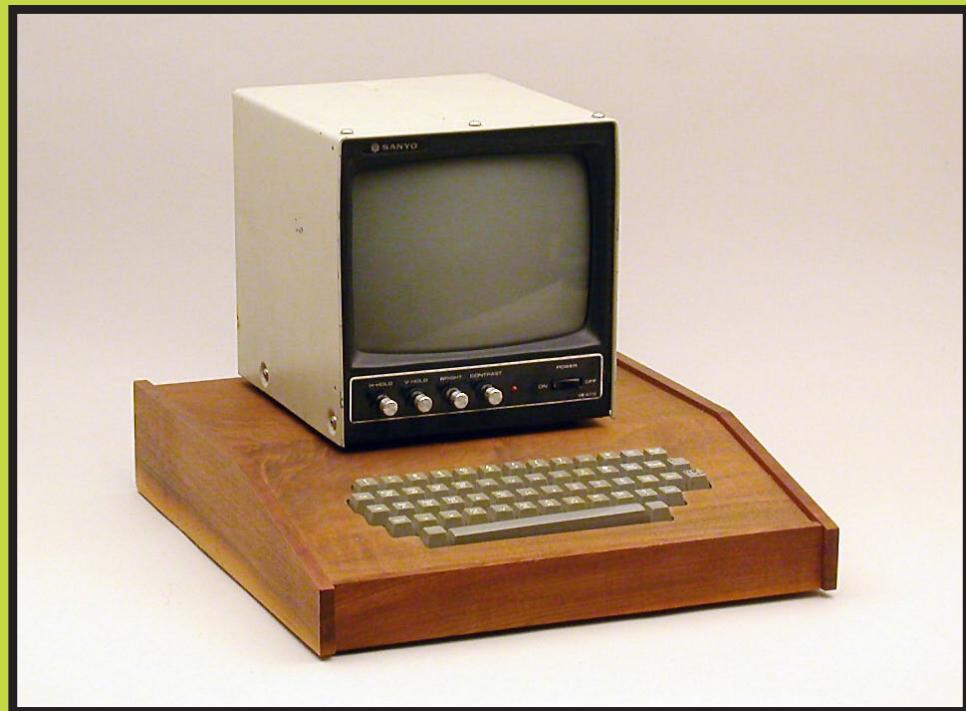


A SCHNITZ TECHNOLOGY BOOK

Apple I Replica Creation

Back to the Garage



Tom Owad, applefritter.com

with foreword by **Steve Wozniak**, co-founder of Apple

A SCHNITZ TECHNOLOGY BOOK

Apple I Replica Creation

Back to the Garage



Tom Owad

www.applefritter.com

The publisher, author(s), and any other persons involved in the creation, design, preparation or distribution of this book cannot guarantee or warrant any results obtained from this book.

This book is sold as-is and there is no guarantee of any kind, expressed or implied, regarding the book or its contents. It is sold without warranty, and in no way or event shall the creators of this book be held liable for damages, loss of profit or savings, or any other incidental or consequential damages arising from this book or its contents. As some states may not allow the exclusion or limitation of liability, the preceding statements may not apply to you. You may have other legal rights, which vary from state to state.

Always take reasonable care when using tools and electrical equipment, and wear proper safety gear when applicable.

Brands and product names mentioned in this book are trademarks, service marks, or copyrights of their respective owners.

Published by
SCHNITZ TECHNOLOGY
1803 Walnut Bottom Road
Newville, PA 17241

<http://www.schnitz.com>
mail@schnitz.com

APPLE I REPLICA CREATION: BACK TO THE GARAGE
Second Edition

Copyright © 2012 Tom Owad. All rights reserved. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored and/or executed in a computer system, but they may not be reproduced for publication.

Designed in the United States of America.

Technical editor: John Greco
Copy editor/designer: Colin Wirth

All photographs by Tom Owad, except for:
Figure 1.1 – Achim Breidenbach
Figures 1.2, 1.3, 1.5–1.10 – Joe Torzewski
Figure 1.4 – SWTPC
Figure 1.11 – Larry Nelson
Figures 1.12, 1.13 – Ray Borrill
Figures 1.14, 1.15 – Liza Loop
Figures 1.16, 1.17 and Cover Image – Steve Fish
Figures E.1–E.40 – Joe Grand

Foreword written by Steve Wozniak
Appendix E written by Joe Grand

Acknowledgements

Much thanks goes to my parents, John and Cindy Owad, who always supported by interests but nevertheless are happy that my racks of PDP-11 equipment no longer fill their living room.

Dr. John Greco, my technical editor, has offered great guidance and been of immeasurable help in structuring and error-checking this book. Sandra Veresink deserves thanks for her editing and stylistic help. Much thanks also goes to Colin Wirth, who copy edited and designed this second edition. Without his help, I never would have gotten it done.

Vince Briel designed the Replica I circuit described in this book and has generously shared his design with us. Sarah McMenomy wrote the plot for the BASIC game in Chapter 5.

John and George Soluk at VAMP Inc. provided valuable assistance in moving our designs to McCAD EDS.

Joe Torzewski founded the Apple I Owners Club back in 1977 and kept it alive all these years. And thanks to Steve Wozniak, who started it all.

Supplemental Software

The software, circuit board designs, Apple I programs and other material referenced in this book is provided as part of the Supplemental Software package, which can be downloaded from <http://www.applefritter.com/replica>.

McCad EDS SE500 by VAMP, Inc. is available from the Mac App Store or <http://www.mccad.com>.

Contents

Foreword by Steve Wozniak	15
Chapter 1 Apple I History	19
The Apple I	20
Apple I Owners Club	25
Pioneer Interviews	27
Joe Torzewski	27
Larry Nelson	30
Ray Borrill	32
Liza Loop	35
Steve Fish	38
Allen Baum	41
Summary	43
Chapter 2 Tools and Materials	45
Essential Tools and Supplies	46
Multimeter	46
Logic Probe	48
Breadboard	49
Wire-Wrap Tools	50
Soldering Iron & Materials	51
Power Supply	54
TTL Chips	56
Circuit Boards and Software Tools	57
Chip Pullers and Straighteners	57
Keyboard and Monitor	59
Ambiance	60
Chapter 3 Digital Logic	63
Breadboarding	64
Electricity	67
Voltage and Current	67
Resistors and Diodes	69
Capacitors	72
Gates	72
AND	73
Inverter, NAND	76

OR, NOR	79
XOR	82
Circuits with Algebra	84
Logic Expressions	84
DeMorgan's Laws	86
Boolean Algebra	86
All You Need is NAND	87
Latches and Flip-Flops	89
SR Latch	89
Flip-Flop	91
What is Data?	93
Counting in Binary and Hexadecimal	93
Bytes	96
ASCII and the Alphabet	97
A Few More Chips	100
Shift Register	100
Buffer and Tri-State Buffer	101
Encoders and Decoders	103
Summary	106
 Chapter 4 Building the Replica I	107
Learning to Solder	108
Assembling the Replica I	110
Parts List	111
Resistors	112
Diodes and Bypass Capacitors	113
Buttons	114
Sockets	114
ASCII Keyboard Socket	115
Capacitors	116
Header and Jumper	117
Crystals	117
Connectors	118
Finishing Assembly	119
Serial I/O Board	121
Using McCAD EDS SE500	123
McCADC Schematics SE500	124
McCADC PCB-ST SE500	125
 Chapter 5 Programming in BASIC	127
Setting Up BASIC	128
Hello World	128
The PRINT Command	128
The TAB Command	129
The GOTO Command	130
Welcome, User (Input, Variables, and Strings)	130
Calculator (Math)	131
Count Down (FOR/NEXT)	133
Some (IF/THEN)	134

Expressions	134
The GOSUB Command	136
Arrays	138
Strings, In Depth	140
Substrings	140
The LEN Function	140
Appending Strings	141
Conditionals	141
Sample String Program	142
PEEK and POKE	143
The CALL Command	145
<i>Richard III</i> : Interactive Fiction	147
Walkthrough	147
Structure	150
Variables	151
Skeleton	152
Initialization	153
Introduction and Conclusion Subroutines	153
PRINT Room Subroutine	155
Outside Tower	156
Bottom of the Tower	157
Middle of the Tower	158
Top of the Tower	159
<i>Richard III</i> Code	160
Summary	167
Chapter 6 Programming in Assembly	169
Using the Monitor	170
Set Up the Assembler	171
Registers	173
Hello World	173
TV Typewriter	176
X and Y	178
Memory Addressing	179
Accumulator: <i>A</i>	180
Implied: <i>i</i>	180
Immediate: #	180
Absolute: <i>a</i>	181
Zero Page: <i>zp</i>	181
Relative: <i>r</i>	181
Absolute Indexed with X: <i>a,x</i>	181
Absolute Indexed with Y: <i>a,y</i>	182
Zero Page Indexed with X: <i>zp,x</i>	182
Zero Page Indexed with Y: <i>zp,y</i>	182
Absolute Indexed Indirect: (<i>a,x</i>)	182
Zero Page Indexed Indirect: (<i>zp,x</i>)	183
Zero Page Indirect Indexed with Y: (<i>zp,y</i>)	183
Interacting with Memory	184
Printing Strings	185

String Subroutine	187
Bit Representation	189
Using the Stack	191
Bit Manipulation	195
Math	198
Summary	201
Chapter 7 Understanding the Apple I	203
Overview	203
The Data Bus	205
The Address Bus	206
The Clock	208
The Processor	209
Pins and Descriptions	210
Address Bus (A0-A15)	211
Clock (ϕ_0, ϕ_1, ϕ_2)	211
Data Bus (D0-D8)	211
Interrupt Request (IRQ)	211
No Connection (NC)	211
Non-Maskable Interrupt (NMI)	211
Ready (RDY)	211
Reset (RES)	212
Read/Write (R/W)	212
Set Overflow Flag (SO)	212
Sync	212
Voltage Common Collector (Vcc)	212
Voltage Source (Vss)	212
Registers	213
The Accumulator	214
Index Registers X and Y	214
Program Counter (PC)	214
Stack Pointer	214
Processor Status Register	215
The Arithmetic and Logic Unit	215
The Stack	215
Memory	216
Where is It?	216
Implementing 8 KB RAM	219
Address Lines (A0-A12)	219
Data Lines (D0-D7)	219
Chip Enable (/CE1, CE2)	219
Write Enable (/WE)	219
Output Enable (/OE)	220
Implementing 32 KB RAM	222
Implementing the EPROM	223
Implementing the Expanded ROM	224
Wiring the 74LS138	225
Wiring the 28c64	225
I/O with the 6821	226

DDR Access	230
CA1 (CB1) Control	230
CA2 (CB2) Control	231
IRA Interrupt Flags	231
Keyboard	231
Video	231
Keyboard Input	232
Video Output	234
Summary	236
 Appendix A ASCII Code Chart	237
 Appendix B Operation Codes and Status Register	239
 Appendix C Operation Code Matrix	243
 Appendix D Instructions by Category	247
Load and Store	247
LDA — Load Accumulator with Memory	247
LDX — Load Index X with Memory	247
LDY — Load Index Y with Memory	248
STA — Store Accumulator in Memory	248
STX — Store Index X in Memory	248
STY — Store Index Y in Memory	249
Arithmetic	249
ADC — Add Memory to Accumulator with Carry	249
SBC — Subtract Memory from Accumulator with Borrow	250
Increment and Decrement	250
INC — Increment Memory by One	250
INX — Increment Index X by One	251
INY — Increment Index Y by One	251
DEC — Decrement Memory by One	251
DEX — Decrement Index X by One	251
DEY — Decrement Index Y by One	252
Shift and Rotate	252
ASL — Accumulator Shift Left One Bit	252
LSR — Logical Shift Right One Bit	252
ROL — Rotate Left One Bit	253
ROR — Rotate Right One Bit	253
Logic	254
AND — AND Memory with Accumulator	254
ORA — OR Memory with Accumulator	254
EOR — Exclusive-OR Memory with Accumulator	255
Compare and Test Bit	255
CMP — Compare Memory and Accumulator	255
CPX — Compare Memory and Index X	256
CPY — Compare Memory and Index Y	256
BIT — Test Bits in Memory with Accumulator	256
Branch	257

BCC – Branch on Carry Clear	257
BCS – Branch on Carry Set	257
BEQ – Branch on Result Zero	257
BMI – Branch on Result Minus	257
BNE – Branch on Result not Zero	258
BPL – Branch on Result Plus	258
BVC – Branch on Overflow Clear	258
BVS – Branch on Overflow Set	258
Transfer	259
TAX – Transfer Accumulator to Index X	259
TXA – Transfer Index X to Accumulator	259
TAY – Transfer Accumulator to Index Y	259
TYA – Transfer Index Y to Accumulator	259
TSX – Transfer Stack Pointer to Index X	260
TXS – Transfer Index X to Stack Register	260
Stack	260
PHA – Push Accumulator on Stack	260
PLA – Pull Accumulator from Stack	260
PHP – Push Processor Status on Stack	261
PLP – Pull Processor Status from Stack	261
Subroutines and Jump	261
JMP – Jump to New Location	261
JSR – Jump to New Location Saving Return Address	261
RTS – Return from Subroutine	262
RTI – Return from Interrupt	262
Set and Clear	262
SEC – Set Carry Flag	262
SED – Set Decimal Mode	262
SEI – Set Interrupt Disable Status	263
CLC – Clear Carry Flag	263
CLD – Clear Decimal Mode	263
CLI – Clear Interrupt Disable Bit	263
CLV – Clear Overflow Flag	263
Miscellaneous	264
NOP – No Operation	264
BRK – Break	264
Appendix E Electrical Engineering Basics	265
Fundamentals	265
Bits, Bytes, and Nibbles	265
Reading Schematics	270
Voltage, Current, and Resistance	272
Direct Current and Alternating Current	272
Resistance	273
Ohm's Law	274
Basic Device Theory	274
Resistors	275
Capacitors	277
Diodes	281

Transistors	283
Integrated Circuits	286
Microprocessors and Embedded Systems	288
Soldering Techniques	289
Hands-On Example: Soldering a Resistor to a Circuit Board	290
Desoldering Tips	293
Hands-On Example: SMD Removal Using ChipQuik	294
Common Engineering Mistakes	298
Web Links and Other Resources	299
General Electrical Engineering Books	299
Electrical Engineering Web Sites	299
Data Sheets and Component Information	300
Major Electronic Component and Parts Distributors	300
Obsolete and Hard-to-Find Component Distributors	301
Index	303

Foreword

Steve Wozniak

It was in sixth grade that I got my first transistor radio. That may have been the most important gift of my life. I found out about music and I knew that it was a big thing in my life, all from this small hand held, 10-transistor radio. I could sleep with music all night long, and did so that entire year. My father was working on missile guidance technology for military projects around this time, and showed me how tightly they could pack transistors and told me how chips were coming with more than one transistor on them, for even more space and weight savings. I commented that they were designing these things to make better and smaller radios and he told me "no," that they were being designed for military and commercial uses and that only after a long time they'd filter down to consumer products that we used in our lives and homes. That thought bummed me out, that normal people in their homes wanting good products were not driving technology.

My introduction to electronics was accidental. I lived in a new community surrounded by orchards. It was Santa Clara Valley but is now called Silicon Valley. I had lots of local electronics-kid friends and we'd do gardening to be paid in parts like resistors. I read a book about a ham radio operator solving a kidnapping and found that anyone of any age could be a ham. You didn't have to be older as with car licenses. So by sixth grade I had my ham license and was learning electronics, back in the days of tubes.

One day I stumbled across a journal in a hall closet at home. It had articles about early computing projects. This was around 1960 when computers were unknown to mortals and filled entire rooms. I read articles on strange storage devices and Boolean algebra and logic gates and how they could be combined. What amazed me most was that none of this stuff was too hard for a fifth-grader to follow. You didn't need advanced math to understand binary arithmetic or logic gates. I became fascinated by what I'd stumbled onto. Based on what I'd read and what my father could teach me I had a couple of large science fair projects in sixth and eighth grades with hundreds of transistors and diodes and all doing logic. One played tic-tac-toe, which I thought was a game of logic (I now know that it's more a game of psychology). The other project added and subtracted 10-bit numbers. I had no idea that I was on the right path to learning how to design computers, or that there were jobs in this field. In fact, I was on a search to find out what a computer was. I myself was too shy to ask or read to get such answers.

At another science fair I stumbled onto a project with a stepping motor advancing a pointer to switches where you could set up operations to be done. This motor could loop, or back up, depending on results. This was the first time in my life that I learned computers had instructions to do things like my adder/subtractor did, and that it could

repeat those instructions.

We had no computer at our high school, but because I was so advanced by then in electronics my teacher, John C. McCullum, a great teacher, arranged for me to learn more by programming a computer at a company in Sunnyvale. In those days just the word "computer" brought looks of awe. I came to love all that I could do in Fortran that year. I also stumbled upon a manual at Sylvania titled "The Small Computer Handbook." It described a real minicomputer, the PDP-8. The engineers let me have that manual and it changed my life forever.

Now I had a manual describing a real computer, its architecture and instructions. I also had some chip catalogs, back when you could only get one gate on a chip. And I had my knowledge of how to combine gates into things like adders. So for the rest of high school my favorite pastime became designing minicomputers on paper. When a single chip cost \$50 (maybe that's \$500 today), you don't have the parts to build your own computer. I found how to get manuals describing various minicomputers from Digital, HP, Varian, DEC, and others. I designed my own versions of these machines over and over. As chips got better, my designs took fewer and fewer parts. My goal became to beat my prior design whenever I redesigned the same computer again. I got very skilled at digital design this way. I had no endpoint; you can always think of one more way to save another chip in a design.

I had no friends, parents or teachers doing this with me, or even aware that I was doing it. My computer designs were weekend projects in my room with my door shut. I sometimes would go late into the night and it helped to drink Cokes. When the Data General NOVA was introduced, and I finally got around to designing this computer, I found that it took half as many chips as my other designs due to its unique architecture. That changed my life a lot too. If you design a computer architecture based on what chips are available, it can save a lot of parts.

All these minicomputers had front rectangular plates full of switches and lights, and they were intimidating and commercial-looking and belonged in racks on factory floors. But if they could run Fortran, that could even enable a mere mortal to use them to play games and solve problems. I knew what I needed and told my father that someday I'd have a 4K NOVA computer. When he pointed out that it cost as much as a house I said I'd live in an apartment, but I'd have my own computer someday.

I could design computers but couldn't afford the parts to build them.

After some college, I wound up designing chips for HP's early scientific calculators. I worked on a lot of interesting side projects but lost track of minicomputers and overlooked the introduction and advancement of microprocessors. In 1974 I saw my first Pong game in a bowling alley. As I stared, mesmerized, at the screen, it occurred to me that I could build a Pong. I could never afford an output device, but like everyone else, I had a color TV at home. There was no video-in jack, but in those open days you got schematics with your TV. I knew TV signals and how they worked from high school electronics, and I knew digital design, so it wasn't long before I built my own Pong, using very few chips.

Later in 1974 I visited an old friend, famous phone phreak John Draper, also known as Captain Crunch. He was typing on a Teletype, an input/output device that I myself could never afford. He was playing chess with a computer in Boston. Then he showed me how he could have the Teletype type out a list of computers and he could switch to one at Berkeley. It was the early ARPANet and you dialed into a number at Stanford using a modem to get onto it.

I absolutely had to do this thing that made you like a king, doing what nobody else could do. I had just built my Pong and knew how to get signals onto a TV screen. So I designed and built a terminal that could put letters and numbers on my home TV. I had to buy a keyboard for \$60, and that was very expensive for me but was barely affordable. I did indeed access the ARPANet a few times, but wasn't as interested in using those computers as I was just knowing that I could reach out to such distant places. It was like ham radio, and also like phone phreaking, in that sense.

So a \$60 upper-case-only keyboard and my Sears color TV gave me the input/output I needed at an affordable cost. Steve Jobs said, "Let's sell it," and we did sell a number to a local timeshare outfit called Call Computer.

In 1975, the Homebrew Computer Club started. I got tricked into going. I never would have gone to something based on microprocessors because I didn't know what a microprocessor was. I didn't know that you could buy an enhanced microprocessor, a CPU, in a case called the Altair. I didn't know that you could add enough to this processor to make it a computer. I was told that a new club was being formed for people with terminals and the like. I figured it would be a great chance for a shy guy to show off with his own video terminal based on very cheap chips.

I got scared the first night at the Homebrew Computer Club. Everyone knew what was going on with these new affordable "computers." I took home a microprocessor data sheet and to my surprise this chip was the CPU that I'd grown up designing in high school. I was back in business. I saw that night that soon I'd design or buy a 4 KB computer and run Fortran for myself finally.

Rather than design a computer from scratch, I saw a shortcut. I could take my terminal with human input and output, and combine it with a microprocessor and some RAM. I'd actually built a computer of my own design five years earlier, one equivalent to the Altair with 256 bytes of RAM and switches and lights on a front panel. I didn't want to do that again. I didn't need to wire dozens of switches and have the chips to get them to the memory of a computer. That took too many parts, and chips, and money. Our calculators at HP had ROMs, and they ran a program when you turned them on that waited for a user to press human buttons, and then the program did the right things. I saw that I could write a short program that monitored the keyboard for input to do what the old front panels had done. I called this program a monitor. It took 256 bytes, which was 2 PROM chips in 1975.

I first got working what was to be the Apple I, using static 1K RAM chips. But it took 32 of these chips to have 4 KB of RAM, enough for a computer language. The 4K dynamic RAMs were just being introduced, the first RAMs to be cheaper than magnetic core

memories. I bought some from a fellow at our club and got them working on my computer. I had to design some refresh circuitry, but overall I saved so many chips and dollars that it was the right way to go. If you look back, you'll find that every single other hobby computer back then used static RAMs because they involved less design work.

