
// it reports the available RAM to the user. We use the
// default destructor, since we don't do anything special
// there.

volatile uint8_t m_seek_read(volatile uint8_t address_msb, volatile
uint8_t address_lsb) {
  volatile union {
    volatile uint16_t sixteen_bit_address;
    volatile uint8_t byte_array[2];
  } addr_byte_union;
  addr_byte_union.byte_array[0] = address_lsb;
  addr_byte_union.byte_array[1] = address_msb;
  // We're using this union to plug in the address_msb and address_lsb
  // variables and get out a uint16_t.  Make sure to put the bytes in
  // in little endian order (lsb first) or your results will be very
  // odd once you go over address 0x0020.  Been there, did that.

  if (addr_byte_union.sixteen_bit_address >= MEM_SIZE) {
    return halt;

  } else {
    return mem_array[addr_byte_union.sixteen_bit_address];
  }
  // Check to see if we're trying to seek a legit address. If not,
  // return a z80 halt instruction and throw an error message.
}
// The m_seek_read member function decodes two separate bytes (address_
msb and address_lsb)
// into a single uint16_t address, then goes to that address and returns
that cell of the
// array. If the decoded address exceeds MEM_SIZE, we return halt,
otherwise we return
// the array at that address.

void m_seek_write(volatile uint8_t address_msb, \
volatile uint8_t address_lsb, volatile uint8_t data) {
  if (m_write_enable) {
    // On Z80 resets we can get spurious triggering
    // of the write ISR resulting in random writes
    // to memory. Reset sets m_write_enable to false, then
    // back to true when the reset is done.
```

```
    volatile union {
      volatile uint16_t sixteen_bit_address;
      volatile uint8_t byte_array[2];
    } addr_byte_union;
    addr_byte_union.byte_array[0] = address_lsb;
    addr_byte_union.byte_array[1] = address_msb;
    // We're using this union to plug in the address_msb and address_lsb
    // variables and get out a uint16_t.  Make sure to put the bytes in
    // in little endian order (lsb first) or your results will be very
    // odd once you go over address 0x0020.  Been there, did that.

    if (addr_byte_union.sixteen_bit_address >= MEM_SIZE) {

    } else {
      mem_array[addr_byte_union.sixteen_bit_address] = data;
    }
    // Check to see if we're trying to seek a legit address. If not,
    // fail silently. Ugh.
  }
}
// The m_seek_write member function decodes the two bite field address
// the same way m_seek_read does. If the decoded address exceeds
// MEM_SIZE, we throw an error message, otherwise we set the
// array[address] to data.

void m_dump() {
  Serial.println("Dumping Simulated Memory");
  int c = 0;
  String line_start_address = "0x";
  uint16_t row_address = 0;
  String hex_data = "";
  String human_readable_data = "";
  // We have some variables. Initialize them.

  dump_page_header();
  //show the header

  for (int row = 0; row < (MEM_SIZE / 16); row++) {
    row_address = 16 * row;
    if (row_address < 0x1000)line_start_address += "0";
    if (row_address < 0x0100)line_start_address += "0";
    if (row_address < 0x0010)line_start_address += "0";
    line_start_address += String(row_address, HEX) + "|";
```

```
        for (int col = 0; col < 16; col++) {
          // We're printing blocks of 256 bytes.
          // each is 2 characters + two spaces
          // in hex, plus 1 character in text.
          if (mem_array[c] < 0x10) hex_data += "0";
          // Add leading zero for hex values below 0x10.

          hex_data += String(mem_array[c], HEX);
          hex_data += " ";
          // Add mem_array[c] as a hex string to hex_data.

          if (isprint(mem_array[c])) {
            human_readable_data += (char)mem_array[c];
          } else {
            human_readable_data += ".";
          }
          // If mem_array[c] is printable, add it to
          // human_readable_data. Otherwise add a . for
          // a placeholder.
          c++;
        }

        Serial.println(line_start_address + " " + hex_data + \
                       " | " + human_readable_data);
        hex_data = "";
        human_readable_data = "";
        line_start_address = "0x";
        // Combine the three strings in one Serial.println.
        // Then clear them.
        if (!(c % 256)) {
          Serial.println("Continue (y/n)");
          if (get_input_string() == "n") {
            row = MEM_SIZE / 16;
          } else {
            dump_page_header();
            //show the header for each page, actually.
          }
        }
      }
    }
    // m_dump produces a human-readable dump of the memory array in
    // 256 byte pages, with each line 16 (0xF) items long.
```

```
void m_edit() {
  String input_string = "y";
  uint16_t address;
  uint8_t  data;
  int comma_index = 0;
  String temp;

  m_edit_instructions();

  do {
    // repeat is loop until the 'while' is satisfied.
    input_string = get_input_string();
    // get the input string.
    if (input_string == "dump") {
      m_dump();
      m_edit_instructions();
      // If the user types "dump," dump the memory to
      // the serial console. Don't try and process it
      // into code. Show the instructions again.
    }
    if (input_string.startsWith("0x", 0)) {
      comma_index = input_string.indexOf(",");
      // find the comma in the input string

      temp = input_string.substring(0, comma_index);
      temp.trim();
      // copy the string from the beginning to the comma into temp.
      // also trim it - get rid of leading and trailing spaces.

      address = hex_string2uint16_t(temp);
      // call hex_string2uint16_t with temp. Set the results into
      // address.

      temp = input_string.substring(comma_index + 1);
      temp.trim();
      data = (uint8_t)hex_string2uint16_t(temp);
      // repeat the process with the back half of the input string.

      Serial.println(">  Addr: 0x" + String(address, HEX) + \
                     " Data: 0x" + String(data, HEX) + " [OK]");
      // Print the line describing what is being entered where.
```

```
          mem_array[address] = data;
          // actually enter it.
        }
      } while (input_string != "exit");
      // if the user types "exit" stop repeating the loop.
    }
    //m_edit edits the memory array directly, allowing the user
    // to deposit a hex value in 0x notation at a hex address,
    // also in 0x notation. It allows the user to dump the
    // array to the screen at will, and exits on the exit
    //command.
};

//-------------------------------------------------------------
// Global variables
// The big advantage to using C++ objects in this project
// was the drastic reduction of global variables. There
// are still quite a few variables, but most of them are
// contained in functions or in classes and don't exist
// until an object of that class is instantiated.
//-------------------------------------------------------------
clock_gen* ourclock = NULL; // pointer to a clock_gen object.
memory_simulator* mem_sim = NULL; // pointer to memory object.
cpu Z80; // declare our CPU object. Not a pointer.

//-------------------------------------------------------------
// Functions which are not members of a class.
//-------------------------------------------------------------

//-------------------------------------------------------------
// Repeat(int number char character)
// This function creates a string of character
// exactly /number/ long.
//-------------------------------------------------------------
String repeat(int number, char character) {
  String temp = "";
  for (int c = 0; c < number; c++) {
    temp = temp + String(character);
  }
  return temp;
}
//-------------------------------------------------------------
// get_input_string()
// This function loops forever waiting for an input string
// and returns the string when it gets one.
```

```
//-------------------------------------------------------------
String get_input_string() {
  while (!Serial.available()) {
  }
  return (Serial.readString());
}

//-------------------------------------------------------------
// string2uint32_t()
//- The String class includes no nice way to turn strings of
// digits into a numeric value. (Actually, there's an
// undocumented one. Classy.)
//
// This function goes through the string from left to right.
// For each position it advances to the right, it multiplies
// the existing value by 10 and adds the value of the
// character at that position minus the value of the
// character '0'. When we reach the end of the string,
// return temp.
//-------------------------------------------------------------
uint32_t string2uint32_t(String input) {
  uint32_t temp = 0;
  input.trim();
  for (int c = 0; c < input.length(); c++) {
    temp = temp * 10 + input.charAt(c) - '0';
  }
  return temp;
}

//-------------------------------------------------------------
// hex_string2uint16_t()
// This function goes through the string from left to right.
// For each position it advances to the right, it multiplies
// the existing value by 16. if the new digit is 0-9 (less
// than the value of ':') add the character's numeric value
// minus the value of '0',  A-F and a-f are processed the
// same way, except that we check to make sure the new
// digit is actually in the range and subtract a different
// value.  Once we have the correct value for the character,
// we add the value of the character at that position minus
// the value of the character '0'. When we reach the end
// of the string, return temp.
//-------------------------------------------------------------
```

```
uint16_t hex_string2uint16_t(String input) {
  uint16_t temp = 0;
  char tempchar;
  input.trim();
  for (int c = 2; c < input.length(); c++) {
    temp = (temp * 0x10);
    tempchar = input.charAt(c);
    if (tempchar < ':') temp += (tempchar - '0');
    if ((tempchar > '@') && (tempchar < 'G')) temp += (tempchar - '7');
    if ((tempchar > '`') && (tempchar < 'g')) temp += (tempchar - 'W');
  }
  return temp;
}

//------------------------------------------------------------
// menu
// This function generates the menu and selects which function
// to call, whether it's in an object or not.
//------------------------------------------------------------
void menu() {
  String input_string = "";
  Serial.println("\n" + repeat(15, ' ') + "*** Menu ***");
  Serial.println(repeat(42, '-'));
  Serial.println("Z80 Operations Commands");
  Serial.println(repeat(42, '-'));
  Serial.println("(1) Reset Z80");
  Serial.println("(2) Set Clock Speed");
  Serial.println("(3) Stop Clock");
  Serial.println(repeat(42, '-'));
  Serial.println("Free Run ISR Commands");
  Serial.println(repeat(42, '-'));
  Serial.println("(4) Attach Free Run ISR to INT1");
  Serial.println("(5) Detach Free Run ISR from INT1");
  Serial.println(repeat(42, '-'));
  Serial.println("Memory Simulation and Z80 Programming");
  Serial.println(repeat(42, '-'));
  Serial.println("(6) Initialize Simulated Memory");
  Serial.println("(7) Enter Program Into Simulated Memory");
  Serial.println("(8) Run Program (Attach Mem_Sim ISRs)");
  Serial.println("(9) Dump Simulated Memory");
  Serial.println(repeat(42, '-'));
  // print the text of the menu.
```

```
Serial.print("Clock is ");
if (ourclock != NULL) {
  Serial.println("Running.");
} else {
  Serial.println("Stopped.");
}
Serial.print("Memory is ");
if (mem_sim != NULL) {
  Serial.println("Available.");
} else {
  Serial.println("Not Available.");
}
Serial.println(repeat(42, '-'));
// check to see if there's an object attached to ourclock. If not,
// then the clock is not enabled. Tell the user one way or the other.
// Likewise, if there's an object attached to mem_sim, memory is
// enabled. Otherwise it's not. Tell the user.

switch ((int)string2uint32_t(get_input_string())) {
  case 1: Z80.reset();
    break;
  // Option 1 is reset. Call the reset member function of Z80.

  case 2:
    Serial.println("Menu: Enter Z80 Clock Speed (1 - 20000000Hz)");
    input_string = get_input_string();
    delete ourclock;
    ourclock = NULL; // clear the pointer. Otherwise things get confused.
    ourclock = new clock_gen(string2uint32_t(input_string));
    mem_sim->m_write_enable=false;
    Z80.reset();
    mem_sim->m_write_enable=true;
    break;
  // Option 2 is start the clock/set the clock. Call input_string()
  // and stash the results in a variable by the same name.
  // Delete anything that's already on the ourclock pointer
  // and set it to null.
  // Do the delete so as not waste system resources. Do the
  // set to null so as not to get weird clock behavior.
  // Then instantiate a clock_gen object. Take input_string and
  // feed it to string2uint32_t, and feed the output of /that/
  // to the clock_gen's constructor, and let it do all the work.
```

```
case 3:
  delete ourclock;
  ourclock = NULL; // Clear the pointer, otherwise things get confused.
  break;
// Option 3 stops the clock. To do this, delete the object pointed to
// by ourclock, and set ourclock to NULL so the next clock doesn't have
// weird behavior.
case 4:
  Serial.println("Menu: Attaching free_run_ISR to INT1");
  noInterrupts();
  detachInterrupt(1);
  attachInterrupt(1, free_run_ISR, FALLING);
  interrupts();
  Z80.reset();
  break;
// Option 4 hooks up the free running ISR to interrupt 1, which listens
// for /mreq events. Every time the Z80 asks for memory and pulls
// this signal low, our ISR will fire. Free-running means we always give
// the Z80 NOP instructions (do nothing, go on to the next address),
// so we can watch and see if the address signals change. Turn
// interrupts off, attach the ISR to interrupt zero on the falling
// edge (/MREQ is active low), then turn interrupts back on.
// Finally, reset the Z80 so it starts from address 0x0000 in
// the output.

case 5:
  Serial.println("Menu: Detatching free_run_ISR from INT1");
  noInterrupts();
  detachInterrupt(1);
  interrupts();
  Z80.reset();
  break;
// You know how the last option attached the free_run_ISR?
// Option 5 detatches it. Turn interrupts off, detatch
// interrupt 1, turn interrupts back on, then reset
// the Z80 on general principles.

case 6:
  delete mem_sim;
  mem_sim = NULL;
  mem_sim = new memory_simulator();
  break;
```

```
// Option 6 turns on simulated RAM for the Z80. Delete anything
// on the mem_sim pointer, and set the pointer to NULL to avoid
// memory strangeness. Then instantiate a memory_simulator
// object and attach it to the mem_sim pointer. Memory_simulator
// objects' constructor takes no parameters.

case 7:
  mem_sim->m_edit();
  break;
// Option 7 is enter a program into simulated memory.
// call the m_edit member function of the memory simulator.
// This lets the user put simple, hand-assembled programs
// into the simulated memory for the Z80 to run. It's
// slightly less tedious than doing it with toggle switches
// on a front panel.(but only slightly).

case 8:
  Serial.println("Menu: Attaching mem_read_ISR to INT1.");
  Serial.println("Menu: Attaching mem_write_ISR to INT0.");
  Serial.println("Menu: Any program therein should run.");
  noInterrupts();
  detachInterrupt(1);
  detachInterrupt(0);
  mem_sim->m_write_enable=false;
  attachInterrupt(1, mem_read_ISR, FALLING);
  attachInterrupt(0, mem_write_ISR, FALLING);
  mem_sim->m_write_enable=true;
  interrupts();
  Z80.reset();
  break;
// Just as option 4 attached the free running ISR to interrupt 1,
// option 8 connects the memory simulator ISR to interrupt 1.
// This means that when the Z80 lowers its /MREQ signal and
// requests memory, our ISR will try to service it with calls
// to the memory_simulator object connected to mem_sim. Why
// isn't the ISR in the object?  Because the Arduino core
// won't let you. Same as with option 4. Stop interrupts, detatch
// anything already connected to interrupt 1, attach mem_sim_ISR
// to interrupt 1 on the falling edge, then turn interrupts back on
// and reset the Z80 so our output starts at 0x0000.
// NOTE - CHNAGED FROM FALLING TO LOW
```

```
    case 9:
      mem_sim->m_dump();
      break;
      // Option 9 dumps the simulated memory array to the serial
      // console, one 256 byte page at a time. Which gets tedious
      // going through 8 kilobytes, but it gets there. We just call
      // the m_dump() function of the memory_simulator object
      // connected to the mem_sim pointer.
      // Such a tangled web we weave.
  } // end of case.
} // end of menu function.

//--------------------------------------------------------------
// free_run_ISR()
// This function is an interrupt service routine for
// INT1. When INT1 fires, we're in a read cycle.
// This ISR prints out the 16 bit address
// requested by the Z80, and returns a 0x00 (NOP) to the
// Z80, telling it to do nothing and go the next address,
// allowing us to observe the address lines (and make sure
// they all work and are connected correctly.
//--------------------------------------------------------------
void free_run_ISR() {
  byte temp;
  String output = "";
  Z80.save_mode();
  // Save the control signals we're sending to the Z80.

  Z80.mode_wait();
  // Set the Z80's mode to wait, so it stops asking for
  // new addresses while the ISR is trying to service
  // this request.

  Z80.data_in(0x0);
  // Always send the Z80 a NOP (0x0).

  output += String("free_run_ISR: Address: 0x");
  if (Z80.addr_msb < 0x10) output += String("0");
  output = output + String(Z80.addr_msb, HEX);
  if (Z80.addr_lsb < 0x10) output += String("0");
  output += String(Z80.addr_lsb, HEX);
  Serial.println(output);
  // Build the output string to show the user
  // what address was requested. This is what
  // free running is for.
  Z80.restore_mode();
  // un-wait the Z80.
}
```

```
//-------------------------------------------------------------
// mem_read_ISR()
// This function is an interrupt service routine for
// INT1. Like free_run_ISR, the first thing it does
// is save the Z80 control signal state, then set the Z80
// into wait mode, so we can survice this memory request
// before the Z80 asks for the next one. After that, it sends
// data FROM the memory simulator TO the Z80 (on read)
// After that, it builds up a string to tell the user what
// just happened and prints it.
//-------------------------------------------------------------
void mem_read_ISR() {
  uint8_t tempdata = 0;
  String output = "";

  Z80.save_mode();
  // Save the Z80 control signal state.