I passed out schematics and code listings freely at our club, hoping that others would now be able to build their own computers. It took another four months for me to write BASIC for this computer. I'd never studied computer language writing, but figured out good approaches for that. Steve Jobs said that there was a lot of interest in having computers but not in wiring them up, so why don't we start a company to make PC boards for \$20 and sell them for \$40. We'd have our own company and all. Steve came up with the name Apple Computer. I sold my most valuable possession, my HP-65 calculator. We came up with a few hundred dollars and started this company.

After the PC board was done, Steve struck up a partnership with Paul Terrell at the Byte Shop, the only computer store in our region, to sell fully-built and assembled computer boards for \$500 each. I was into repeating digits so we priced it at \$666.66 retail.

We did not make or sell a lot of Apple I computers over the next year, but we had other jobs. We did get our name and computer characteristics in many articles over that year and it was easy to see that Apple Computer was getting very well-known in some circles.

I look back on my Apple I design and actually have trouble figuring out some of it. My designs back then were sometimes too clever to figure out. They were designed to save parts and cost.

The Apple II was really the computer designed from the ground up that would kick off personal computing on a large scale. But the Apple I took the biggest step of all. Some very simple concepts are very hard to do the first time. This computer told the world that small computers should never again come with geeky front panels, but rather with human keyboards, ready to type on. After the Apple I, Processor Technology introduced the SOL computer, and it also came with a keyboard and monitor and became the hottest selling Intel-based hobby computer, selling thousands a month. Contrast that with the Apple I, of which we sold maybe only 150. The Apple II was to be the third low-cost computer to come with a human keyboard.

—Steve Wozniak
January, 2005

Chapter 1

Apple I History

In This Chapter

- The Apple I
- Apple I Owners Club
- Pioneer Interviews

Introduction

Steve Wozniak's microcomputer, the Apple I, served to launch the Apple Computer Company and generated an outburst of innovation that revolutionized the way people use computers. This book won't tell you who provided the venture capital or what date Apple was incorporated. Instead, this book is about computers, how to build them, and how to program them. In this first chapter, we will examine the history of the Apple I computer itself, its peripherals, modifications, and the community of users who were the first in the world to own a *personal* computer.

The Apple I

Merely finding an Apple I to buy was a daunting task in 1976. Two hundred were made. The Byte Shop purchased 100 of these. Most of the remainder was sold at other stores and through an ad in *Interface Age*. The retail price of \$666.66 wasn't much, but all that bought you was the circuit board (Figure 1.1). It was up to the user to find a keyboard, power supply, and video monitor.

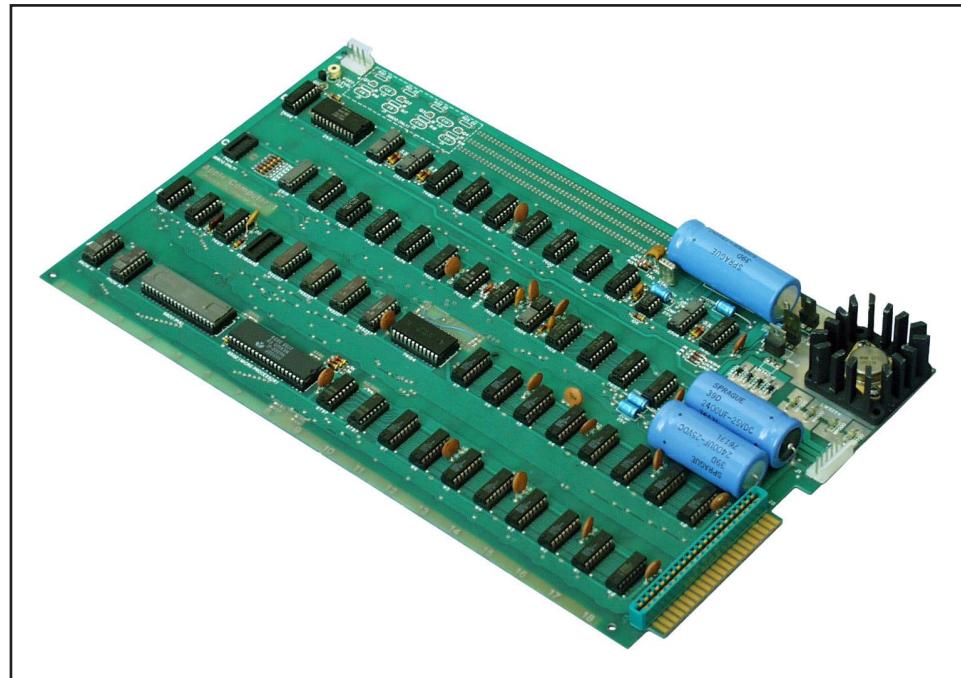


Figure 1.1 Apple I Circuit Board

With a user base of fewer than 200, nobody was making peripherals specifically for the Apple I. The power supply could be built with standard parts. For an ASCII keyboard, a Datanetics Corporation keyboard was the most common choice (Figure 1.2). The user had to make the cable on his own. A video monitor could be used for the display, but it was cheapest to buy an RF modulator and connect the computer to a television. The Apple I was the first computer of its kind to use a keyboard and monitor as standard input/output devices. Most competing systems, such as the Altair, used rows of lights and switches to communicate with the user or provided a serial port for use with an expensive Teletype unit.

Programs were loaded onto the Apple I either by typing the machine code in by hand or by transferring the program from an audio cassette. Apple BASIC was only available on cassette and most users were not keen on loading their programs by hand; therefore, the Apple I's cassette interface (Figure 1.3) was a ubiquitous option. Apple sold a handful of games on cassette, but nobody bought a microcomputer in 1976 to run commercial software.



Figure 1.2 Apple I with Datanetics Keyboard

The concept of commercial software barely existed in 1976. Companies like Apple made their money from hardware sales. Nine programs were available on cassette: BASIC, Mastermind, Lunar Lander, Blackjack, Hamurabi, Mini-Startrek, 16K-Startrek, Dis-Assembler, and Extended Monitor. Apple sold these at just \$5 each to encourage interest in the Apple I. Hobbyists wrote their own software and shared it freely with one another, eager for the opportunity to show off their innovations and share what they learned. Equal liberty was taken with manufacturers' software. Micro-Soft (as it was known before changing its name to Microsoft), which wrote BASIC for the Altair, was the first company to have a financial interest in software sales. "I would appreciate letters from anyone who wants to pay up," Bill Gates wrote in his poorly-received "Open Letter to Hobbyists."

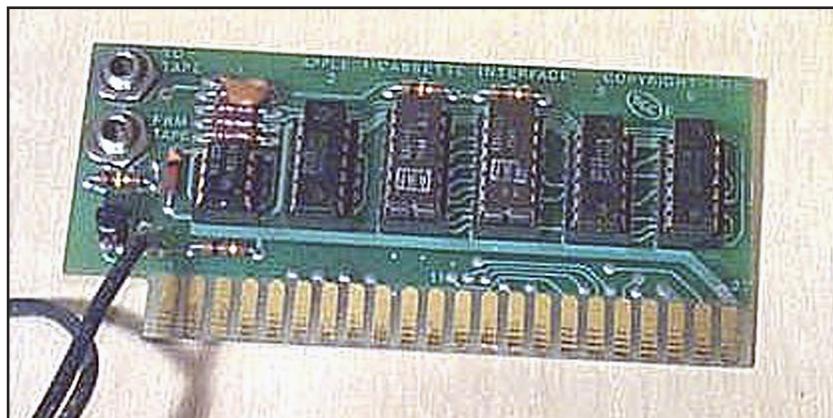


Figure 1.3 Apple I Cassette Interface

With the exception of those nine programs sold on cassette by Apple, any software the Apple I user wanted to buy or copy out of a book had to be laboriously translated by hand from its original format into Apple I BASIC or Assembly. A similar situation existed for the hardware. There were few microcomputer peripherals to which the competent engineer couldn't interface, but none of these was actually intended for the Apple I.

One popular device was the Southwest Technical Products Corporation (SWTPC) PR-40 alphanumeric printer (Figure 1.4). This printer cost \$250 and, like the Apple I, did not support lower-case characters. It used standard adding machine paper and could print 40 characters per line. The Apple I had no parallel port, but users could build a small circuit that would redirect video output to the printer.

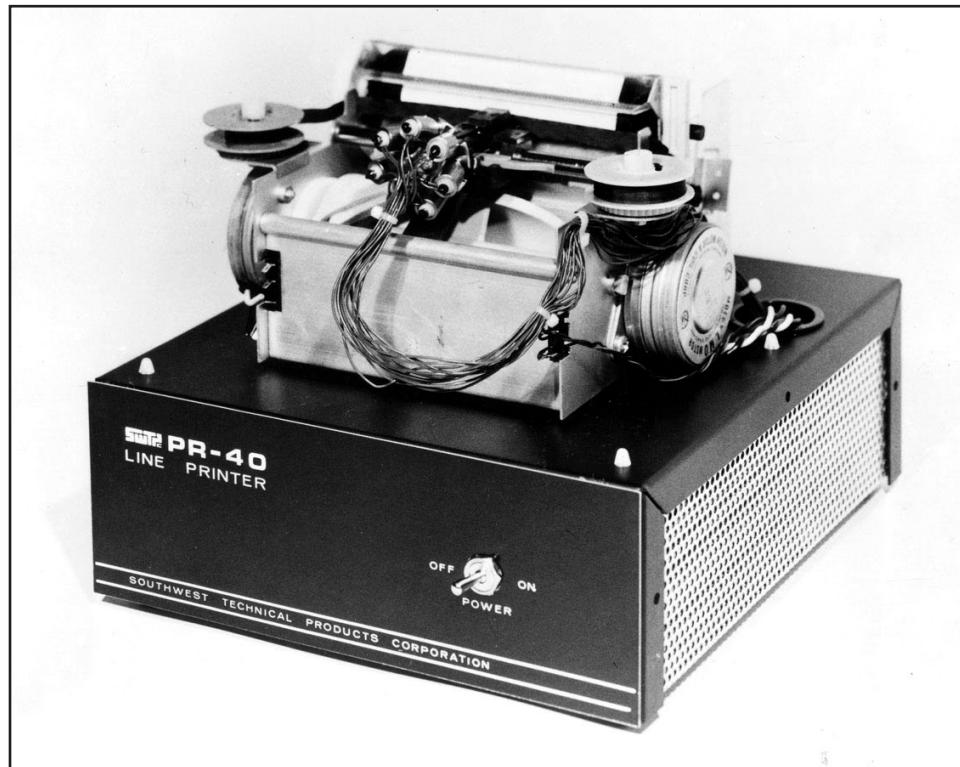
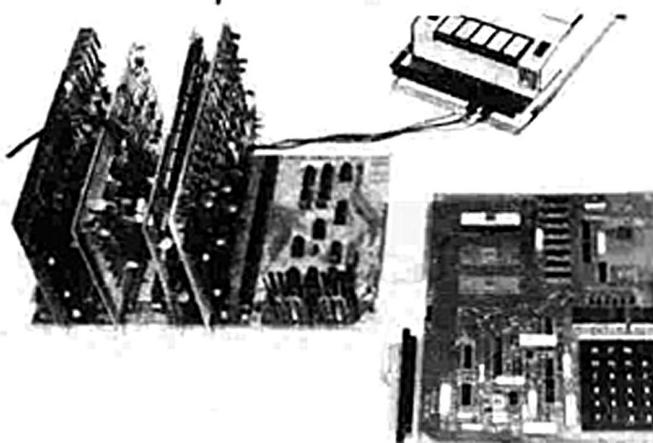


Figure 1.4 SWTPC PR-40 Printer

Another peripheral, called the KIMSI (Figure 1.5), was an interface that allowed popular S-100 peripherals to be used with the MOS KIM-1 computer. With only minor adaptations, the KIMSI could be attached to the Apple I's 44-pin expansion bus. Any S-100 card could be used with this interface, but writing programs that used the cards was up to the Apple I user.

KIMSI



The KIM to S-100 bus Interface/Motherboard

- Combines the power of the 6502 with the flexibility of the S-100 bus
- Attaches to any unmodified KIM
- Complete interface logic and fully buffered motherboard in one unit
- On-board regulation of power for KIM
- Eight slots of S-100 compatibility for additional RAM, Video and I/O boards, PROM Programmers, Speech processors . . .
- Includes all parts, sockets for ICs, one 100 pin connector, and full Assembly/Operating documentation
 - ♦ Kit \$125, Assembled \$165
 - ♦ All units shipped from stock

FORETHOUGHT PRODUCTS
P.O. Box 386-F
Coburg, OR 97401



Figure 1.5 KIMSI Interface

22

**GT-6144
TERMINAL
SYSTEM**

The GT-6144 is a Graphic Terminal System that displays graphic data on a slightly modified black & white television set or standard video monitor. The terminal has its own 6,144 bit static RAM memory which eliminates the need to be used with a specific computer system. The terminal will operate with any computer system whose parallel interface outputs an 8 bit data word and "data ready" strobe. This includes "8080 type" and SWTPC 6800 Computer Systems.

The display screen is divided into an array of cells 64 wide x 96 high. Each cell is individually addressable and may be selectively turned ON or OFF by programmed commands from the computer. With a little imaginative programming fixed or moving images may be displayed on the screen for added enhancement to game programs. The photograph shows Startrek's starship the "USS Enterprise" generated using the graphics terminal with our SWTPC 6800 Computer System. Memory cell data can be loaded in less than 2 microseconds; much faster than most micro-computers can generate the information. The system features a power-up screen blanking circuit which may be enabled or disabled at any time thru program commands from the computer system. In addition, a unique image reverse feature allows you to select between white on black or black on white images by a simple one word command generated by your computer's program. The system will operate on either 50 Hz or 60 Hz power lines with American standard 525 line or European standard 625 line television sets or video monitors.

The unit is supplied less the chassis and does not include the required video monitor or modified television set. Instructions for the addition of a video input jack to the television are included with the kit. You may use the same television set or video monitor used by your CT-1024 terminal system. In fact, control commands from your computer allow you to display graphic, CT-1024 alphanumeric, or even a combination of the two, all on the same display device. Power requirements for the terminal are 5.0 VDC @2A, -12 VDC @20 Ma and 6 VAC @20 Ma. These requirements are met by the optional CT-P power supply.

# GT-61 Graphic Terminal System Kit	\$98.50 PPd in US
# CT-P Power Supply Kit for GT-61	\$15.50 PPd in US

SWTPC Southwest Technical Products Corporation, 219 W. Rhapsody, San Antonio, Texas

Figure 1.6 GT-6144 Terminal System

The most ambitious upgrade to the Apple I was the addition of a SWTPC GT-6144 Graphics Board (Figure 1.6). The graphics board allowed the system to display a 64-by-96 array of rectangles (pixels) in black and white. It was available only as a kit and, like the PR-40 printer, it interfaced to the computer via a parallel port. After building a simple custom interface to attach the graphics board, the user could write programs that drew graphics to the screen or even performed simple animations.

Apple I Owners Club

In the earliest days of the Apple I, Steve Wozniak would visit the Homebrew Computer Club, sharing schematics and giving away code. Fellow hobbyists could even expect hands-on support from Woz as they experimented with their early Apple I computers. As Apple grew, the new Apple II quickly became the center of attention and interest in the Apple I fell to the wayside.

In 1977, Joe Torzewski decided to found the Apple I Owners Club to support users of the Apple I. Joe sent announcements to *Interface Age* and *Byte* magazines. Apple happily sent out letters to Apple I customers instructing them to direct further inquiries to Torzewski. The club was expected not just to provide peer support, but also to take over distribution of Apple I software.

Five people—Joe Torzewski, Larry Nelson, Richard Drennan, Fred Hatfield, and Dr. Arthur Schawlow of Stanford University—made up the core of the group, which had roughly 30 members altogether. Members exchanged programs they'd written and compiled a software library. They ported the Focal programming language to the Apple I and developed methods for memory expansion. As interesting new hardware and peripherals were released, Owners Club members shared their strategies and schematics for getting the equipment working with the Apple I.

The Apple I Owners Club continues today, with founder Torzewski, original member Larry Nelson, and many new members who have never used a real Apple I. With message boards, file libraries, and member web pages, the Owners Club continues to serve hobbyists and offer a community to those eager to learn about computers.

The Apple I Owners Club welcomes new members who are interested in building replicas of the Apple I as well as those simply interested in learning more about how computers work. With the help of new members, the Owners Club hopes to provide many fascinating new articles on hardware expansion, programming, and other projects. If interested, you can join them at www.applefritter.com/apple1.

Apple I Owners Club Premier Newsletter**APPLE 1 POWER**

Well, this is our start and I hope it's just one of many. So far, there are about 30 of us. First of all, I would like to put out a list of names of persons who want to get in touch with other Apple 1 owners and are working on projects, also a list of persons who would be willing to help others with the hardware or software projects.

We need hardware articles on anything that you have hooked up to your Apple. Memory expansion is high on the list.

Where is our Apple 1 hardware expert? I need someone who can devote his time on a project for us. Needs access to a slide projector or should have one. I have 35mm slides and some notes on how Apple put the 16K chips on the Apple 1 board. Someone has to figure out from this (mainly the slides) and try to write it up so we can pass it along to everyone. So, don't be bashful; we need you.

Would like to hear from you guys with the new monitor and what you have done with it and the commands and such. Trying to get all the details from everyone and trying to piece it together.

Software programs in basic and machine language are needed. Maybe, we should work on certain types of programs together as a group for a month or two and then go to another area. Take one type of program and try to modify it and see what we can come up with. Open for ideas on this, so input now. How about sending a list of your program so we can see what is out there?

Here is a basic (Apple-Basic) challenge from Larry Nelson. What is the best Apple-Basic solution to the problem with the following program?

75 DATA 13, 14, 15, 22, 23, 24, 29, 30, 31

77 For W-1 to 9

79 HEAD K

81 B(N)--7

32 NEXT W

There is a program, 4 pages, a primitive, line-oriented text editor that you can use to draw pictures or text and store them, from Professor Schawlow. Well that's about it for now, remember to send in your inputs and program and a stamp or so if you can. Till next time...

Pioneer Interviews

The most fascinating stories of the Apple I come from those who were there at the beginning. The following interviews with original Apple I owners, conducted in 2005, attempt to capture the experience of using an Apple I when it was new in 1976. Joe Torzewski, Larry Nelson, Ray Borrill, Liza Loop, Steve Fish, and Allen Baum discuss their first experiences with the Apple I, their programs, and their hardware projects.

Joe Torzewski

Joe Torzewski is the founder of the Apple I Owners Club. Now retired from industry, he remains an active hobbyist with the Apple I. Pictures of Torzewski's Apple I are in Figures 1.7, 1.8, and 1.9.



Figure 1.7 Putting the System Together

TO: When did you first learn about the Apple I?

JT: I believe I first learned of the Apple I through an ad in one of the older computer magazines, *Interface Age*. When I saw their ad, I thought having video, memory, and processor on the same board was a great idea. The S-100 bus depended on a separate card for everything.

TO: How did your background as an engineer affect your interest in the Apple I?

JT: I liked the idea that everything was there on the board. At that time, they were having problems with the S-100 bus and the Apple I's 44-pin bus looked good for expansion. The 6502 CPU was also easier to program.



Figure 1.8 Torzewski's Apple I

TO: Why did you start the Apple I Owners Club?

JT: I was having problems with Apple Company over support. The Apple II was all that interested them and the Apple I was something they did not want to support anymore. So, Apple started giving my name and address to any Apple I computer owners who contacted them for help. One of the first persons that I met was Larry Nelson, who was an outstanding programmer for the 6502 chip and the Apple I.

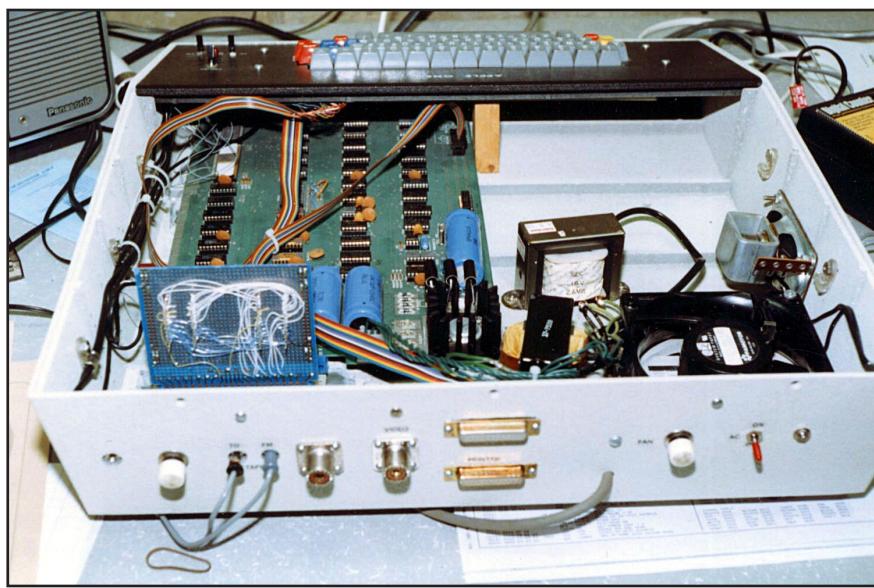


Figure 1.9 Inside Torzewski's Apple I

TO: How did you learn to program for the Apple I?

JT: Everything was self-taught; that's how it was back then. One of the main references to use for machine language programming at the time was from MOS Technology, the maker of the 6502 chip. It explained how to use each command. Larry Nelson and Richard Drennan did a lot of programming and hardware mods.

TO: What programs did you write?

JT: I did write a few games in BASIC and modified a few for the Apple I. I also did some machine language mods to add more commands to programs.