  Z80.mode_wait();
  // Put the Z80 into wait mode.

  tempdata = mem_sim->m_seek_read(Z80.addr_msb, Z80.addr_lsb);

  if (!Z80.M1()) {
    output += String("mem_read_ISR: Z80 Fetched Address: 0x");
  } else {
    output += String("mem_read_ISR: Z80 Read Address: 0x");
  }
  Z80.data_in(tempdata);
  // On a read cycle (from the Z80's perspective)
  // tell the object pointed to by mem_sim to seek the address
  // present on the Z80's address bus, and send the data
  // stored there to the Z80's data bus.
  // Tell the user that the Z80 read the address.

  if (Z80.addr_msb < 0x10) output += String("0");
  output += String(Z80.addr_msb, HEX);
  if (Z80.addr_lsb < 0x10) output += String("0");
  output += String(Z80.addr_lsb, HEX);
  output += String("\tData: 0x");
  if (tempdata < 0x10) output += String("0");
  output += String(tempdata, HEX);
  Serial.println(output);
  // Build the rest of the output string and display it. /M1
  // is the Z80's signal to indicate it's doing an instruction
  // fetch. If it is, tell the user so.
```

```
  Z80.restore_mode();
  //Restore the Z80 to non-halted mode.
}
//-----------------------------------------------------------
// mem_write_ISR()
// This function is an interrupt service routine for INT0.
// like mem_read_ISR(), the first thing it does is
// is save the Z80 control signal state, then set the Z80
// into wait mode, so we can survice this memory request
// before the Z80 asks for the next one. After that, it sends
// data FROM the Z80 TO the memory simulator.
// After that, it builds up a string to tell the user what
// just happened and prints it.
//-----------------------------------------------------------
void mem_write_ISR() {
  String output = "";
  uint8_t tempdata = 0;

  Z80.save_mode();
  // Save the Z80 control signal state.

  Z80.mode_wait();
  // Put the Z80 into wait mode.

  tempdata = Z80.data_out();
  mem_sim->m_seek_write(Z80.addr_msb, Z80.addr_lsb, tempdata);
  output += String("mem_write_ISR: Z80 Wrote Address: 0x");
  // On write cycle (from the Z80's perspective)
  // tell the memory simulator object to seek the address
  // present on the Z80's address bus, and set that address
  // of the memory simulator TO the value on the Z80's
  // data bus.

  if (Z80.addr_msb < 0x10) output += String("0");
  output += String(Z80.addr_msb, HEX);
  if (Z80.addr_lsb < 0x10) output += String("0");
  output += String(Z80.addr_lsb, HEX);
  output += String("\tData: 0x");
  if (tempdata < 0x10) output += String("0");
  output += String(tempdata, HEX);
  Serial.println(output);
  //Build the rest of the output string and display it.
```

```
  Z80.restore_mode();
  //Restore the Z80 to non-halted mode.
}
//-------------------------------------------------------------
// Setup
// Sets the serial console speed.
//-------------------------------------------------------------
void setup() {
  Serial.begin(115200);
}

//-------------------------------------------------------------
// Loop
// Calls the menu() function. Over and over again.
//-------------------------------------------------------------
void loop() {
  menu();
}
```

Output

So what happens when you run the sketch? Here's a log file where I reset the Z80, set the clock, delete the clock, set the clock again, and run the free_run_ISR. This log tells me my Z80 explorer is working correctly. Here's the log file.

```
               *** Menu ***
------------------------------------------
Z80 Operations Commands
------------------------------------------
(1) Reset Z80
(2) Set Clock Speed
(3) Stop Clock
------------------------------------------
Free Run ISR Commands
------------------------------------------
(4) Attach Free Run ISR to INT1
(5) Detach Free Run ISR from INT1
------------------------------------------
Memory Simulation and Z80 Programming
------------------------------------------
(6) Initialize Simulated Memory
(7) Enter Program Into Simulated Memory
(8) Run Program (Attach Mem_Sim ISRs)
(9) Dump Simulated Memory
```

```
-----------------------------------------
Clock is Stopped.
Memory is Not Available.
-----------------------------------------
```

I selected option 1.

```
CPU: Resetting...
CPU: Done Resetting.
```

The menu printed again, and I selected option 2. I chose a clock speed of 5Hz.

```
Menu: Enter Z80 Clock Speed (1 - 20000000Hz)
Clock: Object Created.
We want to count to 4000000.00
For a clock speed of 5
I chose a prescaler of 256
And a match of 15625
We'll count to 4000000
Clock Generator: Running
CPU: Resetting...
CPU: Done Resetting.
```

The menu printed again. Note the change where it says Clock is Running. I selected option 3, to stop the clock.

```
               *** Menu ***
[most of menu cut out.]
-----------------------------------------
Clock is Running.
Memory is Not Available.
-----------------------------------------

Clock Generator: Object Deleted.
```

The menu printed out again. I selected option 2 and asked for a 250Hz clock. This is about as fast as the terminal monitor can keep up with. It's far too fast to read, but a full free run spans 65,535 addresses. It takes some time. Even the slowest Z80s normally ran 10,000 times as fast, but they weren't waiting for a printout at 115,200 baud between every M1 cycle.

```
               *** Menu ***
[most of menu cut out]
-----------------------------------------
Clock is Stopped.
Memory is Not Available.
-----------------------------------------
```

I selected option 2.

```
Menu: Enter Z80 Clock Speed (1 - 20000000Hz)
Clock: Object Created.
We want to count to 80000.00
For a clock speed of 250
I chose a prescaler of 64
And a match of 1250
We'll count to 80000
Clock Generator: Running
CPU: Resetting...
free_run_ISR: Address: 0x0c0a
CPU: Done Resetting.
```

The menu printed again. I selected option 4, Attach Free Run ISR. The serial monitor got very, very busy.

```
               *** Menu ***
[most of menu cut out]
------------------------------------------
Clock is Running.
Memory is Not Available.
------------------------------------------

Menu: Attaching free_run_ISR to INT1
CPU: Resetting...
CPU: Done Resetting.
```

The menu printed again. Why? We're actually still in the loop() function, so menu gets called and will wait forever for input. All the Serial.printlns are coming from free_run_ISR.

```
               *** Menu ***
------------------------------------------
Z80 Operations Commands
------------------------------------------
(1) Reset Z80
(2) Set Clock Speed
(3) Stop Clock
------------------------------------------
Free Run ISR Commands
------------------------------------------
(4) Attach Free Run ISR to INT1
(5) Detach Free Run ISR from INT1
------------------------------------------
Memory Simulation and Z80 Programming
------------------------------------------
```

```
(6) Initialize Simulated Memory
(7) Enter Program Into Simulated Memory
(8) Run Program (Attach Mem_Sim ISRs)
(9) Dump Simulated Memory
-----------------------------------------
Clock is Running.
Memory is Not Available.
-----------------------------------------

free_run_ISR: Address: 0x0001
free_run_ISR: Address: 0x0002
free_run_ISR: Address: 0x0003
free_run_ISR: Address: 0x0004
free_run_ISR: Address: 0x0005
free_run_ISR: Address: 0x0006
free_run_ISR: Address: 0x0007
free_run_ISR: Address: 0x0008
free_run_ISR: Address: 0x0009
free_run_ISR: Address: 0x000a
free_run_ISR: Address: 0x000b
free_run_ISR: Address: 0x000c
free_run_ISR: Address: 0x000d
free_run_ISR: Address: 0x000e
free_run_ISR: Address: 0x000f
```

That's the first 16 addresses. You'll note as you get above 0x00ff that every address whose MSB is odd (so there's a 1 in its lowest bit) will light up the pin 1 LED. This is a good sign that things are working. At 250Hz, it will pulse on and off a little over a second at a time as we run through the whole range of addresses with an odd-numbered MSB. Here are a couple of the addresses where that occurs.