I adapted the game Gomoko in BASIC to run on the Apple I. When you played the game, you had to enter the row and column that you wanted and then the computer would draw the boxes, using dashes going across and exclamation points going down. Then it put your X or O into the box, depending if you moved first or second. You had 100 squares and had to get five in a row, diagonal, vertical or horizontal, in order to win.

TO: What hardware modifications did you perform?

JT: I added an extra 16 KB of memory to the board. This involved adding two more chips in the work area on the board and wiring and cutting tracks on the motherboard to make it work. I got that info from Professor Arthur L. Schawlow. I put in the expansion board with three extra slots and then built an EPROM board (Figure 1.10) and burned in a better Monitor ROM for the Apple I.

Others in the club added the GT-6144 graphics card and even the KIM-1 memory board.

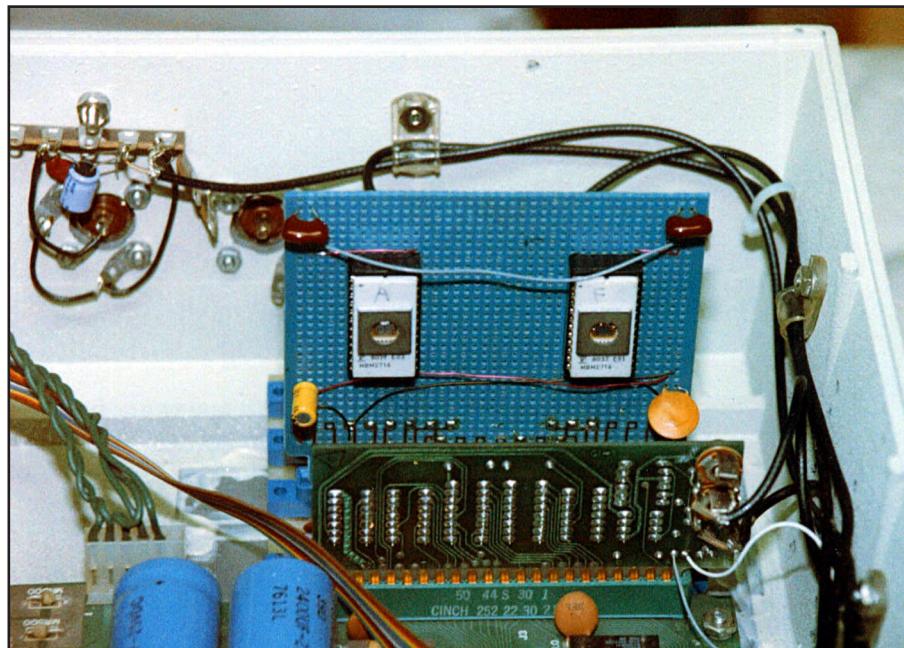


Figure 1.10 Homebrew EPROM Card

Larry Nelson

Larry Nelson has been an active participant in the Apple I Owners Club since 1977. The owner of an Apple I for many years, he has been an enthusiastic user, programmer, and hardware hacker. He is currently retired in southern Indiana and continues to use a replica of the Apple I computer for hobby purposes.

TO: When did you first learn about the Apple I? Why did you want a computer and why the Apple I?

LN: I took courses in electronics in the early Sixties and, although I didn't go to work in the industry, I did have an interest in the field. In 1977 I began to see articles in electronic magazines about the new computers for individual use. When I discovered that the Byte Shop had a store there, I drove 60 miles to Fort Wayne, Indiana. They had displays of several personal computers, and I looked at several. The Apple computer was the one that interested me the most, as it came with a great cassette interface, BASIC, and lots of memory. Soon after I discovered it, I returned to buy one for myself. This would have been in the summer of 1977. The purchase included the computer circuit board with 8 kilobytes of dynamic RAM, a cassette interface, an ASCII keyboard, an RF modulator to adapt the video output to the input of a television, and the book *101 BASIC Computer Games* by David Ahl. Back home, I still had to acquire two power supply transformers, build the RF converter (it was in kit form, Figure 1.11), and wire switches and power cords to the unit. I built a case for the Apple from sheet metal, since I was operating an HVAC business at the time.

TO: Could you describe the process of writing a program for the Apple I?

LN: I had no programs (other than a tape of BASIC), so the book by David Ahl was my only source of input material. My son, then a junior in high school, was interested in computer programming and went to a computer camp at nearby Anderson College for a couple of weeks while I assembled the Apple. The emphasis at the camp was on BASIC programming. The Apple was completed when he came home and he helped me learn to program in BASIC. Most of the listings I had were for "standard" BASIC, which, of course, Apple BASIC was not. The limitations of Apple BASIC included integer-only math, single-dimension arrays, and a non-standard method of string manipulation.

But with careful editing, many corrections of "syntax error" messages, and a lot of guesswork, I got a couple of games up and running, the first being "HELLO." Most games in the David Ahl book were too large for my Apple's memory.

I subscribed to *Byte* and *Interface Age* to gain more knowledge. I finally tried a little programming in Assembly language, although it required assembling a program manually, converting the listing to machine code, then typing the resulting hexadecimal into the Apple. Many hours were spent hunched over the keyboard just to see the results of a simple game. Several times, I copied hexadecimal listings from magazines into the computer, one byte at a time.

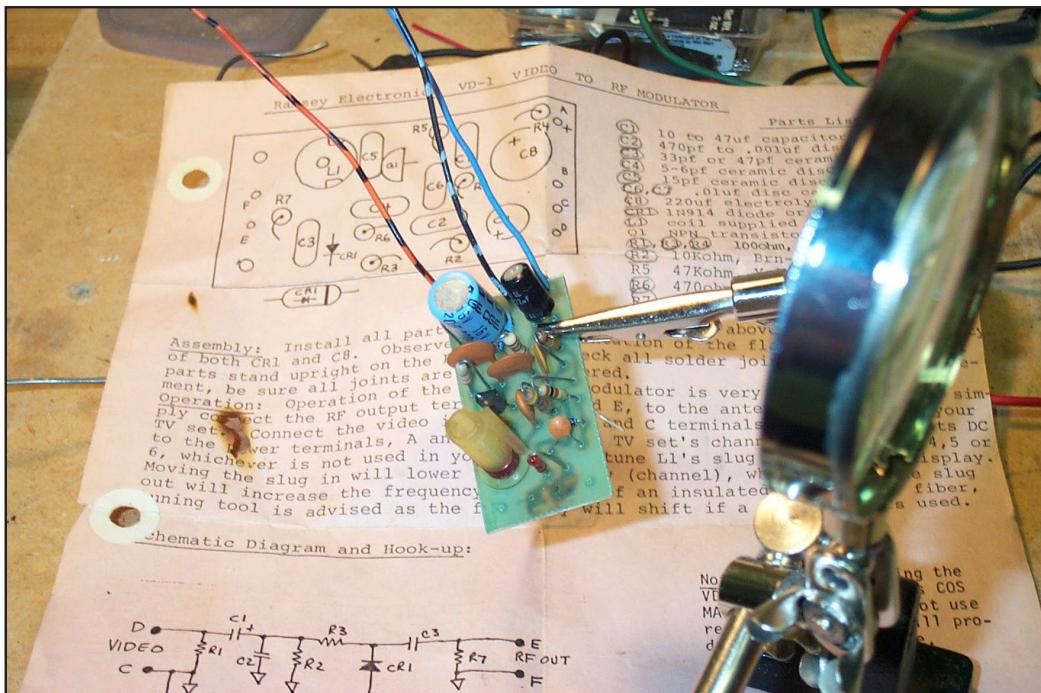


Figure 1.11 Building a VD-1 RF Modulator

A fellow named Peter Jennings wrote a chess-playing program for the KIM-1 computer, another 6502-based microcomputer. I sent off for a listing of his program and successfully adapted it to the Apple I.

TO: What hardware modifications did you perform?

LN: I added a printer (the PR-40) that had 40 columns of width on a 4-inch paper roll. Then I replaced 4 KB of RAM with a 16 KB set of chips. Now I had an amazing 20 KB of memory, more than I would ever need! However, at a swap meet in Fort Wayne, I got another 4 KB of static RAM on a separate board, which I wired to a connector to plug into the Apple I's expansion connector. I don't think I ever had another "MEM FULL ER-ROR***" message after that.

TO: Tell us about the Apple I community.

LN: In response to an inquiry to the Apple Computer Company in late 1977, I found Joe Torzewski. Joe at that time had the Apple I library (about a half-dozen games, routines, and programs), and he began sending me lots of material from other Apple I owners.

Other than Joe Torzewski, I have never met another Apple I owner in person. I did happen upon a local fellow with a KIM-1 and we spent some time discussing the various things we could do with our micros. I was pleased to learn that the Apple I was far more flexible in its abilities to be programmed and the monitor output was superior to the LED display of the KIM-1.

Ray Borrill

Ray Borrill began his career in electronics after leaving the U.S. Army in 1956. Employed as a technician at Brookhaven National Labs, he constructed digital systems for nuclear research. Borrill enlisted in the U.S. Air Force in 1958 and attended courses on the computers used in the SAGE system. Honorable discharged from the Air Force in July 1959, he soon founded Applied Digital Data Systems (ADDS), which became one of the leading suppliers of IBM- and Teletype-compatible CRT terminals. He moved to Indiana in 1973 to work as a systems engineer, designing and developing computer-based systems for psychological research. In February 1976 Borrill opened The Data Domain, one of the pioneering retail computer stores. Borrill died in September 2006 at age 75.

TO: What is your background in computers?

RB: If I have any talent at all, it is the task of telling or writing “war stories” of the computer industry, which I have been directly involved with since the fall of 1958. I got into the personal computer business sort of passively in 1975, when in November I attended the famous Kansas City meeting sponsored by *Byte* magazine. The purpose was to develop a specification describing the operating parameters of an interface between a serial data port on a personal computer and an audio cassette player so that data could be compatible between systems. It was a lofty but naive objective because virtually every manufacturer in the industry already had a product on the market or at least on the drawing boards.

There were more than 25 people there, but I was the only one who did not represent anyone but himself. I met and became somewhat acquainted with Don Tarbell, Don Lancaster, Harry Garland of Cromemco, Hal Chamberlain of *The Computer Hobbyist*, Lee Felsenstein, the people from Processor Technology and IMSAI, and more. I decided there and then to open a computer store as soon as possible.

It took me almost three months, but The Data Domain started in about 750 square feet just off Court House Square, in Bloomington, Indiana, on February 12, 1976. At that time we were authorized dealers for IMSAI, Processor Technology, Cromemco, and several makers of aftermarket add-ons, as well as TV monitors, keyboards, every computer book we could find, every computer magazine on the market, and even computer-generated works of art (Figure 1.12)!

At the World Altair Convention in March, I met and became friends with Ted Nelson, author of *Compute Lib/Dream Machine*. Ted was the keynote speaker and kept the large audience in hysterics for an hour giving his somewhat risqué predictions of the future digital world. Ted was there with his friend Jim Banish, and they told me that they were opening a store in Evanston, Illinois, and thought we should establish some sort of relationship. They were interested in my experience and talent for selling computers and I could take advantage of their great financial management group. The result was that I became the vice president of the itty bitty machine company as well as the sole proprietor of The Data Domain.



Figure 1.12 The Data Domain (Second Store Location)

I have a picture (Figure 1.13) of The Data Domain that I took in April or May of 1976. It is my firm belief that we were the first to use the term “personal computer” commercially. DEC used it internally in 1972 or ‘73. Apple used it in a Wall Street Journal ad in 1978 and got the credit for making it catch on, but I used it first!

TO: How did you become involved with the Apple I?

RB: By June the store was going great guns and I was always on the lookout for new products to sell. One day, I got a call from a young man named Steve Jobs. He had just spoken with Jim Banish of the itty bitty machine company, who told him that I was the guy he would have to convince since I made all the purchasing decisions. He went into his spiel about what a great computer he had since there was no assembly required (a slight exaggeration since one had to wire a power supply, keyboard cable, display monitor and some other ancillary stuff, then find a way to package it all up nicely). But Jobs was a good talker and we needed some more products to sell. So, as was routine in those early days, I ordered 15 Apple I computers with the optional cassette interface card, sight unseen, on the word of a guy I had never met or heard of, and which would be delivered C.O.D. “soon.”

And thus, The Data Domain the itty bitty machine company became two of the first four dealers for Apple Computer. The first dealer was The Byte Shop, and the second was Stan Viet’s store in New York. The Apple I was hard to sell because of the packaging problem, and for some reason we were never supplied with the cassette version of Apple BASIC, which made some buyers very unhappy. But, eventually, all 15 were sold, except for two. One of these was a machine we gave away to the U.S. Olympic tennis team. Only a few weeks later, it was destroyed in a plane crash that killed several members of the team. The other Apple I stayed in my display case for a couple of years. When it

began to gain fame for its design, I took it home and kept it. There it stayed for 25 years until I decided to auction it off in 2001.

TO: What was your impression of the Apple I? Did you do any programming for it?

RB: Of course, in my case, there was no real personal decision in choosing and purchasing the Apple I over some competitor. I listened to Jobs' sales pitch and it seemed like a good idea to be able to offer a computer that did not require any soldering skill. It was inexpensive enough to sell, and we had the opportunity to increase the value of the total sale (and thus the profit margins) by selling the things needed to make a complete system, such as a cassette recorder, power supply components, keyboard, monitor, and a case; and as a further option, we offered to assemble the whole thing. While my techs wrote some programs for the Apple I, I did not and only learned to run some demos that we wrote, and things like that. I spent more of my time teaching people how to make interfaces, etc. for all of our computers. I didn't get as closely involved with the Apple I as I did with the Apple II and S-100 bus computers.

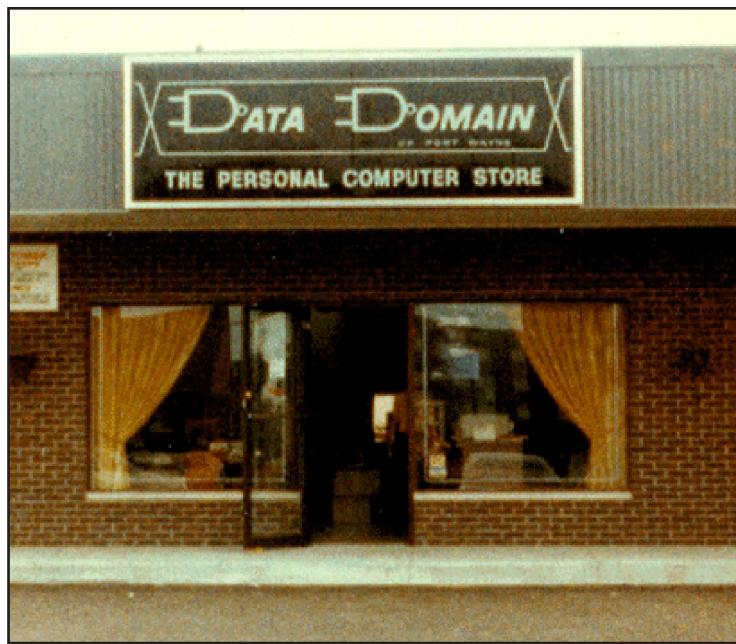


Figure 1.13 The Data Domain, Circa 1976

TO: What were some common peripherals and modifications for the Apple I?

RB: Printers were popular, but in those days about the only printers available to the general public were Teletypewriters and lots of surplus units, both serial and parallel. You wouldn't believe the confusion and frustration of hooking up a simple Model 33 ASR Teletype to a serial interface when you had no idea how a Teletype works and never heard of RS-232C! One of the nice things about the Apple I was that it required no modifications to make it run and was a complete, working system once the ancillary stuff was assembled. Occasionally, someone would wish to change from the 6502 proces-

sor to the Motorola 6800, which the motherboard was designed to allow, or add a serial RS-232C interface in the “kludge” area, but these were not ordinary modifications and rarely done.

The real fact is that the Apple I’s potential was never exploited by Apple. At the Atlantic City convention in August 1976, all attending dealers were shown a demonstration of the color graphics of the upcoming Apple II and that was the end of orders for the Apple I.

Liza Loop

Liza Loop is a social scientist with a focus on how individuals adapt to changing social and physical technologies. She draws her data from the everyday life that surrounds her and often sets up venues, such as the LO*OP public access computer center, so that she can be an up-close participant observer. “Every phenomenon can be interpreted as a complex system,” she said. “By paying attention to the characteristics of the elements inside the system, the relationships, both real and possible, among those elements and the emergent properties of the system as a whole, we can learn how to shape our world in positive ways.”

TO: When did you first learn about the Apple I?

LL: I went to a Homebrew Computer Club meeting in early 1976 and announced that I had a storefront computer center and was planning to take my machines into elementary and secondary schools. I was looking for equipment to use to demystify computing to children and the general public. Woz heard my comments and brought the Apple I to a Sonoma County Computer Club meeting for a demonstration. At that time, he gave us the computer. Before then, I knew nothing about the Apple I.

TO: Why did you choose the Apple I over other computers?

LL: Because Woz gave it to us. Actually, I found it very difficult to use. Our Apple I wouldn’t stay running for a 45-minute class period and caused no end of frustration. I took it back to Woz and asked him to fix it. He made it a little better but still not very serviceable. Eventually, he provided us with an Apple II to replace it with.

TO: Did you do much programming in the classroom?

LL: I only used the machine to demonstrate simple BASIC routines and to play a few games. It was easy to take around to different schools because it was small and light. Probably 50 to 100 students ran their first 5-line BASIC programs on our Apple I. For many of them, the Apple I was the first computer they had ever seen. The workhorse of our classes, however, was a PDP-8e.

TO: *What hardware modifications did you perform?*

LL: Apple I number one came to LO*OP Center as a naked motherboard (Figure 1.14). One of our computer club members built it a box and a power supply. I drove two hours from our storefront in Cotati, California, to Palo Alto to buy a Cherry Pro keyboard. We used an audio tape recorder for “mass” storage.

I believe Woz added some jumper wires when he had the machine back for repair. Today we would probably call that an upgrade! Other than that it is still in original condition.



Figure 1.14 The LO*OP Center's Apple I

TO: *Did you know any other Apple I owners or microcomputer hobbyists? Could you tell us about the community?*

LL: In the fall of 1975 when I opened LO*OP Center (LO*OP stands for Learning Options * Open Portal), I knew next to nothing about computers. I began the Sonoma County Computer Club simultaneously so that I could create a nearby hobbyist community. Out of the Sonoma County woodwork came about 30 folks, mostly men; however, I was not the only woman who knew a lot about programming and electronics. Everyone

was very excited when the MITS Altair arrived and the participants taught each other about the machines they knew. Those club members fanned out across the microcomputer industry as it grew. One worked on the original Wordstar. Another developed the first TRS-80 disk drive. Someone else wrote a stock market analysis program and retired on the bundle he made. The guy who built the Altair is still crafting acoustic guitars in the Sonoma County hills. I see some of them occasionally when I go to the Vintage Computer Festival. We're greyer, but there's a glint in our eyes that says we were involved in something really revolutionary.

For me, computers were a means to an end and not an independent interest. I needed the hobbyists to teach me and to provide hardware support. They did it happily and were very tolerant when I tuned out on the endless discussions of which capacitor applied where would increase throughput how much.

For most of 1976, LO*OP's Apple I was the only Apple in the North Bay, although there were several in Berkeley and on the Peninsula. By the time the Apple II came out, there were several single-board computers on the market that kept the hardware enthusiasts engaged. The Apple II was not a kit, so it didn't appeal as much to the Club's group of tinkerers. There was also a growing demand for a plug-and-play computer amongst science-minded highschoolers, a few teachers, and many forward-looking business types. The Apple II filled that niche in a way the Apple I never could have.

TO: What motivated you to become so involved with computers?

LL: Because I grew up among linguists and electronics pioneers, I always believed that computers would become pervasive within our society. My concern was that we would follow a path leading to all the machine-control and information-handling power of computers residing in the hands of a tiny scientific elite. When, in 1972, I discovered a Montessori primary school that was using a terminal attached to Lawrence Hall of Science as an educational "toy," I was delighted. Making sure that youngsters knew exactly how programs got into computers was the best possible vaccination against public intimidation by that elite (Figure 1.15). From then on, I put all of my available energy into finding ways to get ordinary people to take control of those mysterious machines—computers.

Steve Wozniak was on a similar track. He also wanted to provide broad public access to and understanding of computers. Luckily our paths crossed at Homebrew. I'd like to think that by keeping in touch with him and sharing my woes trying to use the Apple I in educational settings, I contributed, in a small way, to the birth of the Apple II and its success in making the computer an everyday tool.

The task of harnessing the power of computing to the needs of sustainable human society is not complete. Humanity is still in danger of computing its way to extinction. Perhaps, by following the life history of those 200 Apple I's that existed for a short time, we can learn a lesson in the appropriate use of technology.



Figure 1.15 Learning at the LO*OP Center

Steve Fish

Steve Fish was introduced to computers in the U.S. Navy in 1972. He worked for Burroughs Corporation as a computer systems technician in the mid Seventies in its Santa Barbara, California, minicomputer assembly plant. He attended Brooks Institute in Santa Barbara and graduated with a degree in photography. He became immediately involved in the microcomputer revolution and founded Peripheral Visions, which specialized in photographic and audiovisual software and hardware until the early 1990's. Fish now works as a freelance underwater photographer, filmmaker, and Web designer.

TO: Did you have any background in computers previous to the Apple I?

SF: In the mid-1970's, I spent a lot of time prowling the few computer stores that existed in those days, looking for a way to get back into computers. In the Navy, I had worked a little bit with Super Nova Micro computers around 1972-73. After the Navy, I used that experience and worked for a few months as a computer technician for Burroughs before returning to college. Though I had repaired quite a lot of computer processors and peripherals, I hadn't ever really done any programming. Running maintenance routines wasn't all that much fun. I took a "programming" class, but it consisted mostly of punching cards and sorting them into stacks. Not very satisfying.

TO: When did you first learn of the Apple I?

SF: In 1976, when the Apple I was first introduced, a small computer store called Computer Playground had just opened in Orange County, California, near where I was working. In addition to selling the Apple I and the limited other hardware items that

were available at that time, they had a number of Apple I systems installed as “workstations” that you could rent by the hour to play games or write your own programs. I started frequenting the establishment to play games (which was much more fun than sorting a stack of punch cards) and took a BASIC programming course taught by Dr. Will Otaguro, one of the owners. While I don’t remember the first program I wrote, I definitely remember the feeling of instant gratification when I ran it without having to run a stack of cards through a Teletype and wait for a line printer to churn out a ream of paper. These computers were cool! They were self-contained, and when you hit the Return key, execution was immediate.

The Apple I workstations were built into a false wall with a keyboard shelf and a monitor behind a window. Everything else was in the back; you never actually saw the hardware. You could play Startrek, Life, Lunar Lander, or several other games. When you wanted to play a different game, or tinker in BASIC, a staff member (usually Will) would go into the back room where the motherboards and monitors were and load the program for you. The Apple I systems that they had for sale were a bit fancier. They had custom-made walnut cases with built-in keyboards. They looked a little like wooden versions of the (future) Apple II.

The store gained quite a following. Any evening you would find several users sitting at their keyboards, either banging away at a program or playing one of the games. Not many were programming anything in machine language at that point; it was pretty much all in BASIC because there wasn’t an assembler for the Apple I in the early days. Heck, even Woz was hand-coding his machine language at that point; BASIC was written on paper!



Figure 1.16 Fish's Apple I

TO: When did you decide to buy an Apple I?

SF: I didn't buy an Apple I immediately. I rented a lot of time on them, though. When the Apple II came out, I decided to take the plunge and bought a 4 KB Apple II motherboard. Yes, initially, you could buy just an Apple II board and add your own keyboard and power supply just like the Apple I. That stopped after the Apple II was on the market a few months, but I got one out of the first production run (serial number 468) and still have it. When the Apple II was introduced, the Apple I market dried up pretty fast. Everyone (myself included) was hot for the II. Computer Playground merged with another Los Angeles-area computer store called Computer Components and moved into much larger quarters. In the move, the idea of workstations that you could rent time on sort of died off (until the revival of the cybercafé 20 years later). All of the workstation Apple I's and a couple of the nice walnut-cased units were sold off at clearance prices. I bought one of the nice Apple I's in the walnut case in late 1977, mostly out of nostalgia (Figures 1.16 and 1.17). It was the machine that I learned to program on, and they were dirt cheap by that time. I believe that the bare-board systems were donated to an Explorer Post in the area for Geek Scouts to play with.

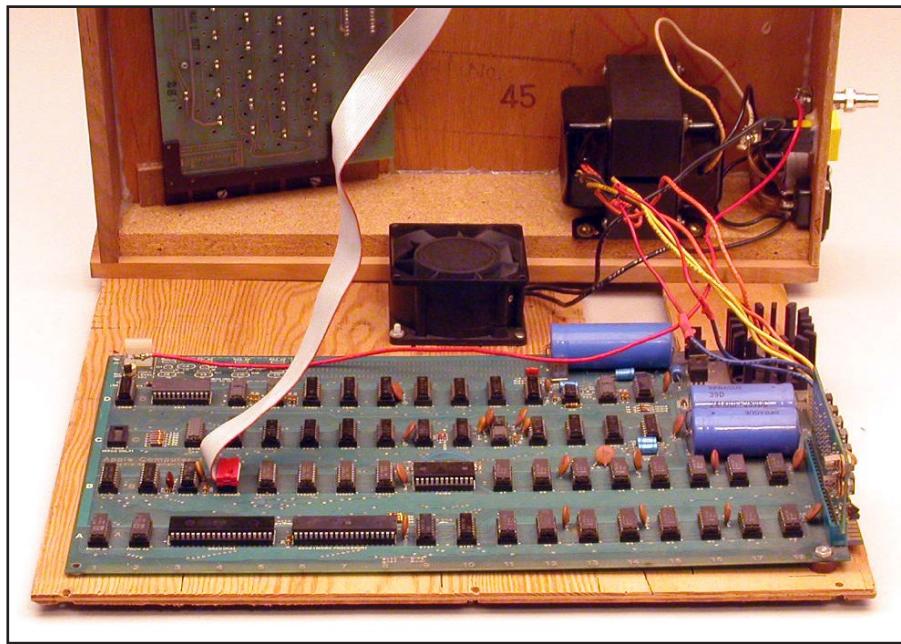


Figure 1.17 Inside Fish's Apple I

TO: How did you use your Apple I?

SF: I'm not sure if I ever even powered up my Apple I after I bought it. I think I probably just put it in the closet, figuring I'd get it out and play with it someday. That day didn't happen until about 28 years later when I found the Apple I Owners Club on Applefritter by chance, and was inspired to get it out and dust it off. I had a few minor problems getting it running, but mostly it was just a couple wires that had broken from being moved around and a keyboard problem. At any rate, it's now alive and well, and

in pretty much the same condition it was when it was brand new. No hacks or mods on the motherboard. It's as close to being mint as a 28-year-old computer can be. It was never owned by anyone else, or even used until 2005. I have the Sanyo nine-inch black-and-white monitor that was originally used on my Apple II, but it is of the same vintage as the Apple I. Also, a Panasonic cassette recorder was originally used with my Apple II, but is the same model that was being used with the Apple I at that time for loading programs. The cassette interface on the Apple I was always a little bit on the fussy side, but once you got the input and output levels set correctly, it hummed along quite nicely. It's only about 1500 baud, but in those days, that was quite fast. Mine still has the original 8 KB RAM configuration. In 1977, 16 KB RAM chips cost about \$500 for a set of eight. Though I have a box full of 16 KB chips now, I'm choosing to leave the Apple I in its original configuration.

I've been an Apple user continuously since 1976. I suspect that there aren't a lot of us left who have never, ever gone over to "the Dark Side." Heck, even Steve Jobs was a NeXT user for a few years. When I compare the capabilities of my Apple I of 1977 with my current Mac PowerBook, it's almost like a science fiction story come true. I feel very privileged to have been involved with personal computing literally from the very start. I wonder what the next 30 years will bring; I bet it will be wickedly cool.

Allen Baum

Allen Baum is a principal engineer at Intel working on enterprise processor architectures. Previously, he worked at Digital Equipment Corporation and Compaq Computer on the StrongARM (SA1500) and Alpha EV7 and EV8 processor designs. Prior to DEC he worked on the Newton, Apple II, and proprietary processors at Apple, and was a designer of the PA-RISC architecture and HP 45 calculator at Hewlett-Packard. He received his BSEE and MSEE from Massachusetts Institute of Technology in 1974. He is named in over 17 patents in the area of processor architectures and is chair of the IEEE Technical Committee on Microprocessors and Microcomputers, which oversees the Hot Chips conference.

TO: Could you tell us about the Homebrew Computer Club and your involvement with it?

AB: My involvement was as a "charter" (I guess) member. I saw a flyer for the first meeting (probably at the People's Computer Center) and thought that Steve [Wozniak] would be interested and dragged him along.

The Homebrew Computer Club was arguably the earliest meeting place for hobbyists who had, or dreamed of having, their own personal computer. It started in Gordon French's garage, where he demonstrated his Altair to us. Most of the early personal computer companies started there.

The club was later "conducted" by ringmaster Lee Felsenstein (himself greatly responsible for many early personal computer successes) and grew to fill the auditorium

of Stanford's Linear Accelerator Center. It served as a forum for product (and project) introductions and demos, software and information exchanges, asking and getting advice on problems (both in design and use of products), and just being able to mingle with other like-minded people.

TO: What did you use your Apple I for? What programs did you write?

AB: I was mostly writing development utilities for it. Steve was hand-assembling his code and typing it in hex. He could do that in his head, and he was an extremely fast and accurate typist. I wasn't, so even after hand-assembling, trying to figure out if what I typed in was what I had intended was tough. So, the first thing I wrote was the disassembler, which would allow me to see whether I typed the right thing in.

The next thing was a mini-assembler, which used the disassembler as a subroutine (basically, the assembler guessed all possible values, ran it through the disassembler, and stopped when it matched). Steve and I worked over those and got the sizes down to under 256 bytes each—I credit him for saving most of the bytes.

After that, I started working on a debugger and a Fortran compiler, but got bogged down with engineers' disease (I kept designing it trying to get it perfect, but never actually finished).

TO: Any stories to share?

AB: I remember Steve sitting in the lobby of Homebrew with his hand-wired Apple I (in addition to being an incredibly fast, accurate typist, he's the best technician I know—his wiring and soldering were unbelievably neat and careful) typing in his BASIC interpreter in hex—all 4 KB, trying to get it done before the break. He'd type a bunch and then would have to check it all, character by character, to make sure he hadn't blown it (he didn't, very often).

I think that's when I decided to write the disassembler and when he decided he'd better get a cassette interface working.

Apple contracted with one of Woz's fellow HP employees to design the cassette interface. It was a board full of opamps and analog parts. I think Woz was pretty disgusted—it was way overkill, so he chucked it and redesigned it to work with a few really simple parts, cutting down to a tenth its original size and cost. Steve was always really focused on getting the absolute simplest, lowest-cost design.

Summary

The Apple I is so primitive compared to computers today that it's difficult to imagine a time when users entered programs in machine code and connecting a printer required use of a soldering iron. Nothing about the Apple I was a mystery to its users. They knew how determine the function of every wire on the motherboard and every byte of code in their programs. As you learn to program and build your replica of the Apple I, you'll have the opportunity to learn all the secrets of its inner workings.

Chapter 2

Tools and Materials

In this Chapter

- Essential Tools and Supplies
- Soldering Iron and Materials
- Power Supply
- TTL Chips
- Circuit Boards and Software Tools
- Chip Pullers and Straighteners
- Keyboard and Monitor
- Ambiance

Introduction

Before we get started building an Apple I replica, let's take a look at a few of the tools and materials we'll need to complete the project. Some are more necessary than others and several won't be needed until later in the book. If you don't have the parts you need to build an Apple I readily available or have trouble obtaining them, you might consider skipping ahead to the programming section for the time being and simply using an emulator instead.

Essential Tools and Supplies

Multimeter

A digital multimeter is the first item on our list and a tool everyone should own. We're about to spend a few hundred pages talking about electricity, so it's important to be familiar with the tool we use to measure it. We'll be using it primarily to measure DC voltage and occasionally resistance. If you don't have a multimeter, consider buying one at once. It doesn't need to be anything fancy to be functional for use in these projects. An inexpensive meter from Harbor Freight or eBay can be had for under \$10 and will work just fine.

Figure 2.1 shows the meter I normally use. I almost ordered it from Jameco, but I saw Harbor Freight had the same unit on sale at half the price. Be sure to shop around to find the best deal. This meter is made by Cen-Tech, which seems to dominate the cheap multimeter market.

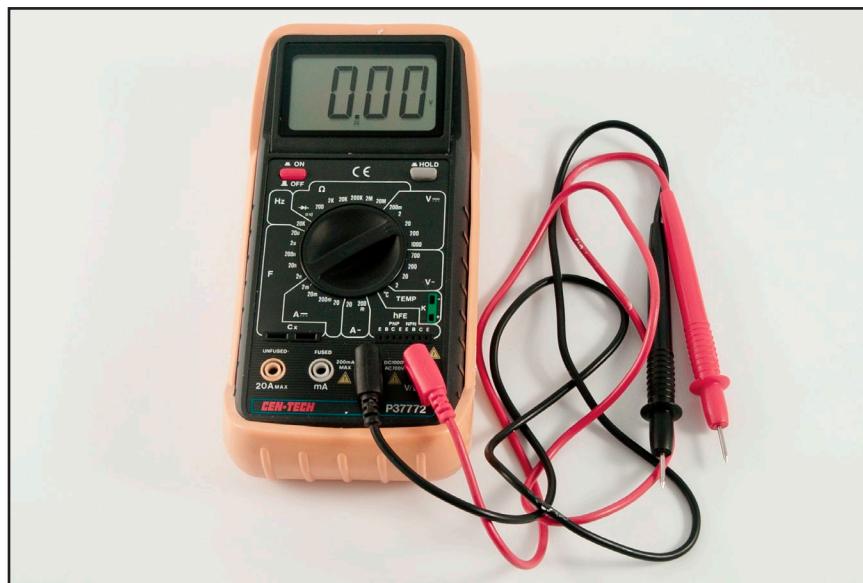


Figure 2.1 Cen-Tech 37772 Multimeter

Shown in Figure 2.2 is one of the less-expensive meters you'll find on eBay or at an import store. It's also made by Cen-Tech, and works moderately well, but is of a significantly lesser quality than its pricier counterparts. I've already broken one of the cables, and when the batteries died in this unit, I lost patience and wired it up to AC using a power adapter from an old modem. A less expensive model can be good to have around as a practical backup unit.

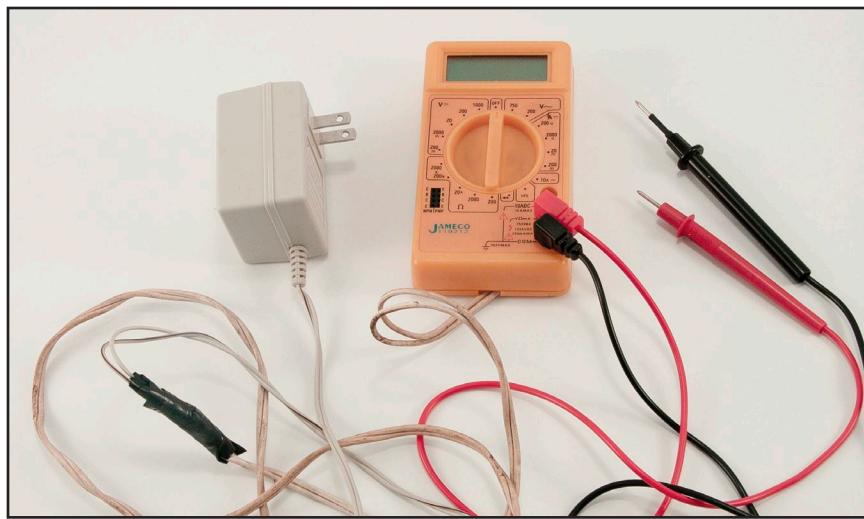


Figure 2.2 Cen-Tech 92020 Seven-Function Multimeter

Finally, if you're feeling especially geeky, large, older meters like a Racal-Dana can be fun (Figure 2.3). They're more precise and more reliable than their cheap, modern counterparts, and you can pick one up for a song from surplus sellers. Of course, they are kind of big... (This particular unit was Navy surplus and came out of a system for testing aircraft circuitry. It's about 20 years old.)



Figure 2.3 Racal-Dana Multimeter

Product Type	Retailer	Item No.	Price
Cen-Tech Multimeter	Harbor Freight	37772-3VGA	\$19.99
Cen-Tech 7-Func Multimeter	Harbor Frieght	92020-0VGA	\$9.99
EZ Digital Benchtop Multimeter	Jameco	132564	\$219.95

Table 2.1 Multimeters

Logic Probe

Digital signals are either off or on, low or high, zero or one. Touch the point of a logic probe to a wire and it will respond with either a “high” or a “low.” This is an invaluable tool for building circuits and one you’ll use frequently. Most logic probes have two light-emitting diodes (LEDs), one of which lights up when the line is high, and the other when it’s low. Since they’re shaped and held like pens, depending on how a logic probe is designed, the LEDs can be hard to see while you’re working. Look for a probe that has its LEDs close to the tip, or try to find one that also uses audio tones as an indicator. In addition, try to find one that looks like it could double as a murder weapon because if the point isn’t sharp (thin) enough, you could find it difficult to get into breadboard holes (discussed later in this chapter).

In Figure 2.4 you can see my logic probe, which is a bit awkward to work with because of the positioning of the LEDs. Figure 2.5 shows a classic Hewlett-Packard logic probe. These are no longer available new, but can still be found on eBay, and are often a better value than the majority of the newer probes.

A logic probe can be purchased for under \$20. If you’d rather not spend the money, it’s possible to get by with just a multimeter. In this case, 2.4 to 5 volts is a digital high, and 0 to 0.4 volts, a digital low.

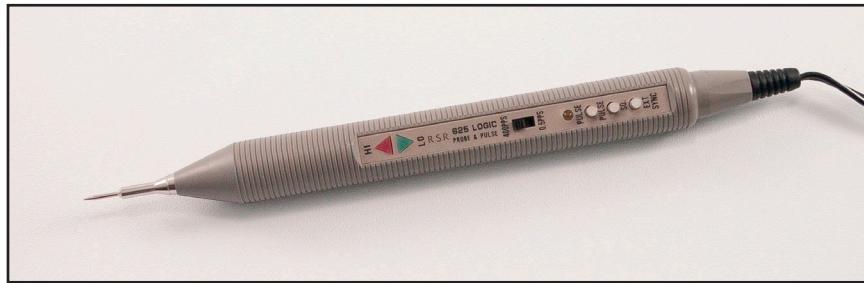


Figure 2.4 LP-900 Logic Probe



Figure 2.5 HP Logic Probe

Product Type	Retailer	Item No.	Price
LP-525K Logic Probe Kit	Ocean State Electronics	LP-525K	\$15.95
LP-550 Logic Probe	Ocean State Electronics	LP-550	\$18.95
LP-900 Logic Probe	Ocean State Electronics	LP-900	\$27.95
Logic Probe	Jameco	149930	\$29.95

Table 2.2 Logic Probes

Breadboard

Before building the replica, we're going to do a few basic experiments in electronics. A breadboard is almost essential for these exercises and is also ideal for making quick prototypes. Breadboards come in a variety of sizes. In Figure 2.6, you'll see three models that Circuit Specialists sells. The medium unit on the left is what I recommend. It's large enough to handle all the projects in this book and has multi-way binding posts for connecting to a power supply. The smaller unit on the right will work just as well, but it's not as convenient.

**Figure 2.6** Breadboards

Size	Retailer	Item No.	Price
Small	Circuit Specialists	WB-102+J	\$6.25
Medium	Circuit Specialists	WB-104-1+J	\$12.99
Large	Circuit Specialists	WB-108+J	\$22.90

Table 2.3 Breadboards

Wire-Wrap Tools

Wire-wrapping is a useful technique to use if you have unique designs that you want to build, play with for a few months, and then take apart. Breadboards are much more convenient for quick, temporary designs, whereas circuit boards are more reliable for long-term setups (especially if your wire-wrapping abilities are anything like mine). If, however, you're designing your circuit as you're building it, or you just don't have the patience to wait around for a printed circuit board, wire-wrap may be your best option.

A basic wrapping tool from Radio Shack (Table 2.4) works considerably well. Be sure to get a few different colors of wire so you can tell them apart. There is nothing worse than trying to trace a tangled mess of thousands of wire-wrapped lines – this cheapskate knows!

If you expect to do a lot of wire-wrapping, consider getting a Cooper automatic wire-wrapping tool. These sell as new for a few hundred dollars, but can often be found at very affordable prices on eBay.

Finally, you're going to need sockets to hold the chips, as well as a wire-wrap board. Sockets cost anywhere from \$0.80 to a little over \$3.00, which means that they can add up quickly. For circuit boards, try the 32-DE-STD wire-wrap board from Douglas Electronics, which is available for \$21.53. Douglas is a reliable company. I have one of its brochures from 1983 in which the same board is advertised with the same part number – and the price has only gone up \$1.53 in the last 29 years.

Product Type	Retailer	Item No.	Price
Wire-Wrapping Tool	Radio Shack	276-1570	\$6.99
Red Wire-Wrap	Radio Shack	278-501	\$3.29
White Wire-Wrap	Radio Shack	278-502	\$3.29
Blue Wire-Wrap	Radio Shack	278-503	\$3.29
8-pin DIP Socket	Radio Shack	900-7242	\$0.80
14-pin DIP Socket	Radio Shack	900-7243	\$1.20
16-pin DIP Socket	Radio Shack	900-7244	\$1.35
18-pin DIP Socket	Radio Shack	900-7245	\$1.56
20-pin DIP Socket	Radio Shack	900-7246	\$1.75
24-pin DIP Socket	Radio Shack	900-7249	\$2.00
28-pin DIP Socket	Radio Shack	900-7250	\$2.42
40-pin DIP Socket	Jameco	41187	\$3.15

Table 2.4 Wire-Wrap Tools

Soldering Iron & Materials

Soldering components to a circuit board is the most reliable and permanent method for building your Apple I replica. While printed circuit boards can be potentially expensive (see Circuit Boards and Software Tools later in this chapter), the soldering equipment itself can be very affordable. A good soldering job on a printed circuit board will result in a product indistinguishable from a professionally-made board.

When choosing a soldering iron, make sure to get one with adequate power. An inexpensive 15-watt iron (Figure 2.7), which can be found for around \$5, can be adequate, but is exceedingly frustrating to use if you don't know what you're doing. A 40-watt iron can be purchased for under \$10 and will save you a world of frustration. These "pencil irons" are contained entirely within the handle. In Figure 2.8 you can see the simple coil-heating mechanism of a standard pencil iron.



Figure 2.7 15-Watt Soldering Iron



Figure 2.8 Disassembled Soldering Iron

If you plan to do a lot of soldering, consider a more expensive, temperature-controlled iron. Temperature-controlled irons allow you to precisely set the temperature of your iron and ensure that it stays within a narrow range. Shown in Figure 2.9 is a Hakko FP-102 soldering station. It, like similar high-end soldering irons, is very reliable and pleasant to work with.