```
free_run_ISR: Address: 0x0500
free_run_ISR: Address: 0x0501
```

On and on it goes. It gets a little dull to watch if everything's working right. Finally, after many thousands of addresses I cut out for brevity:

```
free_run_ISR: Address: 0xfffd
free_run_ISR: Address: 0xfffe
free_run_ISR: Address: 0xffff
```

After that, I sent the Cestino a 3, for option 3. Even though the menu had long since scrolled off the screen, deleting the clock object stopped the torrent of addresses.

If you got this far, and your output from the free_run_ISR looks more or less like mine, your Z80 explorer is probably working correctly. We'll learn to program the Z80 in the next section.

Assembly and Machine Language

There are essentially three types of operation the Z80 knows how to do. Copy data from one place to another, change the control flow of the program, and math/logic functions. When you expand all the permutations of all its instruction families (we've only seen the ALU instructions so far) there are hundreds and we're not going to cover them all. It's easier to try and understand the *kind* of instructions you have available.

This isn't C++. What you have is empty RAM and tools for modifying what's in it, mostly one byte at a time. From the discussion of the ALU and its instructions back in Z80 Microprocessor Anatomy, you know that the Z80 knows how to add and subtract, rotate left and right, AND, OR, XOR, and how to check individual bits in bytes. Any higher level mathematical instructions have to be built by the programmer.

The rest of the instruction set is the same way. You can LD a byte from a memory location pointed to by a register, a byte in a register, or a literal value passed as a parameter to a different memory location pointed to by a register, or a different register.

There are no complex data structures, either. Machine instructions know nothing about strings or characters or uint8_ts. They know registers, flags in the F register, literal values, bits, and pointers to memory. That's it.

You also have to understand memory. Some would say that to understand assembly or machine instructions, you must understand memory first. Thankfully, the Z80's memory map is very simple, at least the way we're using it. Addresses start at zero and run to the end of memory (8192 bytes, in our case, but it can go to 65,535 bytes, or 64k). The first 256 bytes may or may not have special functions (interrupt handlers may go there, if you're using them, and so on.) and somewhere you'll have a stack, pointed to by the SP (stack pointer) register, and it will grow *backwards* from high values of memory to lower ones. Nothing protects one memory function from another. If you tell the Z80 to push so much stuff to the stack that it clobbers your code? It clobbers your code. If you give it an address to write to that is in the middle of the stack or your code, guess what. It clobbers the stack or your code. Working at this level gives you a lot of power, and as much speed as the CPU can give you, but the cost is knowing for sure what the program will do at every phase of its operation. There are no safety nets.

By contrast, resetting the Z80 takes seconds (five in our case, although most Z80 systems reset for a few milliseconds). So the price of a bug isn't that high.

I've used the word *instructions* a lot. I should mention what it means, and what assembly language and machine language are.

An instruction is a command. You've seen that they can be one or more bytes. They can take parameters, in the form of literal values or pointers to other parts of memory. They cause the Z80 to do something specific.

Machine language is a series of instructions and their parameters (and data) in memory. Machine language is the raw bytes, in hex or binary (we'll use hex.) Assembly language is code that you feed to an assembler to get machine language. Traditionally, every assembly mnemonic, like NOP, LD, or ADD has machine code associated with it, and you can translate literally from assembly to machine code and from machine code back into assembly.

The truth is that assemblers also have what are called macros. If you want to use an area of RAM as a variable, you can set a macro to do that, and every time you touch that area of ram in your assembly code, the assembler will generate the instructions needed to access it. It's a fuzzy line between a sophisticated assembler and a compiler.

What we'll be doing is called hand assembly. We'll be looking up the instructions, putting the opcodes together so we get the right version of the instruction, doing the translation of instructions and addresses and *everything* into hex codes, which we will put in memory by hand.

You've seen videos of the old days when people toggled programs into early micro-computers and read the results out on arrays of LEDs. This is exactly what they were doing. Once you understand how to write machine language, assembly language is a lot more clear.

Program 1: Infinite Loop

The first machine language program we'll write is an infinite loop. Anyone who's ever messed with a computer with a BASIC interpreter has done this:

```
10 print "Hello world"
20 goto 10
```

We're not even going to do the printing part. We don't have any hardware to do the job, and we don't need it. The mem_read_ISR will show us the loop as it happens by showing us the changes in address.

The infinite loop is one instruction with a two byte memory address for a parameter. Here's the assembly version, with the mnemonics.

```
JP 0x0001
```

Traditionally, in an assembler, we'd put a semicolon after that to explain what we were doing and why. We're hand assembling, so that winds up in your notebook instead.

The opcode for the version of JP that goes to a literal address is 0xC3. Because our Z80 is little endian, we have to put the least significant byte of the address we want to go to first, then the most significant byte. We also need to write down what address in memory each instruction will be on, both so we know where to put it, and so, when we're writing the code, we know what addresses to jump to.

What we wind up with is this:

```
0x01 0xc3
0x02 0x01
0x03 0x00
```

Moving forward, I'll put the two representations together. (That's how they are in my notes, too.)

```
        JP 0x0001
0x01 0xC3
0x02 0x01
0x03 0x00
```

Go ahead and start up the Z80 Explorer, and get to the menu. Mac users, you may have to reset the Cestino manually when you start the serial monitor.

First, Initialize Simulated Memory, option 6. You'll remember from the sketch that this instantiates the memory simulator object and points the mem_sim pointer at it. We're at a point in the game where that no longer matters. We need to focus on the Z80.

Next, select option 7, Enter Program Into Simulated Memory.

Type the address and opcode pairs above.

Here's what it looks like on my screen:

```
        *** Mem-Sim Line Editor ***
Enter an address and data in HEX eg: 0x0000,0x76
"exit" to quit
"dump" to view memory

>  Addr: 0x1 Data: 0xC3 [OK]
>  Addr: 0x2 Data: 0x1 [OK]
>  Addr: 0x3 Data: 0x0 [OK]
```

Typing "dump," I get:

```
Dumping Simulated Memory


Address  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   Data (text)
---------------------------------------------------------------------------
0x0000| 00 c3 01 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0010| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0020| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0030| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0040| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0050| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0060| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0070| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0080| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0090| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00a0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00b0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00c0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00d0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00e0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00f0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
Continue (y/n)
```

It looks a little strange with nothing in the human-readable part of the field, but if you look at the hex dump, you can see that, in fact, everything is fine, and our three-byte program is there.

Go ahead and answer "n" to exit from the editor, and type "exit".

Then set the clock to about 10Hz, and attach the Memory Simulator ISRs (options 2 and 8, respectively.)

You should get output from the mem_read_ISR that looks like this:

```
mem_read_ISR: Z80 Fetched Address: 0x0000       Data: 0x00
mem_read_ISR: Z80 Fetched Address: 0x0001       Data: 0xc3
mem_read_ISR: Z80 Read Address: 0x0002          Data: 0x01
mem_read_ISR: Z80 Read Address: 0x0003          Data: 0x00
mem_read_ISR: Z80 Fetched Address: 0x0001       Data: 0xc3
mem_read_ISR: Z80 Read Address: 0x0002          Data: 0x01
mem_read_ISR: Z80 Read Address: 0x0003          Data: 0x00
mem_read_ISR: Z80 Fetched Address: 0x0001       Data: 0xc3
mem_read_ISR: Z80 Read Address: 0x0002          Data: 0x01
mem_read_ISR: Z80 Read Address: 0x0003          Data: 0x00
Clock Generator: Object Deleted.
```

Note that because memory address 0x0000 has nothing in it, it's a NOP. We'll read it and go on to the next address. We fetch again from 0x0001, and get the instruction 0xC3. That's our JP instruction.

Next we read two more addresses. Note that they're not instruction fetches. We're not in an M1 cycle, so the /M1 signal is not active (low). It's reading those two bytes in 0x0002 and 0x0003 as parameters for the JP, which is exactly what it should do. Look what happens next.

All of a sudden, we're back fetching at address 0x0001. The JP worked! I let it run for three more trips through the loop and hit option 3, Stop Clock.

The menu was scrolled off the serial monitor, but loop calls menu() every time we exit menu(), and we only exit menu when we've given it an option number. So menu is sitting there waiting for input the whole time while we watch our Z80 program loop endlessly from address 0x0001 to 0x003, over and over again.

Program 2: Index Controlled Loop

Infinite loops are fun for demonstrations, but they're not the most useful of programs. For the next machine language program, we'll write a loop controlled by an index. This is more or less the same as:.