Figure 2.9 Hakko FP-102 Soldering Station

Soldering Safety

Don't neglect safety. Molten solder to the eyeball is sure to ruin your day; so, always wear safety goggles. Likewise, be sure to work in a well-ventilated room and keep a fan on to blow the fumes away from you. And if the iron starts to fall—*don't* try to catch it!

Solder is inexpensive and can be found at any electronics store. However, be certain that you purchase rosin-core and not acid-core solder. Even though it's doubtful you'd find it in an electronic store, acid-core solder will ruin your circuits.



Figure 2.10 Solder

Mistakes are made, and components die. Sometimes it's necessary to remove a component that has been soldered down. There are two different methods for doing so. One is to heat the solder and suck it into a desoldering pump (Figure 2.11). The other method is to wick the solder out with a desoldering braid (Figure 2.12). The braid is placed against the solder and then heated. As the solder melts, it diffuses into the braid.



Figure 2.11 Desoldering Pump



Figure 2.12 Desoldering Braid

Product Type	Retailer	Item No.	Price
Hakko FX-951	Hakko USA	FX-951	\$224.99
40-Watt Pencil Iron	Radio Shack	64-2071	\$8.39
15-Watt with Grounded Tip	Radio Shack	64-2051	\$8.39
Xytronix Soldering Iron, 40W	Jameco	224602	\$9.95
Rosin Core Solder—8.0 Oz.	Radio Shack	64-007	\$7.29
Solder Roll—1.1 Lbs.	Jameco	141786	\$7.95
Desoldering Tool	Radio Shack	64-2098	\$7.29
Desoldering Pump	Jameco	19166	\$4.95
Desoldering Braid	Radio Shack	64-2090	\$3.19
Desoldering Braid	Jameco	63695	\$1.95

Table 2.5 Soldering Supplies

Power Supply

To power your Apple I replica, as well as your other experiments, you're going to need an AT-style computer power supply. An AT power supply is distinguished by the connectors shown in Figure 2.13.

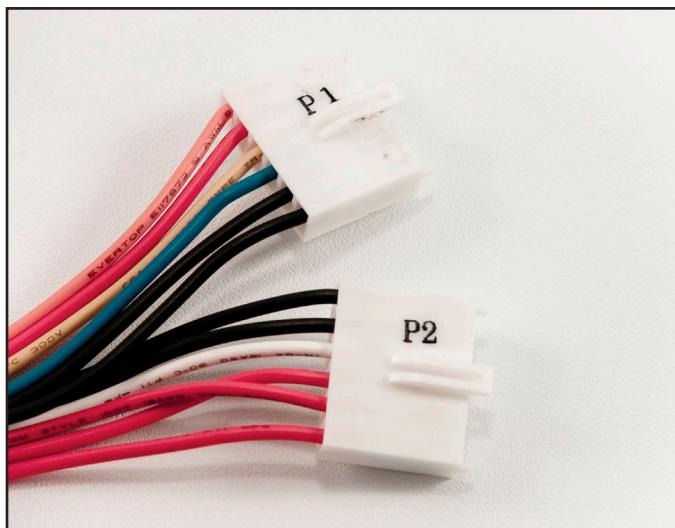


Figure 2.13 AT Motherboard Connectors

An AT power supply (Figure 2.14) should be fairly easy to find. Any pre-ATX computer will have one, and systems like those get thrown out every day. Go to a Hamfest (www.arrl.org/hamfests.html) and you'll find one for no more than a couple dollars. On eBay you can purchase one for around \$5, and new ones can run anywhere from \$35 to \$50. When looking at power supplies, you'll usually see the wattage rating advertised. For our simple project, any rating will work. If you already have a spare ATX power supply, you only need to find an adapter similar to the one pictured in Figure 2.15.

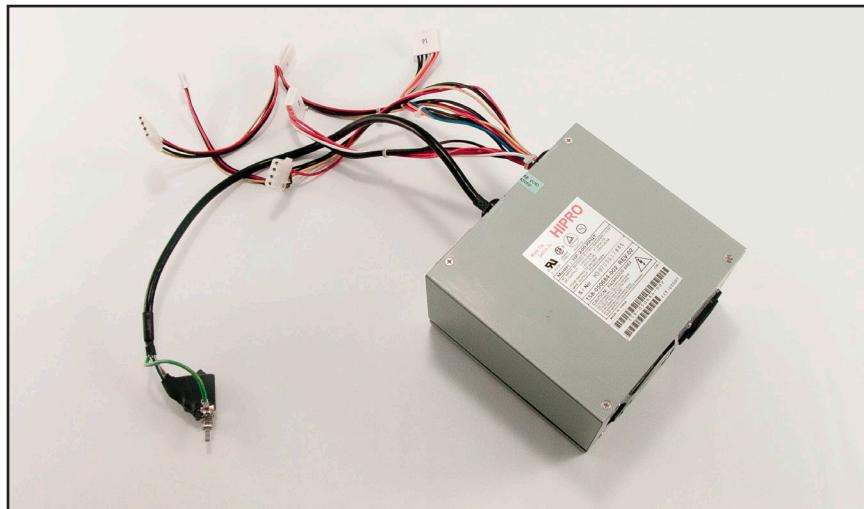


Figure 2.14 AT Power Supply

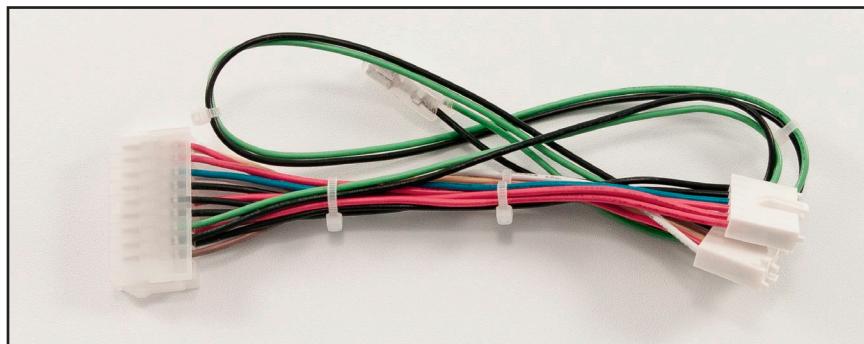


Figure 2.15 ATX-to-AT Adapter

AT supplies are “switching” units and generally require a load to operate properly. With nothing connected to it, check the output voltage on the 5-volt (red) line of your power supply. It should be 5 volts. If it’s not, try connecting it to something like a spare hard drive in order to create a load and regulate the voltage. With the AT power supplies I’ve used, this hasn’t been necessary, but it’s a good idea to check before you connect anything important.

TTL Chips

We'll be doing many experiments in the next few chapters, so it's good to have chips readily available. The chips you'll need to complete the introductory projects are listed in Table 2.6.

IC	Type	Jameco Part No.	Price
74LS00	NAND	46252	\$0.29
74LS02	NOR	46287	\$0.29
74LS04	INV	46316	\$0.29
74LS08	AND	46375	\$0.36
74LS32	OR	47466	\$0.29
74LS74	Flip-Flop	48004	\$0.29
74LS86	XOR	48098	\$0.35

Table 2.6 Needed TTL Chips

TTL chips will be labeled with 74xx, where "74" is the series of chips we'll be using and "xx" is a number which specifies each particular chip. Often you'll see a few letters in between the "74" and "xx," such as 74LSxx. These classifications are described as follows:

- **74Lxx, low-power** Requires 1/10 the power and runs at 1/10 the speed of a standard TTL IC
- **74Hxx, high-power** Requires twice the power and operates at twice the speed
- **74Sxx, Schottky** Runs at higher speed
- **74LSxx, Low-power Schottky** Runs at high speed and requires 1/5 the power
- **74ALSxx** Advanced Low-power Schottky
- **74HCxx** CMOS chips; equivalent to the CMOS 4xxx series

TTL Background

Transistor-Transistor Logic (TTL) is a method for creating the logic gates in integrated circuits by using two transistors at the output circuit. The primary alternative to TTL is Complementary Metal Oxide Semiconductor (CMOS). CMOS circuits use less power than TTL but are also more prone to static damage.

Another option is to take the chips out of old computers and other electronics that are being discarded. Don't let any piece of hardware make it to the trash without being thoroughly dismantled. Circuit boards are always a great place to find chips, while printers and disk drives are a great source for small motors.

Circuit Boards and Software Tools

Printed circuit boards (PCBs) usually offer the greatest reliability, but producing them is an involved process. If you want your Apple I replica to use a printed circuit board, your most economical choice is to use a Replica I board from Briel Computers (www.brielcomputers.com). Purchasing one of these boards is much faster and cheaper than designing and building your own, or having a PCB manufacturer make you a prototype.

If you decide you want to modify the designs presented here, you can find all the necessary files in the Supplemental Software package. The files can be edited, and the boards designed, with the included McCAD Schematics and PCB software. Documentation and a training video for the software can be found at www.mccad.com.

McCAD PCB will produce industry-standard files which you can then send to a manufacturer for printing. Expect to pay about \$100 for the service of having your board printed. A selection of PCB manufacturers is listed in Table 2.7.

PCB Express	www.pcbontime.us
Overnite Protos	www.pcborder.com
ExpressPCB	www.expresspcb.com
PCBExpress	www.pcbexpress.com

Table 2.7 PCB Manufacturers

Chip Pullers and Straighteners

As you work, you'll frequently find yourself in a situation where you want to remove chips from their sockets. The best way to do this is to take a small flat-blade screwdriver, insert it under the chip, and gently pry upwards. If the chip is soldered onto the board, you can use an IC extraction tool (about \$2) to gently pull it up as you desolder. However, you should not use it to remove socketed chips as it will put pressure on solder joints.

One tool that's very helpful to have is a chip straightener. Invariably, the pins on the chip you're attempting to insert will be bent out just a little bit further than what you need. Bending them back into position can be an onerous task without a straightener.

Product Type	Retailer	Item No.	Price
IC Pin Straightener	Jameco	99362	\$6.95
DIP IC Extractor	Jameco	16838	\$1.49

Table 2.8 Chip Pullers and Straighteners

Chip Sockets

Sockets (Figure 2.16) allow you to easily insert and remove chips without soldering them directly to the circuit board. Instead of the chip, the socket gets soldered to the board. This facilitates the replacement of chips and also spares them from the extreme heat of being soldered in directly.

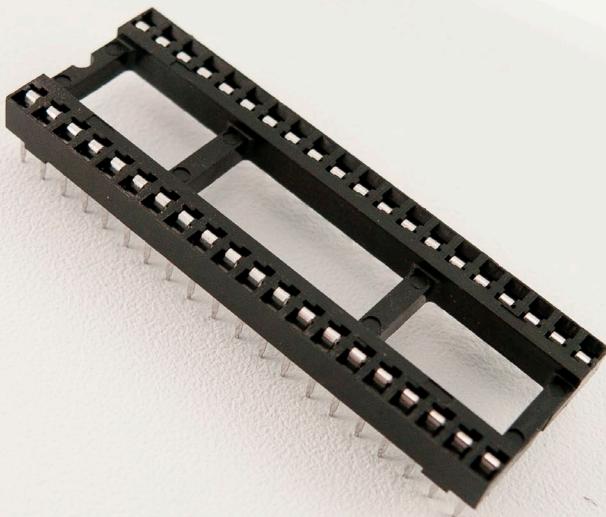


Figure 2.16 DIP Socket

Keyboard and Monitor

Whether you end up building a replica Apple I from a kit or using your own design, you're going to want a keyboard and monitor. The original Apple I used an ASCII keyboard, and this option remains today. Unfortunately, ASCII keyboards are all but impossible to find. Figure 2.17 shows an ASCII keyboard taken from an Apple II+, but these machines are becoming collectibles in their own right. As an alternative, Vince Briel designed an interface for a PS/2 keyboard, which is now part of his Replica I design. With this, any modern PS/2 keyboard, such as that in Figure 2.18, can be used with the Replica I.



Figure 2.17 ASCII Keyboard



Figure 2.18 PS/2 Keyboard

Video is output as a standard composite signal. If you have an old composite monitor (Figure 2.19) such as from an Apple IIe or Commodore 64, it will work perfectly. Otherwise, composite monitors can also be found on eBay for a pittance, or a television with composite video input will work just as well.



Figure 2.19 Video Monitor

Ambiance

What are your neighbors doing right now? Probably something ordinary. By contrast, you're designing a computer in your garage, bedroom, or basement. Wherever you are, it's important that you begin referring to this location as your "laboratory." Pronunciation is equally important. Lower your voice and deeply enunciate each syllable. Practice explaining to your parents or spouse that you are "working in your laboratory." Because, let's face it, ordinary folks don't build computers.

Now that you've got a laboratory (did you pronounce that correctly?), a lab coat is clearly in order. There are a plethora of new lab coats on eBay, often factory seconds, that sell for under \$10. A lot of lab coats now come in blue, but I strongly recommend white for that quintessential mad scientist look. Order a spare—you never know when you might have company.

Let's not forget safety glasses! Sure, we mentioned them earlier in regards to soldering, but safety glasses can be quite stylish too. For the best reactions from others, be sure to get colored lenses. Mine are tinted green.

Finally, we need music—something ominous to set the mood. I asked my fellow mad scientists what they listen to and came up with a few suggestions. Dr. Webster recommends Carl Orff's classical *Carmina Burana*. Reverend Darkness is a fan of *Night on Bald Mountain* as well as *Toccata and Fugue in D Minor*, played by an orchestra (not by the usual pipe organ). Dead_elvis listens to *Danse Macabre* by Saint-Saëns and Mozart's *Requiem Mass*. Also worthy of consideration are the soundtracks to *The Twilight Zone* and *2001:A Space Odyssey*. Stuka recommends the MIDI soundtrack from the game Doom, which is available for free online.

Chapter 3

Digital Logic

In this Chapter

- Breadboarding
- Electricity
- Gates
- Circuits and Algebra
- Latches and Flip-Flops
- What is Data?
- A Few More Chips

Introduction

This chapter provides an introduction to digital logic, which is the basis of micro-computer design. Although it is not critical that you understand everything in this chapter, in order to gain a full understanding of the microcomputer, a background in digital logic is imperative. Don't be afraid to move on if you don't understand a particular section. Each level we'll be covering (digital logic, microcomputer, and software) contains a certain degree of abstraction from the layer following it.

Breadboarding

The experiments in this chapter will all be done using breadboards. To begin, perform the following steps:

1. Find a power supply and snip off one of the hard drive connectors.
2. Strip down the end of the red wire (5 volts) and the black wire (ground).
3. Turn on the power supply and check these cables with your multimeter to ensure that they really are 5 volts and ground.
4. On your breadboard, you'll see a series of red and blue lines running the length of the plastic. Connect the power supply cables to the binding posts, then use a couple pieces of spare wire to connect the binding posts to these lines (Figure 3.1).

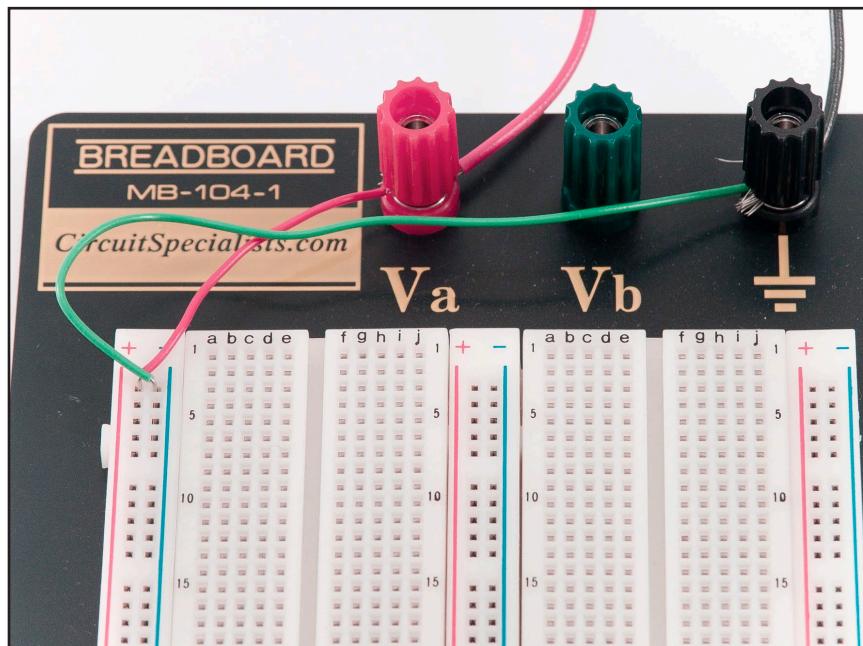


Figure 3.1 Connecting Power and Ground

The holes beside these lines are for power and ground, respectively. A look inside might help us understand how breadboards work (Figure 3.2).

Inside each hole is a metallic socket, into which you can plug a wire or component. Every socket in the ground strip is connected; if you connect one socket to your ground supply, all sockets in the strip are grounded. The same rule applies to the power strip. We have three columns of power and ground strips with nothing connecting them, so we need to use jumper wires across the bottom (Figure 3.3).

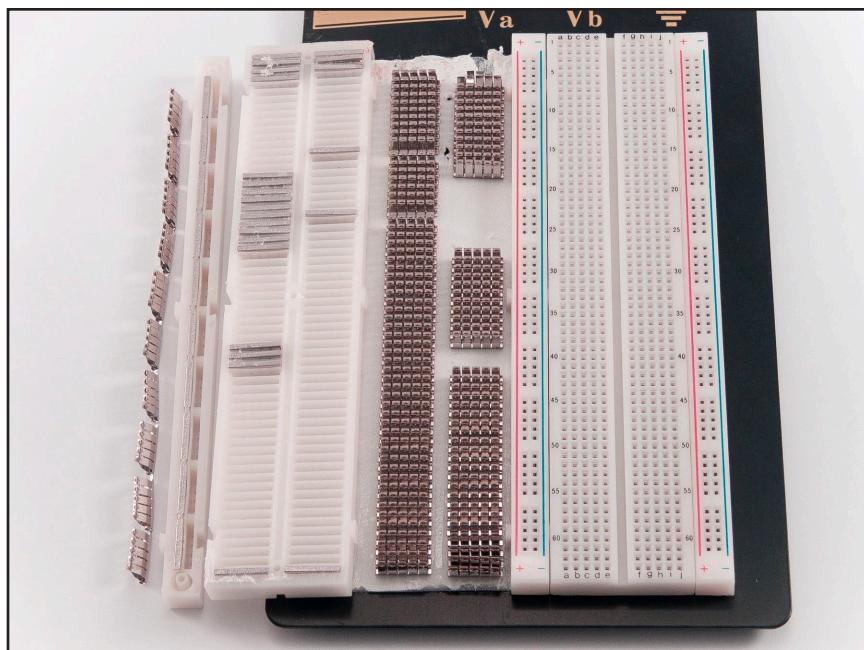


Figure 3.2 Inside the Breadboard

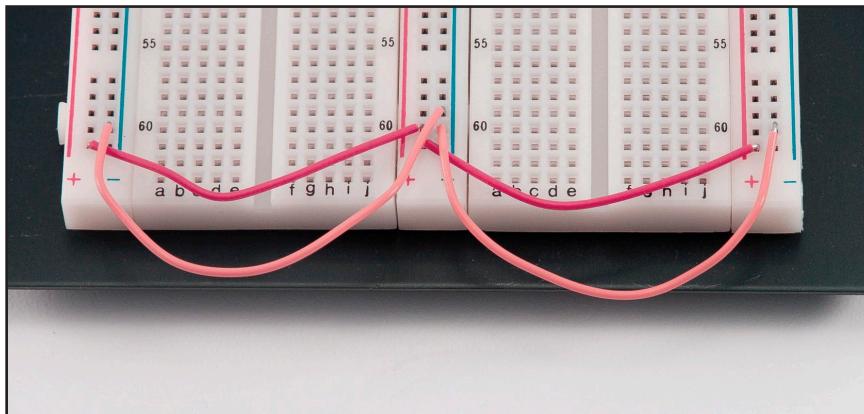


Figure 3.3 Jumpering Power and Ground Lines

In order to help gain an understanding of the rest of the board, take a look at Figure 3.2 once again. Each horizontal row of five sockets is linked. Therefore, if we wanted to connect three resistors in a series (one after the other), we could configure them as shown in Figure 3.4.

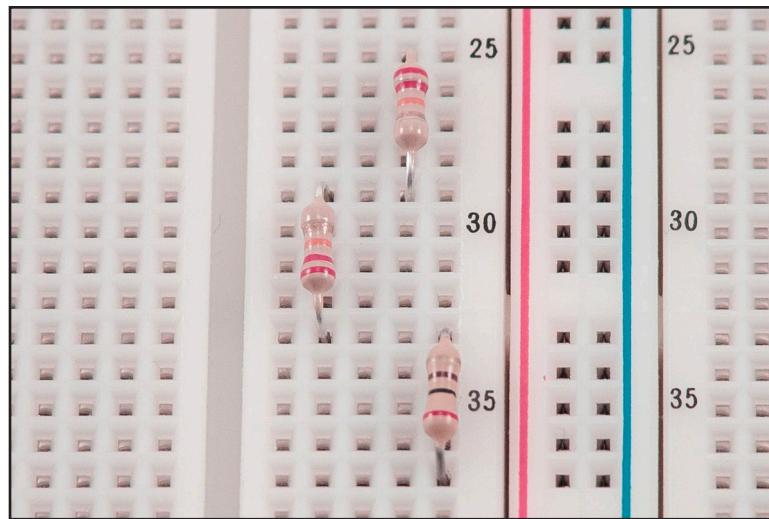


Figure 3.4 Resistors in Series

Next, let's make sure everything's configured correctly by running some tests using a logic probe:

1. Connect your probe's alligator clips to the power and ground lines (Figure 3.5).
2. Touch the point of the probe to any socket on the power line. The "HI" light should come on.
3. Test a socket in each strip by reapeating step 2.
4. Repeat step 2 yet another time for the ground sockets; for these, the "LO" light should come on (Figure 3.6). The sockets connected to neither power nor ground should produce no light at all.