```
for (PORTB = 5; PORTB == 0; PORTB--) {
    //do nothing.
  }
```

You could use an int, of course, and declare it right in the loop. We've done it lots of times, except that I sneakily decrement using PORTB--. In this case, however, using the ATmega's PORTB register to do the job is much closer to how the loop works in machine language.

Here's the code

```
        LD B,0x05 ; load register B with the value 0x05.
0x01 0x06
0x02 0x05
        DEC B; Decrement the B register
0x03 0x05
        JP Z 0x0A; Jump to 0x0A when B hits 0 and the Z flag is set.
0x04 0xCA
0x05 0x0A.
0x06 0x00
        JP 0x03 ; If we get here, we're not done. Jump to 0x03
0x07 0xC3
0x08 0x03
0x09 0x00
        HLT ; If we get here, halt the CPU.
0x0A 0x76
```

All that just for a loop? Yes. We're indexing this loop on the B register, as it's a general purpose register. We set it to 5 in the first instruction.

```
LD B, 0x05.
```

In the next instruction we decrement B. This works, but doing it here before the test means the loop will actually only run four times. Oops.

```
DEC B
```

The next instruction is the test. You'll find a lot of loops run backwards in assembly and machine code. It's much simpler and much faster to decrement and check a flag than to add and compare. The DEC instruction sets the Z flag automatically if the value it decremented reaches zero. So all we have to do is test to see if the Z flag is set, and if it is, jump out of the loop. From writing out all the bytes in my notes, I know that to escape the loop, I have to go to 0x0A, so that's where the JP Z takes us. Assemblers figure this kind of thing out for you automatically.

```
JP Z 0x0A
```

If we haven't jumped out of the loop, we go ahead and loop with a JP call, just like we did in the infinite loop.

```
JP 0x03
```

If we get to this point, then we've jumped out of the loop in the JP Z instruction earlier. So we go ahead and halt the Z80, which tells it to stop running the program.

```
HLT
```

Go ahead and stop the clock on the Z80 explorer (option 3) and initialize simulated memory again (option 6), then enter the ops above. When they're in and they look right, you should get output that looks like this:

```
        *** Mem-Sim Line Editor ***
Enter an address and data in HEX eg: 0x0000,0x76
"exit" to quit
"dump" to view memory.

>  Addr: 0x1 Data: 0x6 [OK]
>  Addr: 0x2 Data: 0x5 [OK]
>  Addr: 0x3 Data: 0x5 [OK]
>  Addr: 0x4 Data: 0xca [OK]
>  Addr: 0x5 Data: 0xa [OK]
>  Addr: 0x6 Data: 0x0 [OK]
>  Addr: 0x7 Data: 0xc3 [OK]
>  Addr: 0x8 Data: 0x3 [OK]
>  Addr: 0x9 Data: 0x0 [OK]
>  Addr: 0xa Data: 0x76 [OK]
```

Typing "dump" I get:

```
Dumping Simulated Memory


Address  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   Data (text)
---------------------------------------------------------------------------
0x0000| 00 06 05 05 ca 0a 00 c3 03 00 76 00 00 00 00 00  | ..........v.....
0x0010| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0020| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0030| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0040| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0050| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0060| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0070| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0080| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0090| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00a0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00b0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00c0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00d0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00e0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00f0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
Continue (y/n).
```

Looks good. Type n to quit, then go ahead and restart the clock. This will reset the Z80, but it won't erase memory. Set the clock to about 20Hz. When the reset finishes, you'll get output like this:

```
mem_read_ISR: Z80 Fetched Address: 0x0000       Data: 0x00
mem_read_ISR: Z80 Fetched Address: 0x0001       Data: 0x06
mem_read_ISR: Z80 Read Address: 0x0002          Data: 0x05
mem_read_ISR: Z80 Fetched Address: 0x0003       Data: 0x05
mem_read_ISR: Z80 Fetched Address: 0x0004       Data: 0xca
mem_read_ISR: Z80 Read Address: 0x0005          Data: 0x0a
mem_read_ISR: Z80 Read Address: 0x0006          Data: 0x00
mem_read_ISR: Z80 Fetched Address: 0x0007       Data: 0xc3
mem_read_ISR: Z80 Read Address: 0x0008          Data: 0x03
mem_read_ISR: Z80 Read Address: 0x0009          Data: 0x00
```

Well, B is not zero yet, so we've looped back to 0x0003. I've cut out three more iterations of this loop in the log.

```
mem_read_ISR: Z80 Fetched Address: 0x0003       Data: 0x05
mem_read_ISR: Z80 Fetched Address: 0x0004       Data: 0xca
mem_read_ISR: Z80 Read Address: 0x0005          Data: 0x0a
mem_read_ISR: Z80 Read Address: 0x0006          Data: 0x00
mem_read_ISR: Z80 Fetched Address: 0x000a       Data: 0x76
mem_read_ISR: Z80 Fetched Address: 0x000b       Data: 0x00
mem_read_ISR: Z80 Fetched Address: 0x000b       Data: 0x00
mem_read_ISR: Z80 Fetched Address: 0x000b       Data: 0x00
Clock Generator: Object Deleted.
```

Hey look. All of a sudden we don't get past 0x0006. That's where the second byte of the JP Z 0x000A instruction was. B must have hit 0, and the Zero flag got set. We jumped to 0x000A. Once at 0x000A, we picked up a HLT instruction. After that we stop advancing through memory addresses (at 0x000b) and sit there forever. I stopped the clock after that..

Program 3: Interrupts

Way back at the beginning of this chapter, I mentioned that I'd show you exactly what interrupts do with the Z80. This next program does that. We'll create essentially two machine language programs. One is a loop, and the other is an interrupt handler, which is essentially an ISR.

The two programs aren't connected together in any other way. If the /INT line on the Z80 never goes low, the interrupt handler is never executed.

There's a little bit of setup. Remember when we built the Z80 Explorer, I mentioned that like the ATmega, the Z80's interrupts are disabled by default. If you want to verify that, run the infinite loop again and press the button a few times. Nothing happens.

So the first instruction we'll give the Z80 is the Enable Interrupts instruction.

```
EI ; Enable interrupts
```

The Z80, unlike the 8080 from which it was copied, has three different interrupt modes. Mode zero behaves exactly the way the 8080's interrupts did. Complicated and hard to use. Mode two lets us pass a numeric value in on the data bus to tell the Z80 what the interrupt vector should be. We could do that, but the way the Z80 explorer is wired, the ATmega is completely unaware of the Z80's interrupt signal. We'll use mode 1.

When a mode 1 interrupt is triggered, two things happen. First, the PC (program counter) register is pushed onto the stack. Then the Program Counter is set to 0x0038, and we execute code from there on.

0x0038 is our interrupt vector. Keep this in mind.

Once our interrupt handler (ISR) executes a RETI instruction, the Z80 pops the old value of PC back off the stack into PC, and we resume executing code from where we left off when the interrupt was called.

Which in the case of this program is another infinite loop, a JP command that uses its own address as the target.

You'll see every step of this happen.

Here's the code:

```
        EI; Enable Interrupts
0x10 0xFB
        IM1; Set interrupts to mode 1
0x11 0xED
0x12 0x56
```

There's no direct way to load the stack pointer with an address. Instead, we'll load the 16 bit register pair HL with the address of the top of memory (0x2000), and set the stack pointer from that.

```
        LD HL 0x2000; Load the HL register with the highest memory address.
0x13 0x21
0x14 0x00
0x15 0x20
        LD SP HL; Load the stack pointer from HL.
0x16 0xF9
```

Here's the infinite loop.

```
        JP 0x17; Loop infinitely
0x17 0xC3
0x18 0x17
0x19 0x00
```

That ends the main section of this program. Next, we have the interrupt handler. We start by disabling interrupts. If we don't, another interrupt can interrupt our interrupt handler, which means that the value of PC stored on the stack may be wrong when we go to return.

Normally, you'd put the body of your interrupt handler between the DI and EI instructions. Since our handler doesn't do anything, there's nothing there. EI reenables interrupts.

```
        DI; Disable interrupts
0x38 0xF3
        EI; Enable Interrupts
0x39 0xFB
```

Finally, we return from the interrupt handler. This pops the old value of PC off the stack back into the PC, and sends the Z80 on its merry way executing our infinite loop. RETI is a two-byte instruction.