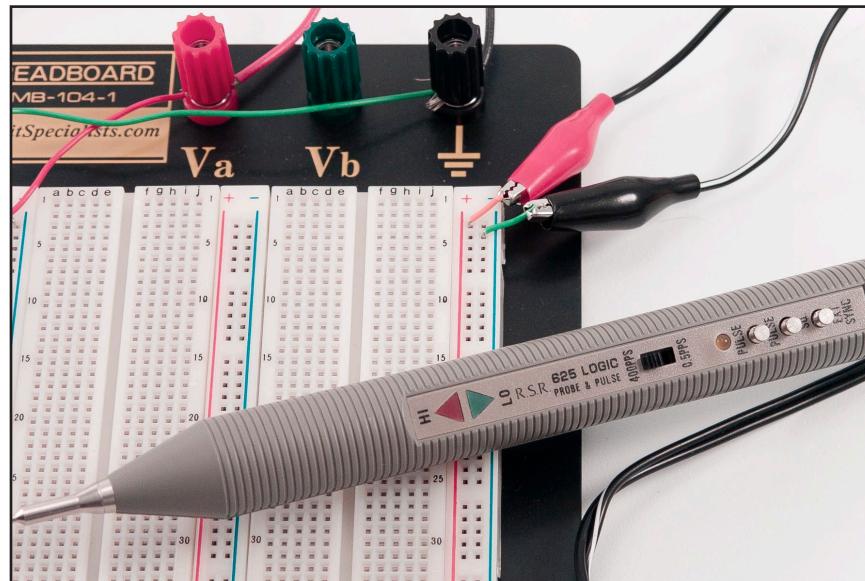


Figure 3.5 Connecting the Logic Probe

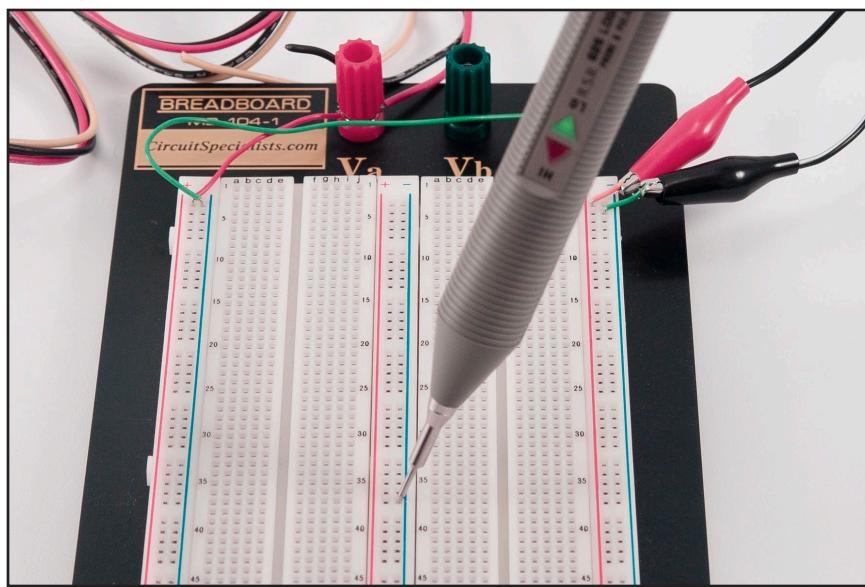


Figure 3.6 Testing with the Logic Probe

Electricity

Next you'll find a very brief, hands-on introduction to electronics fundamentals. In this chapter, I've provided enough of the basics that you shouldn't have too much difficulty completing your Apple I replica; however, a more substantial understanding of electronics will serve you well in the future. If you'd like to do some further reading, *The Art of Electronics* by Horowitz and Hill (Cambridge University Press, 1989) is the most widely acclaimed book on the topic but is perhaps too difficult for beginners. A more appropriate book for the novice is *Hands-On Electronics* by Kaplan and White (Cambridge University Press, 2003).

Voltage and Current

Voltage (V) is the potential difference between two points, measured in volts. When you take your multimeter, set it to Volts DC, and touch the probes to the breadboard — negative to negative, and positive to positive — you'll get a reading of 5 volts. That's the amount of energy it takes to move charge from the lower point (ground) to the higher.

Current (I) is used to measure the rate of flow of an electrical charge and can be thought of like the flow of water. Current is measured in amperes or amps (A). Whereas voltage is measured *between* two points, current measures the rate of flow *at* a particular point. While DC voltage is steady (hook your circuit up to a 5-volt supply and 5 volts is

what you'll get), current only “pulls” as much as is needed. If you have a power supply capable of 10 amps and your circuit only needs 5 amps, then 5 amps is all that it will pull.

Power (P) is voltage multiplied by current. It's measured in watts, which you're probably familiar with seeing on light bulbs. In fact, light bulbs serve as an excellent example. Consider a 20-watt and a 50-watt halogen bulb (Figure 3.7). Both require 12 volts. The difference is in the current, which we can calculate using $P=V\times I$.

$$I = P \div V \quad 20 \text{ watts} \div 12 \text{ volts} = 1.6 \text{ amps} \quad 50 \text{ watts} \div 12 \text{ volts} = 4.16 \text{ amps}$$

Appropriately, we see that the brighter light bulb draws significantly more current than the dimmer one.

Exercise: Replace all the 40-watt bulbs in your house with 200-watt bulbs. See what happens to your electricity bill.

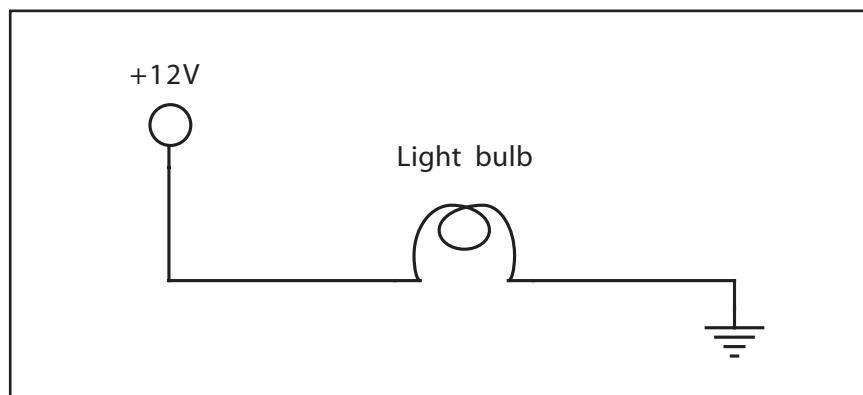


Figure 3.7 Light Bulb Circuit

Intersections

If you have two wires you want to connect in a schematic, place a large dot at the point of connection. If there is no dot, readers (and the schematic software) will assume there is no physical connection between the two lines. See Figure 3.8 for an example.

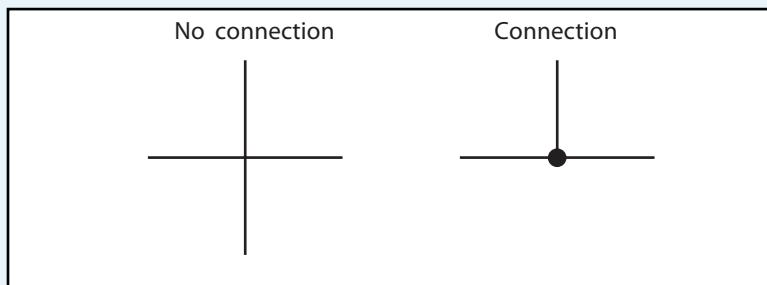


Figure 3.8 Intersections

Resistors and Diodes

Now that we've explained the basics of current and voltage, we can take a more hands-on approach to resistance. Let's wire up a light-emitting diode (LED). A *diode* is a device that only allows current to flow in one direction. For this experiment, let's use a standard red LED, such as model #276-041 from Radio Shack. These LEDs have a recommended voltage of about 2.25 volts, with a maximum voltage of 2.6 volts (you'll find all of this information on the packaging). The maximum current is 28 mA. Our power supply provides 5 volts, so if we hook the LED up directly to it, the LED will probably burn out. (If you have a spare, try this!)

To reduce the voltage, use a *resistor*. A resistor is a partial conductor, usually made of carbon. We can calculate resistance using the equation:

$$R = V \div I$$

where R (resistance) is calculated in Ohms, V (voltage) in volts, and I (current) in amps.

Our circuit is shown in Figure 3.9. Since we have 5 volts being supplied, and our LED uses 2.25 volts, that leaves 2.75 volts that we need to have a resistor take care of. 2.75 is the value we want to use for V in this equation. Now we can calculate:

$$R = (5V - 2.25V) \div .028 A = 98 \text{ Ohms}$$

If you calculated decimal places, you're taking these numbers too seriously. Next, we need to find a 98-Ohm resistor. A resistor with a smaller value will provide less resistance and potentially allow damage to your LED. One with greater resistance merely means that your LED won't be quite as bright. We're just experimenting, so brightness level isn't a major concern. I used the nearest resistor I had available, which happened to be 300 Ohms.

Now it's time to wire it up:

1. Snip off the ends of the resistor, making it fit conveniently into your breadboard.
2. Hook up the resistor and the LED in series. It doesn't matter which one you connect first; however, be aware that the orientation of the LED does matter (see the sidebar Diode Polarity). The finished circuit is shown in Figure 3.10.

Try swapping your resistor with some resistors of varying sizes, in order to see what happens. Also try putting the LED in backwards.

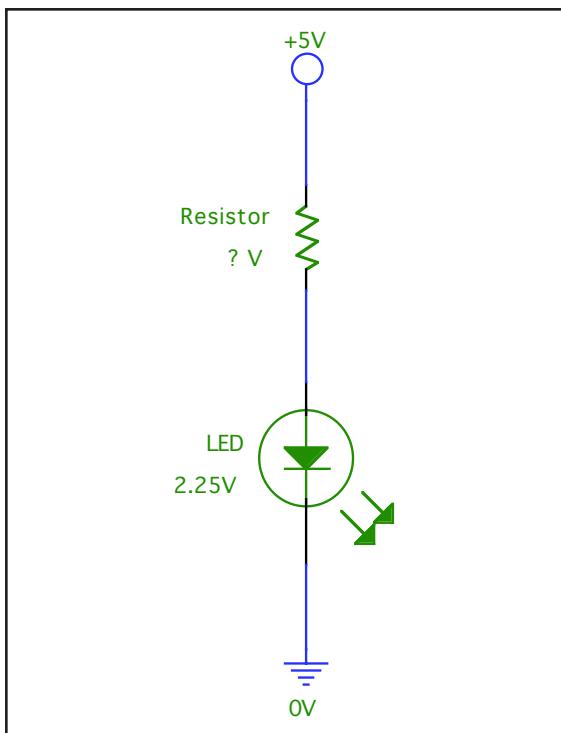


Figure 3.9 LED Circuit

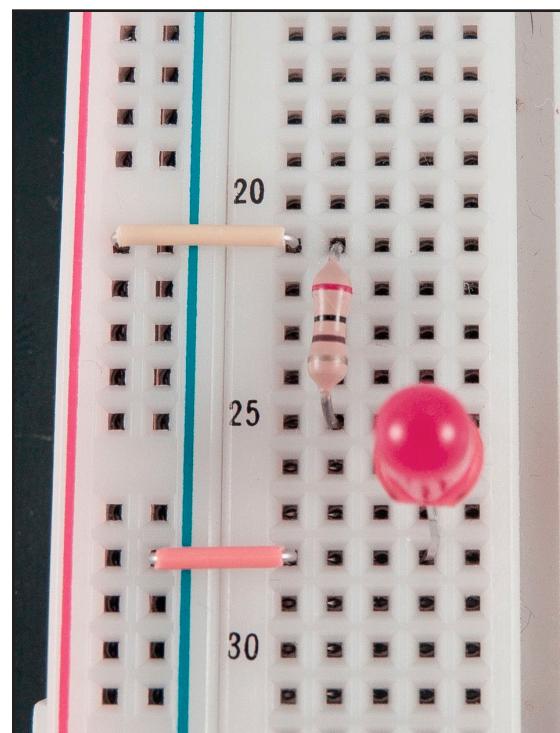


Figure 3.10 Wired LED

LED Polarity

An LED typically has two leads projecting from its base (Figure 3.11). The longer of the two is the positive lead (also known as the “anode”), and the shorter, the negative lead (also known as the “cathode”). The cathode is also denoted by a flat edge on the plastic LED housing. The anode will connect to positive voltage (5V in this case) and the cathode will connect to ground.

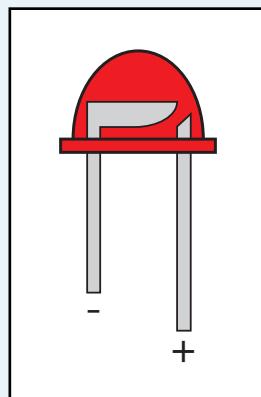


Figure 3.11 LED Diagram

Resistor Codes

Most resistors have four colored bars, which will give you the value in Ohms, as shown in Table 3.1. A resistor of yellow-red-red-gold would be 4,200 Ohms, with an accuracy of 5%. If we needed a 30-Ohm resistor, we would search for one with the colors orange-black-brown-gold.

Memorizing this chart will save you time if you're going to do a lot of work with resistors. At the very least, knowing the values for Bar 3 will help prevent you from using an exponentially incorrect resistor. If you don't have the time to memorize them or have no desire to haul the chart around with you, there are a multitude of Java applets on the Internet which will do the conversions for you. There are also many applications for graphing calculators that allow you to complete the conversion without the work. These include RCOL for the HP 48, which is available at hpcalc.org. Finally, you can also use your multimeter to measure the resistance.

Color	Bar 1	Bar 2	Bar 3	Bar 4
Black	0	0	x1	
Brown	1	1	x10	
Red	2	2	x100	
Orange	3	3	x1,000	
Yellow	4	4	x10,000	
Green	5	5	x100,000	
Blue	6	6	x1,000,000	
Magenta	7	7		
Gray	8	8		
White	9	9		
Gold			x0.1	5%
Silver			x0.01	10%

Table 3.1 Resistor Codes

Capacitors

A *capacitor* (Figure 3.12) stores energy in an electric field. In building our circuits, we'll be using these as de-spiking capacitors to filter the power supply. When the output of one of our chips changes, it causes a sudden voltage drop (a negative-going spike). When this occurs, the capacitor will partially discharge its energy to reduce the severity of this spike.



Figure 3.12 Capacitors

Gates

Those little black chips that cover circuit boards look extremely complex. Today, most of these *integrated circuits* (ICs) are quite intricate, but there are still many simpler chips available. If you examine the chips in a modern computer, you will notice that many of them have hundreds of pins, placed so closely that they appear almost impossible to work with. Open an older computer, and you'll discover chips that look much more accessible, with few pins and wider spacing. These older, more "classic" chips are what we'll be working with through the duration of this book. The 7400 series of ICs serves as the basis for digital logic and is very easy to understand. Throughout the rest of this chapter we'll concentrate on building a few very basic digital circuits using 7400-series ICs, resistors, and LEDs.

AND

IF it is dark AND IF there is a car in the driveway THEN turn on the porch light.

Under what circumstances is the porch light turned on? Only if both statements are true—it must be dark and there must be a car in the driveway. One of the two is not enough. In a schematic, we'd express this as seen in Figure 3.13. The “D” symbol is a logical AND. We can also use a truth table. Follow along with the truth tables in Table 3.2 and see if they match the aforementioned assertions. You'll note that all three tables contain the same data. True is a logical “1” which is implemented with a 5-volt signal. False is a logical “0” and is implemented by a 0-volt signal.

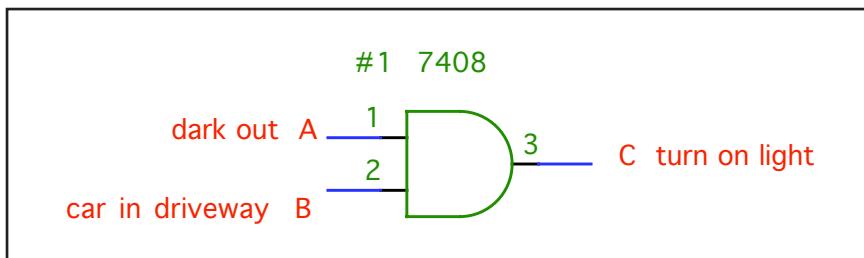


Figure 3.13 AND Gate

Input A	Input B	Output C
False	False	False
False	True	False
True	False	False
True	True	True

Input A	Input B	Output C
0	0	0
0	1	0
1	0	0
1	1	1

Input A	Input B	Output C
0V	0V	0V
0V	5V	0V
5V	0V	0V
5V	5V	5V

Table 3.2 AND Truth Table

With all of this in mind, perform the following steps:

1. Take a 7408 IC and insert it into your breadboard. This chip has four AND gates, arranged as shown in Figure 3.14. Note the location of the notch in the diagram.
2. Orient all chips so that the notch is at the top, and pin 1 will always be in the upper-left corner.
3. Hook up the supply voltage and ground. Pin 14 is Vcc (Voltage common collector) and should be tied to your 5-volt line. Pin 7, GND, should be connected to ground.

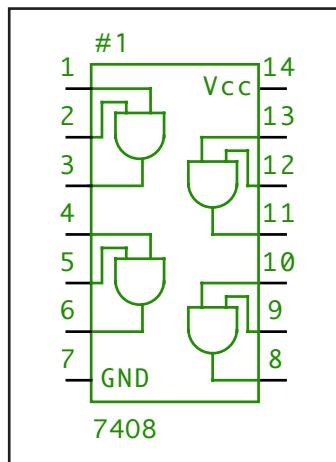


Figure 3.14 7408 IC

Pin Numbering

Pin numbering always starts at the upper-left corner, goes down the left side, and follows up the right side. Pin 1 is marked either by a circle in the upper-left corner, or by a notch in the top center (at which point it's up to you to know left from right). See Figure 3.15.

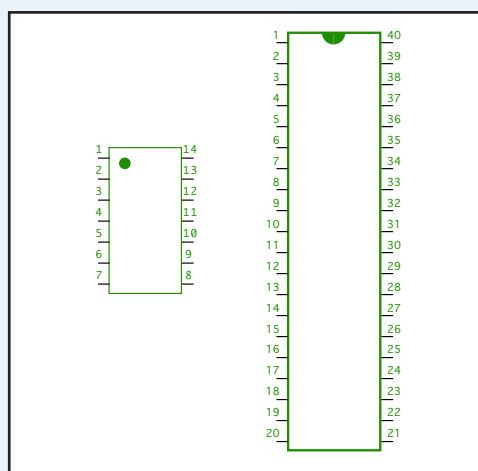


Figure 3.15 Pin Arrangement

Now let's wire the first AND gate on the chip:

1. Connect pins 1 and 2 (inputs to the AND gate) to the supply voltage line.
2. Connect pin 3 to the input of the LED that was wired up in Figure 3.10.
3. Turn on the power, wait a few nanoseconds for the signal to propagate, and the LED should light up (Figure 3.16). Since both Input A and Input B are high, the output will also be high and the LED will be powered on.
4. Take the wire from pin 2 (or pin 1) and move it from supply voltage to ground. To grasp the importance of this, imagine that instead of moving the cable by hand, it's hooked up to a light sensor that outputs 0 volts when it's bright outside, and 5 volts when it's dark.

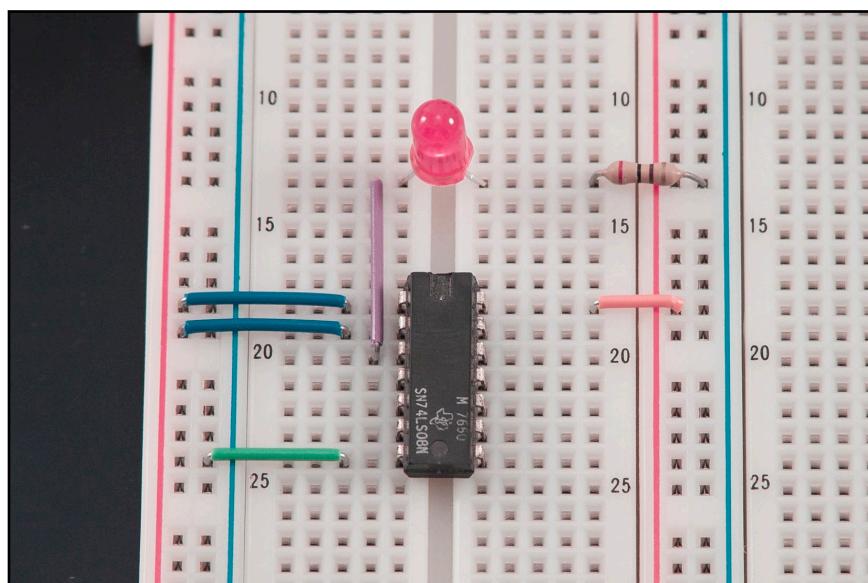


Figure 3.16 Using AND With an LED

Nanoseconds

When a chip's inputs are changed, the output does not change instantaneously. The new signals take time to propagate throughout the chip. This propagation varies depending upon the speed and complexity of the design, but TTL gates tend to take about 10 nanoseconds (ns) to complete propagation. There are 1,000,000,000 nanoseconds in a second.

If you like a challenge, try playing around with the chip a bit more by wiring the inputs of some gates up to the outputs of others. Create the equivalent of a 4-input AND gate. You can also examine the logic table in Table 3.3 to find patterns.

Input A	Input B	Input C	Input D	Output E
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

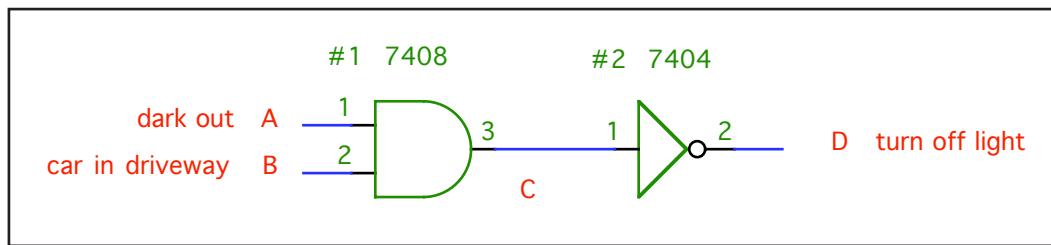
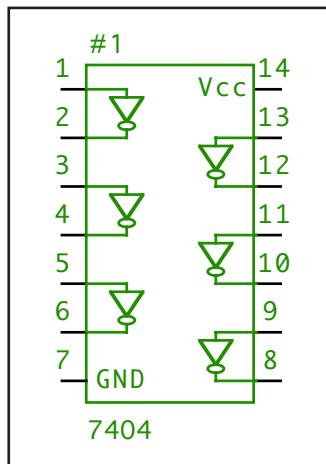
Table 3.3 4-Input AND Gate

Inverter, NAND

IF it is dark AND IF there is a car in the driveway THEN turn OFF the porch light.

This statement is almost identical to the one in the previous section, even though our goal has changed to producing the opposite effect. The only change that needs to be made is the addition of an *inverter* (see Table 3.4). The inverter takes whatever signal it is given and outputs the opposite; if you put 0 volts in, you'll get 5 volts out, and vice versa. In Figure 3.18, you'll see the symbol for an inverter along with an AND gate. The triangle means "buffer" (replicate the signal) and the small circle means "invert it." Find a 7404 (Figure 3.17) and connect it to supply voltage and ground. Now connect the output of your AND gate (pin 3) to the input of the 7404's inverter (pin 1). The output of the inverter (pin 2) goes to the LED to complete our circuit, shown in Figure 3.19.

Input A	Output B
0	1
1	0

Table 3.4 Inverter Truth Table**Figure 3.17** 7404 IC**Figure 3.18** AND Inverted

Input A	Input B	Output C
0	0	1
0	1	1
1	0	1
1	1	0

Table 3.5 NAND Truth Table

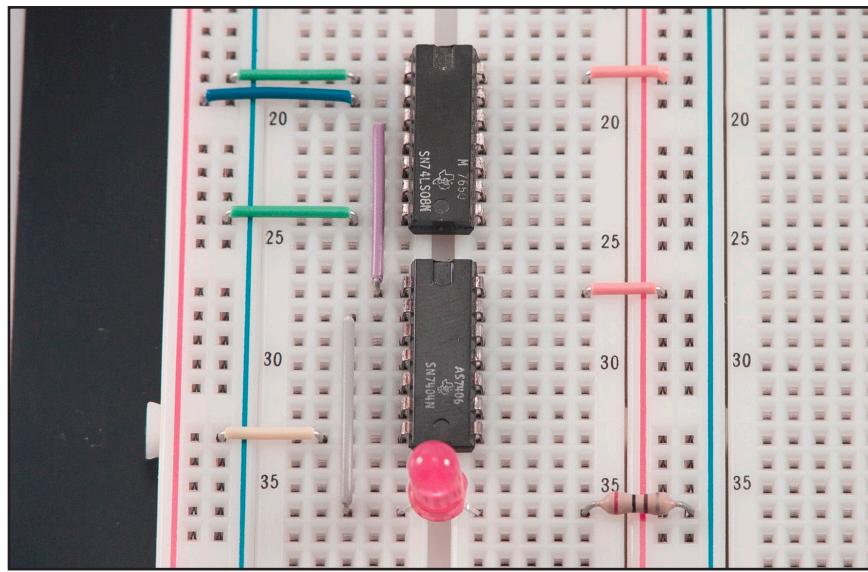


Figure 3.19 AND and NAND Gates, Wired to LED

The inverted AND is so common that it has its own gate, the *NAND* (Not AND) gate. This gate is functionally equivalent to an AND followed by an inverter. The NAND is pictured in Figure 3.20, with its layout in Figure 3.21. Table 3.5 displays its operation, which, hopefully, you were able to surmise.

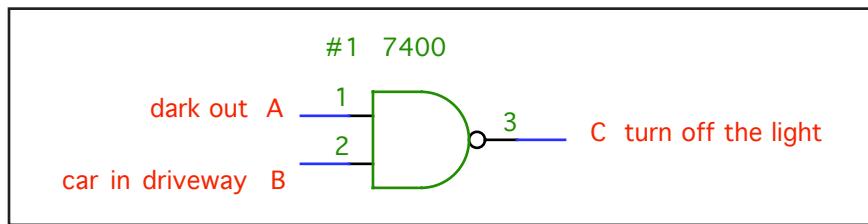


Figure 3.20 NAND Gate

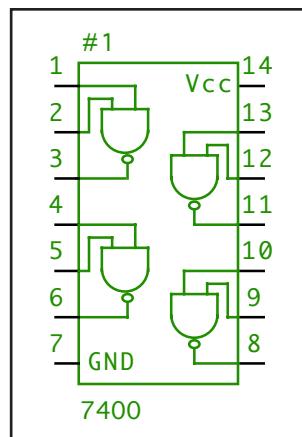


Figure 3.21 7400 IC

OR, NOR

IF it is dark AND IF there is a car in the driveway OR if the light switch is on
THEN turn on the porch light.

Here we take our output from the AND gate and OR it with “the light switch is on.” The OR gate in TTL logic is the 7432 (Figure 3.22), and exhibits the characteristics provided in Table 3.6.

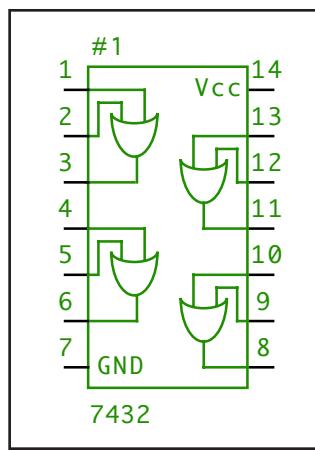


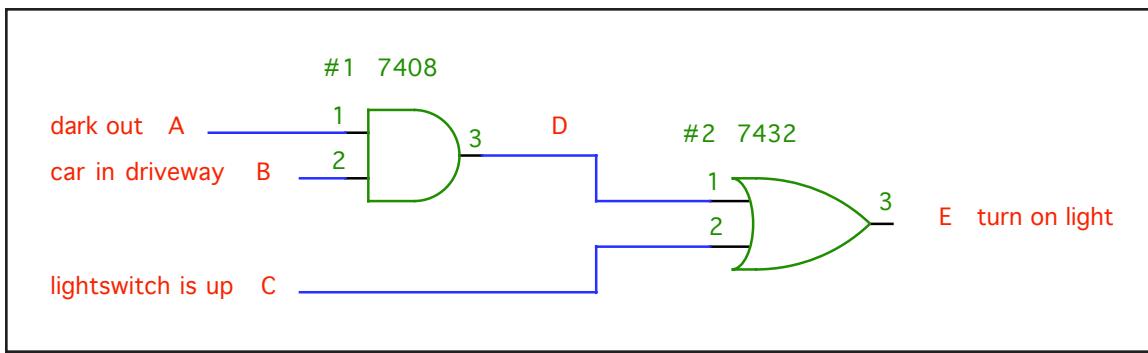
Figure 3.22 7432 IC

Input A	Input B	Output C (A OR B)
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.6 OR Gate Truth Table

Perform the following steps:

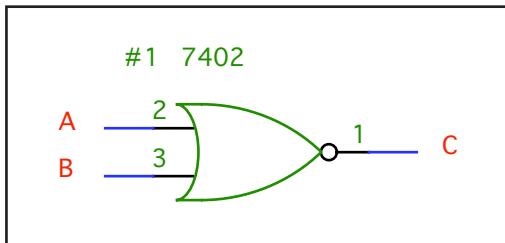
1. Take your 7432 and connect power and ground.
2. Using the circuit from Figure 3.16, hook the output of the AND gate to an input of the OR (pin 1). The other OR input (pin 2) should be connected directly to power or ground to simulate the light switch being on or off.
3. Next, wire the OR’s output (pin 3) to the LED (Figure 3.23). Table 3.7 describes the behavior of this entire circuit. Note that the light comes on for five out of the eight possible combinations.

**Figure 3.23** AND and OR, Wired

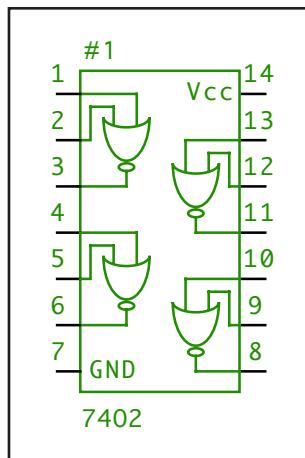
Input A	Input B	A AND B	Input C	Output E ((A AND B) OR C)
0	0	0	0	0
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	1	1

Table 3.7 Truth Table for Figure 3.23

Like NAND, there is also a NOR gate (Figure 3.24), which is OR followed by an inverter. You can find NOR gates on the 7402 (Figure 3.25).

**Figure 3.24** NOR Gate

Input A	Input B	Output C (A NOR B)
0	0	1
0	1	0
1	0	0
1	1	0

Table 3.8 NOR Gate Truth Table**Figure 3.25** 7402 IC

Combining outputs with OR

Let's say you have two AND gates and you want to combine their outputs so that if either AND gate's output is HI, our final output is HI as well. What happens if you connect the two output lines, as in Figure 3.26? When output A goes HI and output B goes LO, the two signals will compete, potentially damaging your chips.

If you're tempted to try this configuration, you're probably looking to use an OR gate, as in Figure 3.27.

continues...

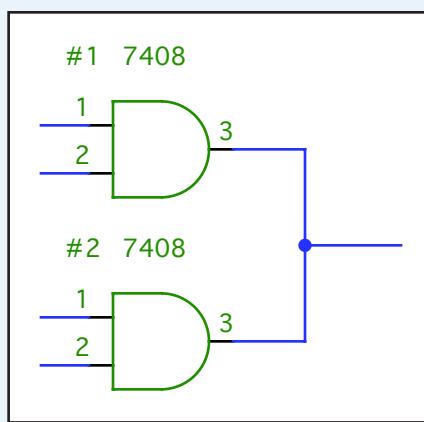


Figure 3.26 Bad Circuit

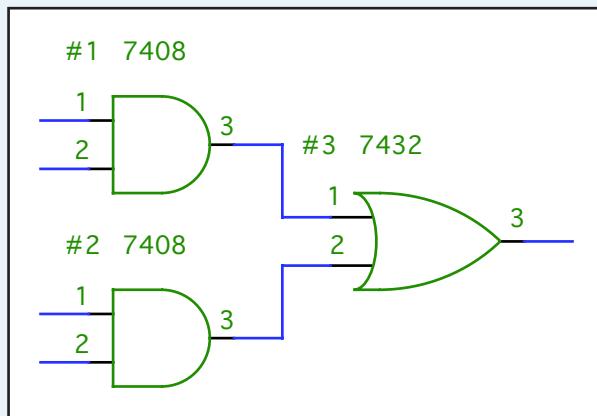


Figure 3.27 Good Circuit

XOR

IF it is dark OR IF there is a car in the driveway—but NOT if both—THEN turn on the porch light.

Perhaps this behavior would be desirable if you wanted to discourage visitors after dusk. XOR is short for “exclusive OR” and means “if one or the other is true, but not if both are true.” You can find XOR gates on the 7486 (Figure 3.28). In Figure 3.29 is the schematic for this circuit, along with an equivalent circuit that does not use an XOR gate but achieves the same effect.

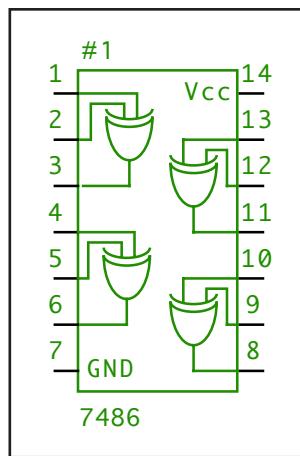


Figure 3.28 7486 IC

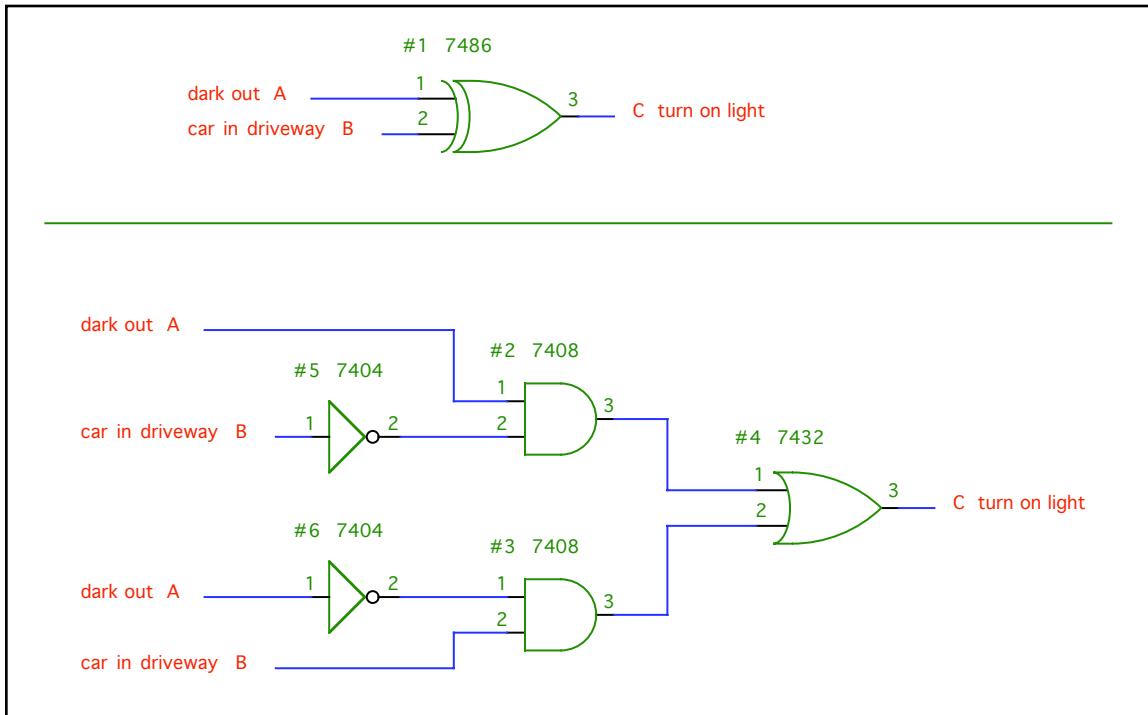


Figure 3.29 XOR Gate Circuit

Input A	Input B	Output C (A XOR B)
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.9 XOR Truth Table

Circuits with Algebra

Don't worry, it's not as bad as it sounds. Using logic expressions, DeMorgan's Laws, and Boolean algebra, you can quickly sketch out basic circuits on paper or even in a simple text editor. Logic expressions will allow you to show gates and lines with symbols and letters. DeMorgan's Laws will allow you to swap gates in order to get more efficient circuits. Boolean algebra will allow you to simplify your circuits.

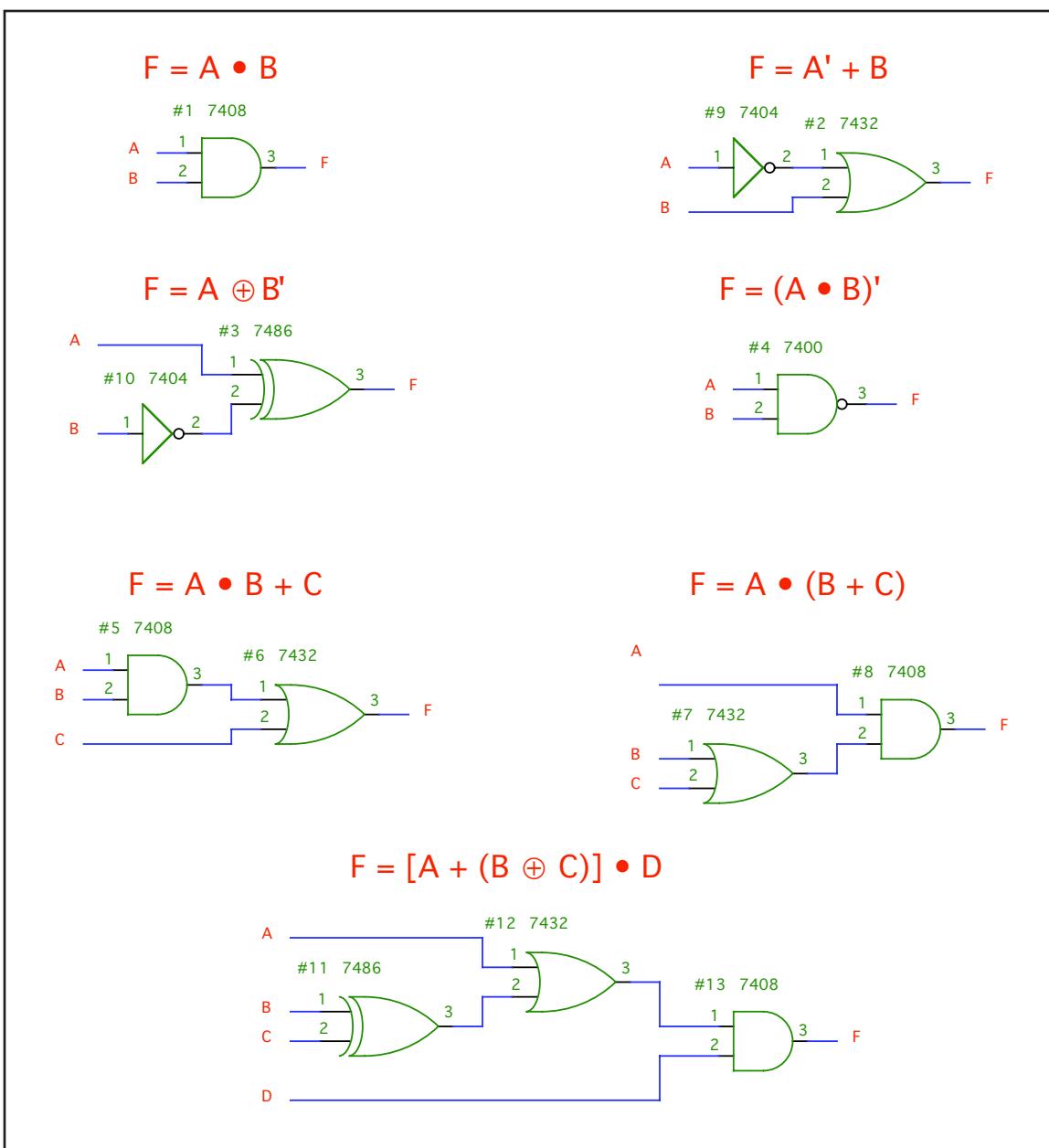
Logic Expressions

Ambiguities in the English language can make it very difficult to precisely express digital logic, so a specific algebra has been developed for this purpose. Understanding this algebra is merely a matter of getting accustomed to its symbols. Table 3.10 displays these symbols, listed in order of precedence:

Symbol	Meaning
'	NOT
•	AND
+	OR
⊕	XOR

Table 3.10 Algebraic Symbols for Logic Expressions

Figure 3.30 shows a few logic expressions and their equivalent circuits. Take a good look at these and make sure you understand them. Try writing a few of your own.

**Figure 3.30** Equivalent Circuits for Logic Expressions

DeMorgan's Laws

DeMorgan's Laws sounds dull and tedious, but they're a real boon for the improvising hacker. These laws explain how we can substitute different combinations of gates to best use our available resources. They can be expressed as:

$$(A + B)' = A' \cdot B'$$

$$(A \cdot B)' = A' + B'$$

Let's take $(A \cdot B)'$ for example (a NAND gate). This statement means that A and B are not both true; therefore, at least one of them is false. Consequently, it follows that A is not true or B is not true, which we can write as $A' + B'$. This looks like a slight change, but it can make a huge difference. Consider, for example, that the signals you are receiving may already be inverted or that you may not even have any NAND gates. Also, note that each of the chips we've been using contains multiple gates. DeMorgan's Laws allow us to more fully utilize the chips with which we're already working.

There's an easy way to remember DeMorgan's Laws in practice (Figure 3.31). To find the equivalent for any gate, first swap its symbol (OR to AND, AND to OR). Then, look at each input and output. Everywhere there's an inverter, remove it, and everywhere there isn't, add one.