```
        RETI; Return from interrupt
0x3A 0xED
0x3B 0x4D
```

Go ahead and enter the opcodes for this program into memory, after reinitializing simulated memory and stopping the clock, of course.

You should get output like this:

```
        *** Mem-Sim Line Editor ***
Enter an address and data in HEX eg: 0x0000,0x76
"exit" to quit
"dump" to view memory

>  Addr: 0x38 Data: 0xf3 [OK]
Dumping Simulated Memory

Address  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   Data (text)
---------------------------------------------------------------------------
0x0000| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0010| fb ed 56 21 00 20 f9 c3 17 00 00 00 00 00 00 00  | ..V!. ..........
0x0020| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0030| 00 00 00 00 00 00 00 00 f3 fb ed 4d 00 00 00 00  | ...........M....
0x0040| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0050| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0060| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0070| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0080| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x0090| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00a0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00b0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00c0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00d0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00e0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
0x00f0| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
Continue (y/n)
```

Looks good. Let's go ahead and run it. Make sure your memory simulator ISRs are hooked up and start the clock. Note that while I trimmed the log so it starts when we fetch at address 0x0010, the Z80 will start at 0x0000, as always. It will fetch 16 NOPs before it

does anything. When I wrote this one, I hadn't yet put the m_write_enable mechanism in place, and memory below 0x10 was sometimes getting messed up.

```
mem_read_ISR: Z80 Fetched Address: 0x0010       Data: 0xfb
mem_read_ISR: Z80 Fetched Address: 0x0011       Data: 0xed
mem_read_ISR: Z80 Fetched Address: 0x0012       Data: 0x56
mem_read_ISR: Z80 Fetched Address: 0x0013       Data: 0x21
mem_read_ISR: Z80 Read Address: 0x0014          Data: 0x00
mem_read_ISR: Z80 Read Address: 0x0015          Data: 0x020
mem_read_ISR: Z80 Fetched Address: 0x0016       Data: 0xf9
mem_read_ISR: Z80 Fetched Address: 0x0017       Data: 0xc3
mem_read_ISR: Z80 Read Address: 0x0018          Data: 0x17
mem_read_ISR: Z80 Read Address: 0x0019          Data: 0x00
```

Okay, we've entered our infinite loop.

```
mem_read_ISR: Z80 Fetched Address: 0x0017     Data: 0xc3
mem_read_ISR: Z80 Read Address: 0x0018        Data: 0x17
mem_read_ISR: Z80 Read Address: 0x0019        Data: 0x00
```

Here's another cycle of it. There were quite a few more, but they are all the same. We pick the log up when I started holding down the /INT button. Note that we finish the entire instruction cycle for our JP before anything happens.

```
mem_read_ISR: Z80 Fetched Address: 0x0017     Data: 0xc3
mem_read_ISR: Z80 Read Address: 0x0018        Data: 0x17
mem_read_ISR: Z80 Read Address: 0x0019        Data: 0x00
```

Bang. We're writing to the stack. Twice. That's our program counter (PC register) getting saved.

```
mem_write_ISR: Z80 Wrote Address: 0x1fff      Data: 0x00
mem_write_ISR: Z80 Wrote Address: 0x1ffe      Data: 0x17
```

Having saved the program counter, we jump to 0x0038. No JP command required. It just happens.

```
mem_read_ISR: Z80 Fetched Address: 0x0038       Data: 0xf3
mem_read_ISR: Z80 Fetched Address: 0x0039       Data: 0xfb
mem_read_ISR: Z80 Fetched Address: 0x003a       Data: 0xed
mem_read_ISR: Z80 Fetched Address: 0x003b       Data: 0x4d
```

We're finished executing our tiny interrupt handler. Now we pop the old PC value off the stack back into PC.

```
mem_read_ISR: Z80 Read Address: 0x1ffe       Data: 0x17
mem_read_ISR: Z80 Read Address: 0x1fff       Data: 0x00
```

And here we are, back at our infinite loop at 0x0017

```
mem_read_ISR: Z80 Fetched Address: 0x0017     Data: 0xc3
mem_read_ISR: Z80 Read Address: 0x0018        Data: 0x17
mem_read_ISR: Z80 Read Address: 0x0019        Data: 0x00
```

Infinite loops are boring once the novelty wears off, so I stopped the clock.

```
Clock Generator: Object Deleted.
```

Program 4: Fun with the Stack

In the last program we touched the stack only briefly. The stack is one of those functions that is incredibly useful in the Z80. It's a staple of machine language (and assembly) programming, so we should talk about it a little more.

For those who haven't had formal computer programming classes, stacks are a data structure best visualized by plate dispensers (seriously, that's what they're called), those gadgets in cafeterias that hold stacks of plates. You can put a plate on the top of the dispenser. In programming we'd call this pushing a plate. When you put a second plate on the top, the spring in the dispenser compresses so the plates sink in and only the topmost plate is visible. This goes on for dozens of plates. When you want a plate out, you take one from the top, the springs expands, and the next plate becomes available. In software, this is called popping a plate. Push the plate on, pop the plate off. You only have access to the top of the stack, and it's the last plate you put in. Stacks are last-in, first-out (LIFO) data structures.

In the Z80, the stack is defined by the Stack Pointer (SP) register.

When you PUSH an 8 bit value onto the stack, the SP decrements and your value is put in memory at that address. When you push a 16 bit value onto the stack, the bytes are pushed onto the stack, but they're pushed there in reverse order, so the MSB is at the next highest address, and the LSB is at the next highest after that. If you were to read memory in the normal order, however, you'd find that little endian-ness has been respected. It's fairly arbitrary, since you normally take things off the stack with a POP, and POP will put the bytes in the order they were in when they were PUSHed.

POPping the stack copies the byte or bytes on the stack (a 16 bit push had better be followed by a 16 bit pop, or the stack will probably stop making sense).

If you've ever had programming teachers mention that "recursive programming is bad because you run out of stack space," consider that if you wind up using a CALL and RET to call your function, and it calls itself again and again, every call pushes more stuff onto the stack. Eventually, yes, you do run out of stack space (or in the Z80, your stack clobbers something important). If this hasn't happened to you, don't worry. It's an argument out of academic circles, mostly.

So now that we know what the stack is, and we've seen the mnemonics go by that make it work, let's do one more machine language program that demonstrates the stack. It'll put the stack pointer at 0x00FF, which is the end of the first 256 byte page of memory, so we can see what's in the stack in dumps, and it will show the last in/first out nature of the stack to give you a message

The program is called Fun with the Stack.

Here's the code. We start with an initialization section.

```
        LD HL 0x00FF; Set the HL register to 0x00FF.
0x10 0x21
0x11 0xFF
0x12 0x00
        LD SP HL; Load the SP from HL
0x13 0xF9
```

We'll have data starting at 0x0040. We'll use HL as a pointer to that address.

```
        LD HL 0x0040; Load HL with the address of the start of data.
0x14 0x21
0x15 0x40
0x16 0x00
```

There are two loops in this program, and we start the first one here, at 0x17. The first loop will load B with a byte of data, then copy that data to A, compare A with 0xFF to see if we're done. If not we'll add 16 to A and push the result onto the stack. That's why the data doesn't look like anything in the memory dump. Not only is it out of order, it's slightly encrypted. Then we'll increment HL and go back to 0x17.

```
        LD B, (HL); Load B with the contents of memory at address HL.
0x17 0x46
        LD A, B; Load the accumulator A with the value in B.
0x18 0x78
        CP 0xFF; Compare 0xFF to register A.
0x19 0xFE
0x1A 0XFF
        JP Z 0x0026; If the zero flag is set (A=0xFF) jump to 0x0026
0x1B 0xCA
0x1C 0x26
0x1D 0x00
        LD A, B; Copy B into A again, in case the accumulator changed.
0x1E 0x78
        ADD A 10;
0x1F 0xC6
0x20 0x10
        PUSH AF; Do a 16 bit push of AF, even though all we want is A.
0x21 0xF5
        INC HL; Increment HL
0x22 0x23
        JP 0x17; Jump to 0x0017 to repeat our loop.
0x23 0xC3
0x24 0x17
0x25 0x00
```

The first loop pushed our message onto the stack after decoding it. The second loop will pop it back off the stack into a new area of memory starting at 0x0050. We initialize the second loop by setting HL to 0x0050.

```
        LD HL 0x50; Load HL with 0x50, the start of our output area.
0x26 0x21
0x27 0x50
0x28 0x00
```

We're into loop two, and already doing something a little underhanded. We know that we have less than 16 items of data, so we know that the low order byte of HL is the only one that will ever change. So we index the loop on L. When L hits 0x58, the maximum address we've allowed for output, the loop will end. Elegant programming this is not.

```
        LD A, L; Load A with the low order byte of HL.
0x29 0x7D
        CP 0x57 ;Compare 0x58 and Register A.
0x2A 0xFE
0x2B 0x58
        JP Z 0x35; If the Z flag is set, jump to 0x35.
0x2C 0xCA
0x2D 0x35
0x2E 0x00
```

If we get here, we haven't escaped from our loop. So we go ahead and pop AF (16 bit pop), and write A to location HL in memory. Then increment HL and repeat.

```
        POP AF; Pop AF from the stack.
0x2F 0xF1
        LD(HL),A; Load memory location HL with the value of A.
0x30 0x77
        INC HL; Increment HL to the next memory location.
0x31 0x23
        JP 0x29; Jump to 0x0029 to repeat the loop.
0x32 0xC3
0x33 0x29
0x34 0x00
        HLT; Halt the CPU when we're done.
0x35 0x76
```

That's all the code, but we still need the data. It's short.

```
0x40 0x11
0x41 0x62
0x42 0x55
0x43 0x58
0x44 0x64
0x45 0x62
0x46 0x65
0x47 0x36
0x48 0xFF
```

Normally I'd put the log here, but that would spoil the secret message. You already know what will happen. Loop one will read the data from 0x40 to 0x48, add 10 to it, and push it on the stack. Loop two will iterate from 0x50 to 0x57, pop the data back off the stack and write it to that address of memory. Make sure to do a dump so you can see the secret message this program will leave in memory for you.

And enjoy.

Credit Where Credit Is Due

I must credit Sergey Malinov, who made the boards of the first computer that I built from scratch. At the time, his boards were part of the N8VEM project, which has since disbanded. His website is here: http://www.malinov.com/Home/sergeys-projects.

The Retrobrew community website, which sprang from the ashes of the N8VEM project, is slowly rebuilding, and has much of the old information here: http://retrobrewcomputers.org/doku.php.

Quinn Dunki's Blondihacks site and the long running Veronica computer build was a huge inspiration. Her site on that computer is here: http://quinndunki.com/blondihacks/?page_id=1761. Above all, it was her website that first made it clear to me how the basic memory fetch/instruction cycle worked, and demonstrated free running. It was such an effective demonstration, I had to include it.

Dave Jones's EEV blog provided a lot of basic technological knowhow, and some attitude: http://www.eevblog.com/. While there was no soldering in the projects in this book, I have found no better soldering tutorials than his.

I must also credit Jeff Duntemann, http://duntemann.com/, for advice, knowhow, experience, good stories on the subject, encouragement, and an excellent book on assembly language. He was there when the Z80 was king. He's also my closest friend and fellow science fiction author.

Finally, I'll sneak in a credit for Marcia Bednarcyk. Nerd, software engineer, wonderful human being, and for more than 20 years, my wife.

Further!

We've come a long way. Starting with Arduino skills, we built the Cestino. We learned ports and 8 bit binary. We climbed the tree from transistors all the way to microprocessors, and now we're writing hand-assembled machine code for a Z80 and watching it run on something we built from scratch.

Neat, huh?

So where do we go from here?

For the Z80 explorer, there are lots of answers. The first and most obvious would be: some way for the memory editor to save files so we don't have to retype everything every time we reboot. Believe me, I thought longingly of that functionality while I was developing the programs in this section. The Atmega actually can use part of its flash to store external data.

It might be nice to build a proper assembler into the editor, and perhaps wrap the whole sketch into it with a command parser, and use that instead of the menu. Apple II did it that way. You could escape to a machine language monitor (which is what we're talking about) if you knew the right keystrokes. No external software required. Sketches can always be improved.

Obviously, there are lots more assembly programs you could write for the Z80. A Z80 with 8k of RAM can do quite a lot. The original Microsoft Basic ran on an 8080 or a Z80 with 4k, and had a little space left to write and run programs; 8k would leave a lot more. Where you get that code and how you get it into the Z80 is an exercise I'll leave to the reader.

But this book is about hardware, mostly. What could you do with the hardware of the Z80 explorer to make it better? First and foremost, add real RAM. You can get 512k of static ram on a single IC from Mouser and the like (Alliance AS6c4008, 4mega-bit 512k8 through-hole SRAM) for less than five dollars U.S. By now you can probably read the datasheet and figure out how to connect the Z80 to it. Having freed up two full ports on the Cestino (at the expense of dramatically complicating how you get data from the Cestino into memory) you could add an SD card on the SPI interface of the ATmega1284P for storage. (Bear in mind that SD cards are 3.3v animals. Adafruit has good information on this kind of thing here: `https://learn.adafruit.com/adafruit-micro-sd-breakout-board-card-tutorial/look-out`.

If you took the memory ISRs out of the picture, you could turn the clock up to the maximum your Z80 is rated for, write a few assembly language routines (an interrupt handler and a driver, basically) to let the Z80 communicate with the ATmega and tell *it* what to do.

At some point in that project, you might want to get away from breadboards and into a method of construction that is more mechanically stable. I can't tell you the number of times a project just stopped working (usually as a deadline loomed) during the writing of this book, and it turned out that a wire was loose. The breadboards in the photos are inexpensive, and pretty badly worn, at this point. Breadboards have their limits electronically too, and as I said in the beginning, we're pushing our luck a little bit running 20MHz signals through it. We get away with it, but it might not always be so. We might want to add a proper power supply at that point too. Our hotwired ATX power supply from the ATA explorer would do that nicely. A little tinkering and the hotwiring could be switched. Off switches are always nice.

So let's see. More software, speed, real RAM, storage, we can use the ATmega for RS232 communications, power supply … I don't think we can call this the Z80 explorer anymore. I think it's become a computer. Don't believe me? Consider.

Given a series of assembly programs for accessing the peripherals in a standardized fashion called a Basic Input Output System. (Ever wonder what BIOS stands for and what it does? Now you know.) you could run CP/M on it, using your desktop for a terminal (which CP/M does well) which would open a whole universe of software (much of it approaching 40 years old and free for the asking.) This is what made the Z80 famous. It was easy (by comparison) to connect it to a small set of peripherals and come up with a computer that could do useful work. Actually doing it is way beyond the scope of this chapter. We'd have to cover construction, programming and get into assembly/machine language in all its depth. It'd be a book unto itself. Hmm.

What could you do with a Z80 computer and 40 year old software today? Well, a few years ago, I mail-ordered the boards for a Zeta computer from Sergei Malinov. You can find the newer version of those same boards here: `http://www.malinov.com/Home/sergeys-projects/zeta-sbc-v2`, although whether he's got any left at this point is unclear. The Zeta is essentially what we've been talking about: a Z80, some SRAM, some peripheral chips (it talks to 3.5 inch floppy drives) and an EEPROM with CP/M, the BIOS, and a bunch of applications on it.

The boards arrived in an envelope. With the Internet as my shopping mall, I sourced the parts, and over a period of a couple months (plus a few more to debug some construction errors on my part) I had a working Z80 computer. Since it was part of a larger family of computers, someone else had already written the BIOS for it. All I had to do is download the firmware and burn it to my EEPROM.

I mention this, because it is this computer and Wordstar, rather than my monster Mac, that I'm typing this last section on.

Once you know some electronics, you can learn more. We've made some amazing things out of junk, just for fun. We live in the best of times to learn electronics. Never has more information been available on the subject, much of it coming out from under patents, and most of it available online at the click of a mouse. Never before has the hobbyist electronics scene been so large and so well embraced by the mainstream. Never, ever, have the parts been so cheap. In 1980, when the Z80 was a front-line computer, 16k of ram cost over $300 in mighty 1980 dollars. Multiply by about four to get their equivalent price in 2016. If you fried one of those (early CMOS dynamic RAM ICs were horrifically sensitive to static), it hurt. Today, the parts are free if they're in the junk box. Even if they're not, they're shockingly cheap new. It's a good time to learn electronics for other reasons, too.

Although the industry strives to sell us magical and wonderful gadgets, more and more of us know, and rejoice in the knowledge, that there is no magic, just people with skills. Are electronics complicated? Sure. But look how far we've come just in this book. From one LED to the rudiments of a personal computer.

I'm not an engineer. I'm a former system-admin, technical support guy who normally writes science fiction. You're holding in your hands 99.9 percent of all the assembly code I've ever written, and all the hand-assembled machine code. These projects, from building the Cestino to the EPROM/Flash Explorer, to (and most especially) the Z80 explorer, have been triumphs for me. I hope they are for you, too. That sense of triumph is what sets off the next project, and gets the juices running for the one after that. What can I do with this? Can I add more to it? Can I make it do this other thing? Where can I go with this?

You know the answer already.

Go Further!

Index

A

Archetecture
Harvard, 32–34
Princeton, 33
Von Neumann, 33
Arduino
board, 1, 2, 59–62, 68
bootloader, 10, 30, 49, 52–53, 56–60, 62–63, 66, 69, 80, 301
core, 29–30, 52-53, 56–57, 59, 73, 76, 79, 91, 93–94, 138, 155, 246, 277–279, 293, 301, 346
IDE, 58, 326
sketch, 30, 59–60, 64, 73, 95, 279, 304
software, 10, 16, 29–30, 56, 59–60, 94, 301
Assembly
machine language, 313, 381–394, 396
ATA
ATA1, 208
IDE, 24, 204, 206, 208
PATA, 204–210, 213–215, 217, 219–220, 269
PIO, 208
SATA, 205, 207–208, 244, 270

B

Binary
bit, 84, 87, 94, 129, 142, 164, 395
byte, 83–84, 87, 91, 94, 164, 207
counting, 82–83, 129, 142
shifting, 87–88
twiddling, 84, 88
word, 83–84

C

Cestino, 1–4, 9, 11, 14,–15, 17–19, 24–26, 29–53, 55–56, 58–67, 69–71, 79–80
CISC, 34, 314
Clock
ATmega1284P, 34–36, 45, 59, 63, 67, 70–71, 132, 157, 280–281, 328
Z80, 27, 304–305, 308, 327–328, 330, 333, 336–337, 345, 348–350, 388–397
Compiler
c, 63, 73, 77, 79, 91, 94–97, 99, 115, 122, 138, 163, 165, 170–172, 174, 182–183, 277– 278, 288, 311, 323, 325–326
c++, 73–75, 77, 79, 88–89, 91, 94, 101, 108, 120–122, 137, 140, 144, 147, 156–158, 163, 171–173, 177, 185, 188, 190, 197–198, 230–231, 234, 240, 245, 252, 255, 259, 263, 277–278, 288, 304, 311, 321–326, 337, 340, 342–343, 345–347, 362–363, 366, 368–370, 381
class, 73–75
object, 73
prototype, 326
Complex instruction set computing (CISC), 34, 314

D

Datasheet
ATmega1284P, 11, 30, 38, 131, 133, 277, 301
74xx00, 11, 126, 131, 134, 152

© James R. Strickland 2016
J.R. Strickland, *Junk Box Arduino*, DOI 10.1007/978-1-4842-1425-1

E

Endian, 90, 170, 173, 209, 224, 231–233, 235, 238, 241, 317, 340, 354, 382

F

Floating, 35, 117, 122, 155–157, 164, 166, 169, 239, 305, 311, 314, 337

G

Gibibyte, 130, 207–208

H

Hexadecimal, 163, 172, 343

I, J

IC
- 2716, 23, 165
- 2732
- 2764, 23, 165
- 27128, 163, 165–166, 170–172, 181–182, 200
- ATmega1284P
- 27Cxx, 163
- 20MHz full can TTL oscillator, 18, 127
- SST39SF020A, 23, 163
- 74xx00, 11, 22, 127, 131, 135–136, 148, 151, 153
- 74xx92, 127, 141–142, 145, 159
- Z84C00x0, 305

In-System Programmer (ISP), 55, 58–60, 80
Interrupt
- ISR, 277–279, 284, 288–290, 292, 313, 330–331, 350, 352–353, 355, 382, 384, 387–388, 390–391
- service routine, 277–279, 284, 288–290
- vector, 277–279, 289–291, 307, 328–329, 388

K

Kilobyte, 161, 165–166, 172, 207

L

Larson, 20, 81, 96–100
Logic probe, 22, 149–160, 169, 179, 210, 274–276, 304, 309

M, N, O

Magic, 29, 304, 396
Mebibyte, 206–208

P, Q

Part
- capacitor, 14–15, 22–23, 30, 151
- junk, 1, 3, 6, 12, 16, 18, 21, 304
- LED, 16, 18, 20, 26, 43–45, 273, 275–276
- pin headers, 9, 24, 31, 37, 204, 218
- resistor, 12–14
- speaker, 16, 20, 55, 72, 76, 78–79
- tactile button, 16, 19, 26, 30, 36, 40, 305, 329
- transistor, 21, 26, 34, 46, 103–105, 108–126, 148, 273

R

RAM
- address, 35, 171, 183, 306, 308, 310, 313–314, 316, 320, 323–324, 331, 340, 387
- Dennard, 124, 308
- DRAM, 124, 307–308, 320
- SRAM, 32–33, 308, 395–396

RISC, 32–35, 149, 314
ROM
- address, 166, 169–175, 182–183
- EEPROM, 16, 22–23, 32–33, 58, 163–166, 200, 396
- EPROM, 10, 16, 22–23, 161–201, 207–209, 307, 397
- flash, 23, 161–201, 307, 397
- Frohman, 164

RS232
- ttl232, 9, 24, 204

S

Science
- atom, 42, 106–107
- band, 14, 105–107, 201

charge, 14, 43, 107–108, 164–166
conduct, 14, 40, 49, 96, 107–108, 118, 126, 164
current, 11–12, 14, 18, 20, 25, 41–44, 47, 56, 69–70, 76, 81–82, 95–98, 101, 108–113, 117, 125–129, 131–132, 151–155, 164, 201, 205, 212, 275–276, 286, 309, 333–335, 337, 353
Dalton, 106
diode, 43–44, 107–108, 110, 117, 126–127, 165, 285
dope, 107
electron, 42, 106–108, 123, 164–166, 169
energy, 12–14, 20, 42, 106–107, 111, 216
Heisenberg, 106
Kirchoff, 110–113
Ohm, 6, 12, 14, 41–43, 47, 107, 109–113, 285
orbital, 42, 106–107, 123
physics, 34, 42, 105, 107–108, 110, 123–124, 148
probability, 106
quantum, 105–107, 123, 165–166, 304
resistance, 6, 12–13, 34, 41–43, 47, 49, 76, 108–113, 115–121, 124, 131–132, 153, 179, 276, 291
semiconductor, 105–109, 113, 133–134, 141
silicon, 33, 107, 109, 126, 165, 308
voltage divider, 110–116, 118, 124, 138
Software abstraction, 57
Speed, 42, 63, 79, 98, 107, 122, 128, 132, 138, 148–150, 153, 157, 159, 168, 205, 209, 275–276, 279, 302, 304–305, 314, 325, 327–328, 330, 332, 349, 381, 396
Stack, 129, 130, 305, 313, 381, 388–394
Standard
ANSI, 208, 270
ATA, 208, 244, 269
JEDEC, 163, 165, 168, 170, 172, 181–182, 201, 207
TTL, 128
Supplier
Adafruit, 2–3, 5–6, 8–11, 21, 24, 29, 37, 59, 204, 395
Atmel, 11, 18, 29–32, 34, 46, 58, 64, 124, 133, 281
Digikey, 3, 8, 16, 21–22, 27, 181
Futurelec, 18, 22–23, 142, 163
Mouser, 3, 8, 10, 14, 16, 18, 21–22, 27, 163, 181, 395
Radio Shack, 6, 14, 173
Sparkfun, 6, 8, 10, 21

T, U, V

Timer
counter, 279–285
timer/counter, 279–284, 287, 289–292, 294, 301, 330, 336, 338, 349
Tool
breadboard, 2–3, 6, 9, 10, 14, 17, 20, 21, 26, 29–30, 35–40, 43, 52, 58, 60–62, 66, 71, 76, 115, 127, 130, 132, 136, 149, 153–154, 163, 166, 168–169, 171, 200, 204, 216, 218–220, 236, 275, 286, 305, 327, 329, 395
calculator, 11, 276
FTDI, 8–10
hookup wire, 4–5, 30
jumpers, 3–4, 38, 170, 205
multimeter, 5–6, 36, 38, 43–44, 70, 123, 157, 216–217
screwdriver, 6–7, 16, 36, 71
universal programmer, 10, 23, 58–59, 63
wire cutter, 7–8
wire stripper, 7–8
Transistor
base, 103–126, 163, 207, 273
collector, 103–124, 127, 129, 286
emitter, 103–124, 126
TTL
ic, 133–134
interface, 8–9

W, X, Y

Weasel, 207

Z

Z80
D780, 27
instruction, 26, 35, 304–309, 311–320, 344, 353, 381–382, 384–385, 387–389, 394
Z84C0020, 305