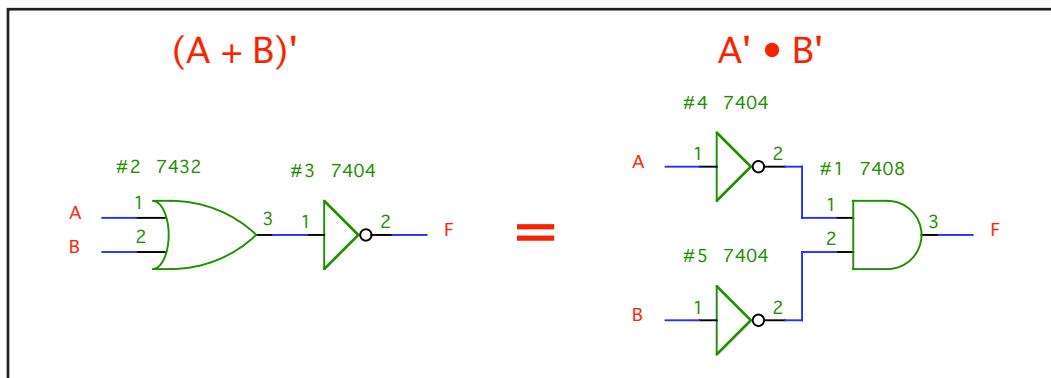


Figure 3.31 DeMorgan's Laws

Boolean Algebra

Boolean algebra is a method for manipulating logic expressions. It allows us to take complex expressions and reduce them to simpler statements that are logically equivalent. Some useful equivalencies are shown in Table 3.11.

$A \cdot 1 = A$	$A + 0 = A$
$A \cdot 0 = 0$	$A + 1 = 1$
$A \cdot A = A$	$A + A = A$
$A \cdot A' = 0$	$A + A' = 1$
$(A')' = A$	$((A')')' = A'$
$A \cdot B + A \cdot C = A \cdot (B + C)$	$(A + B) \cdot (A + C) = A + (B \cdot C)$
$A \cdot (A + B) = A$	$A + (A \cdot B) = A$
$A \cdot (A' + B) = A \cdot B$	$A + (A' \cdot B) = A + B$
$(A \cdot B)' = A' + B'$	$(A + B)' = A' \cdot B$

Table 3.11 Equivalences

Make sure you understand why all of these equivalences are true. Let's take

$$A + 1 = 1$$

as an example. The statement reads "if A is 1 or if 1 is 1." Given that "1" is always "1," this statement is always true and we can simply assert "1."

All You Need is NAND

We've already shown that XOR can be expressed as a combination of AND and OR gates (Figure 3.29). Likewise, NAND and NOR can be expressed by using an inverter along with AND and OR, respectively. Thanks to DeMorgan's Laws, we can use a combination of inverters and OR gates to produce an AND gate, or a combination of inverters and AND gates to produce an OR gate. At this point, we have a means of re-expressing every gate except the inverter. We can do that with a NAND gate. Connect both inputs of your NAND gate to the same line (we'll call it P). A NAND gate is expressed:

$$(A \cdot B)' = C$$

Since in our case, both A and B are the same line, we can write:

$$(P \cdot P)' = C$$

Using the equivalence

$$A \cdot A = A$$

given in the previous section, we can reduce this to

$$(P)' = C$$

which means that our NAND gate now functions as an inverter.

A *complete set* is a collection of chips that can be used to produce any logical statement. One example of a complete set would be the 7408 (AND), 7432 (OR), and the 7404 (NOT). Another example is the 7400 (NAND). Any circuit can be built by using nothing but NAND. This means that any computer of any complexity could be built using only 7400 ICs. This is not practical when you consider the colossal size such a computer would have to be, but it is interesting to consider the magnitude of the concepts you can express using nothing but NAND.

A second NAND gate is used to negate the negation, and we're back to an ordinary AND (Figure 3.32).

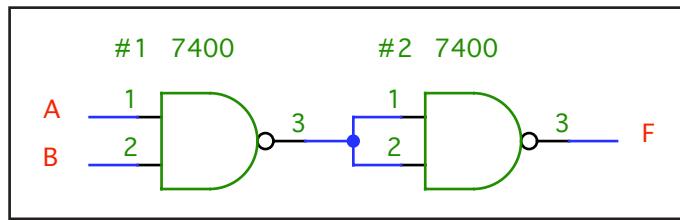


Figure 3.32 AND with NANDs

Thanks to DeMorgan's Laws, we know that an AND gate surrounded by inverters is equivalent to an OR (Figure 3.33).

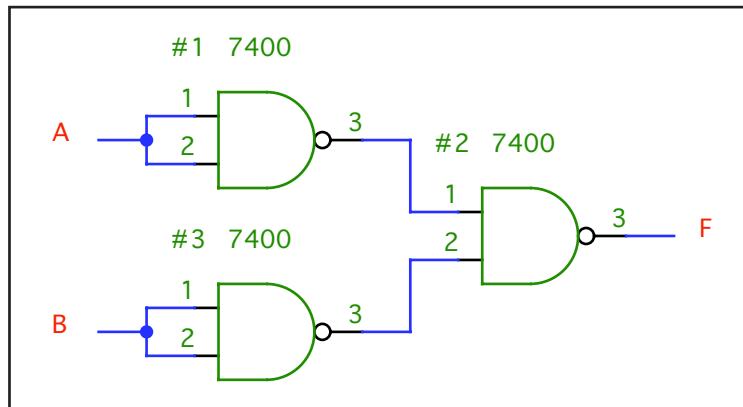


Figure 3.33 OR with NANDs

An XOR gate can be expressed as $(A \cdot B') + (A' \cdot B)$. Using that equivalency and the previous examples as parts, we can express an XOR using nine NAND gates (Figure 3.34). This circuit, incidentally, can be reduced to five NAND gates pretty easily and there's also a way to do it with just four.

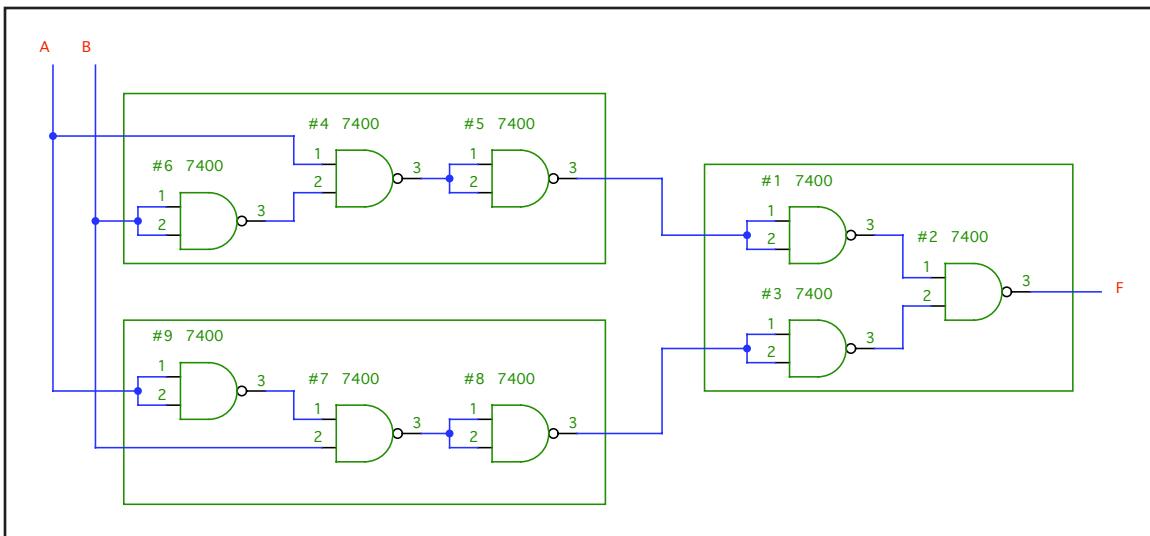


Figure 3.34 XOR with NANDs

Latches and Flip-Flops

You may have noticed a shortcoming in the circuits we've described thus far. What if we want the porch light to *stay on* after it detects a car in the driveway? The circuits we've covered have no memory. Now we're going to discuss ways of adding memory to our circuits, using only the gates we've covered in previous sections.

SR Latch

A set-reset (SR) latch allows us to “set” a bit to 1 or “reset” it to 0. We can do this using ordinary logic gates. The trick is to loop the output back into the input (Figure 3.35).

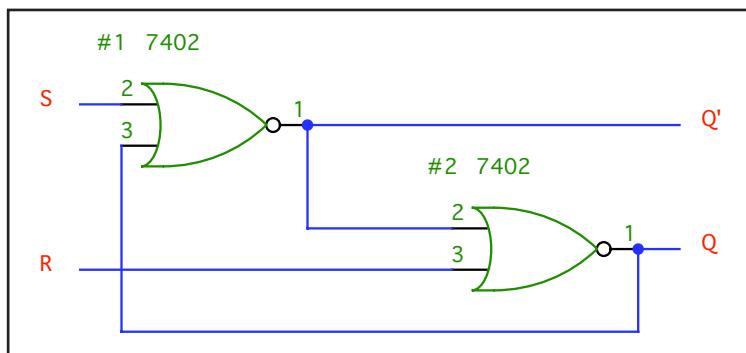


Figure 3.35 SR Latch

Grab a 7402 and wire up this circuit. Connect an LED to the output Q (Figure 3.36).

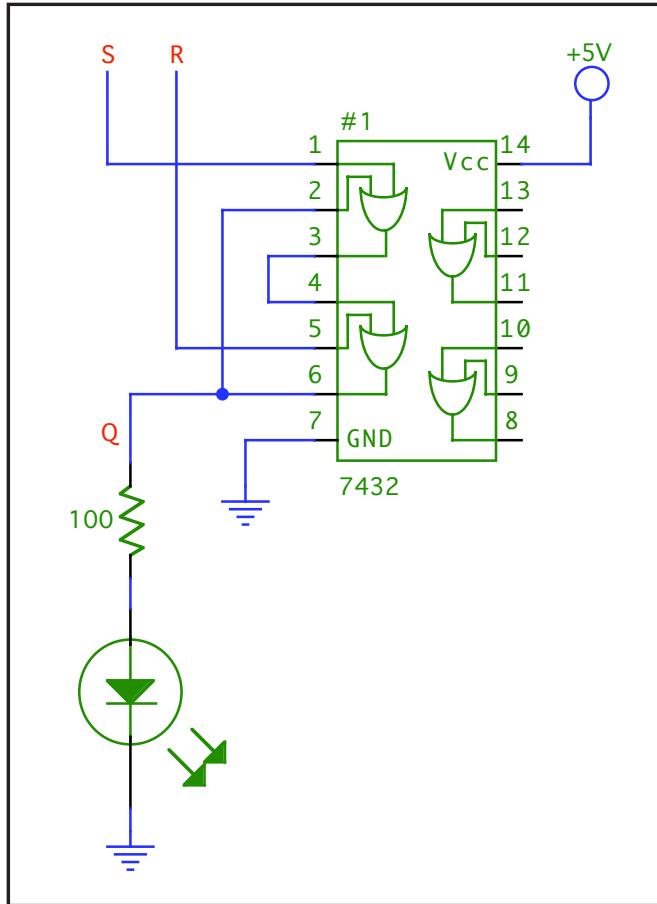


Figure 3.36 TTL SR Latch

This circuit is confusing because our outputs are determining our inputs. Consequently, the first thing we need to do when we power this circuit on is to “reset” it, so that we have a circuit with known outputs. Looking at the #2 NOR gate, we can see that the input R alone being high is enough to make the output Q low. In your circuit, wire R to high and S to low. The output Q will be low. Now, check the inputs to the #1 NOR gate. Both of these are low, making Q’ high. Q’ is an input to the lower NOR gate. This means that even when Reset goes low, the circuit will stay in the reset state.

Now, let’s take R low and make S high. The output of the #1 NOR becomes 0, which means both inputs to the #2 NOR are now 0, making the Q output 1. Make S low and Q stays high, because Q is being looped back into the input of the #1 NOR gate. Q will remain high until the circuit is reset.

Table 3.12 is the complete table of operation. What’s most important to realize here is that all it takes is a *momentary* high on the set or reset lines to *permanently* set or reset the output Q.

S	R	Q
0	0	Previous value of Q
0	1	0
1	0	1
1	1	not used

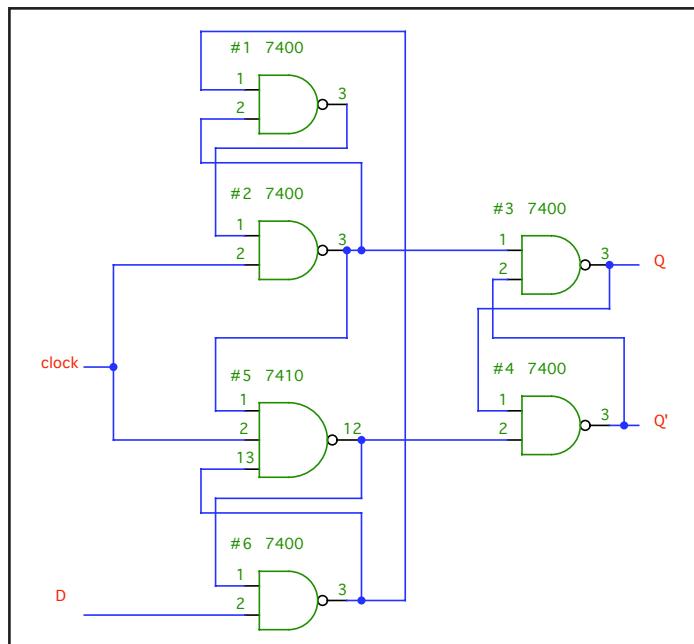
Table 3.12 Latch Operation

Flip-Flop

Imagine we have a situation where we want to grab a value at a certain instant and then hold onto it. For example, “every hour, check to see if the water level is above the indicator. If it is, sound the alarm for a full hour, then check again.” We can accomplish this using a *D flip-flop*.

Figure 3.37 shows the circuit in order to help you build it on your breadboard. We’re not going to examine its operation, but if you want to figure it out, follow the levels in your mind or with a logic probe. Normally, when you see a flip-flop in a schematic, it’ll look like Figure 3.38.

If you don’t want to wire up all those NAND gates, use a 7474 TTL flip-flop, which is shown in Figure 3.39. Notice the inverters at the inputs to the Preset and Clear lines. Due to their presence, we use a HI signal instead of a LO to deactivate these lines.

**Figure 3.37** Flip-Flop in Logic Gates

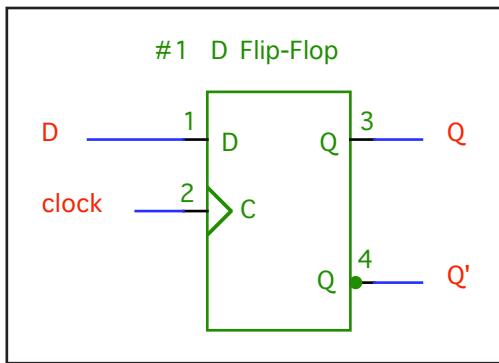


Figure 3.38 D Flip-Flop

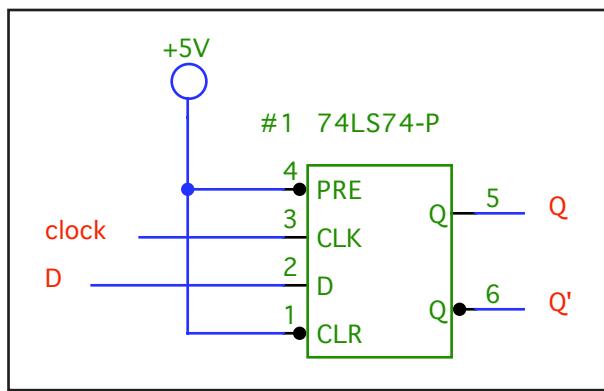


Figure 3.39 74LS74 IC

To use the circuit, perform the following steps:

1. Connect the clock input to LO and the D input to whatever value you want (let's say 1, in this case).
2. Now we want to pulse the clock. Disconnect the clock from LO, touch it to HI, then move it back to LO. That's a pulse. In a D flip-flop, a pulse is detected at the moment of the change from LO to HI.
3. You can leave the clock connected to HI for as long as you like and it is still only seen as a single pulse.

What is Data?

Before we look at the more complex chips that are used in the Apple I, we need to understand what all this data is that we're passing around.

Counting in Binary and Hexadecimal

Digital logic only has two states, high or low, so our numbers need to be expressed in *binary* (base-2) format. A binary digit, like a digital signal, is always either 1 or 0. Converting from decimal to binary is tedious, but you can calculate the values quickly by adding up each place value.

Consider that in decimal we have place values 1, 10, 100, and so on, such that we could calculate the number 203 by multiplying the digit by the place value:

$$(2 \times 100) + (0 \times 10) + (3 \times 1) = 203$$

In binary we can do the same thing, except that our place values are now 1, 2, 4, 8, and so on (see Table 3.13). If we take 1101, we can calculate:

$$(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 13$$

Decimal	Binary	Hexadecimal
1,000,000	64	16,777,216
100,000	32	1,048,576
10,000	16	65,536
1,000	8	4,096
100	4	256
10	2	16
1	1	1

Table 3.13 Values in Decimal, Binary and Hexadecimal

Distinguishing Numbers

If we write "10," is it a decimal "10" or a binary "two?" To distinguish binary numbers from decimal numbers, append a 'b' to the end so that it reads "10b." Hexadecimal numbers are prefixed with a '\$' or sometimes with "0x;" hence, a hexadecimal "10" would look like "\$10" or "0x10."

Each digit in binary is called a *bit*. With 10 bits, we can express 0 through 1023 in binary. Note that you also have 10 fingers. Counting on your fingers, with each finger corresponding to a bit, is a good way to get used to the binary system (Figure 3.40). As you count, you'll also notice some patterns, which are also apparent in Table 3.14. The ones column alternates 0 and 1 with each row; the tens column alternates every two rows, and so on.

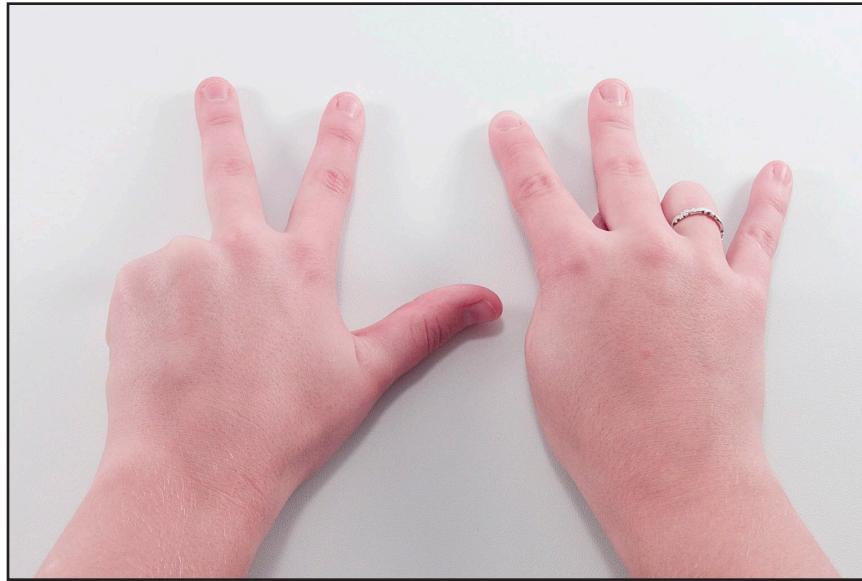


Figure 3.40 125 in Binary

Binary numbers are difficult to read and write, not to mention pronounce. Fortunately, they're very easy to translate into the more convenient *hexadecimal* format (base-16). Note that the binary numbers in the table are divided into groups of four. With four bits, we can count up to 15, which correlates perfectly with hexadecimal. Since we only have 10 symbols (0-9) in decimal, we need to create six more for hexadecimal. For these last six symbols, the letters A through F are used. For example, 'B' is 11, 'C' is 12, and 'F' is 15. Let's take a look at a few hexadecimal examples using place-value calculations.

$$\$CF = (12 \times 16) + (15 \times 1) = 207$$

$$\$10 = (1 \times 16) + (0 \times 1) = 16$$

$$\$FFFF = (15 \times 4096) + (15 \times 256) + (15 \times 16) + (15 \times 1) = 65535$$

Decimal	Binary	Exponential	Hexadecimal
0	0000 0000b	0	\$00
1	0000 0001b	2^0	\$01
2	0000 0010b	2^1	\$02
3	0000 0011b	$2^1 + 2^0$	\$03
4	0000 0100b	2^2	\$04
5	0000 0101b	$2^2 + 2^0$	\$05
6	0000 0110b	$2^2 + 2^1$	\$06
7	0000 0111b	$2^2 + 2^1 + 2^0$	\$07
8	0000 1000b	2^3	\$08
9	0000 1001b	$2^3 + 2^0$	\$09
10	0000 1010b	$2^3 + 2^1$	\$0A
11	0000 1011b	$2^3 + 2^1 + 2^0$	\$0B
12	0000 1100b	$2^3 + 2^2$	\$0C
13	0000 1101b	$2^3 + 2^2 + 2^0$	\$0D
14	0000 1110b	$2^3 + 2^2 + 2^1$	\$0E
15	0000 1111b	$2^3 + 2^2 + 2^1 + 2^0$	\$0F
16	0001 0000b	2^4	\$10
17	0001 0001b	$2^4 + 2^0$	\$11
18	0001 0010b	$2^4 + 2^1$	\$12
19	0001 0011b	$2^4 + 2^1 + 2^0$	\$13
20	0001 0100b	$2^4 + 2^2$	\$14
32	0010 0000b		\$20
64	0100 0000b		\$40
128	1000 0000b		\$80
255	1111 1111b		\$FF
256	0000 0001 0000 0000b		\$01 00
1000	0000 0011 1110 1000b		\$03 E8
1023	0000 0011 1111 1111b		\$03 FF
1024	0000 0100 0000 0000b		\$04 00
14287	0011 0111 1100 1111b		\$37 CF
65535	1111 1111 1111 1111b		\$FF FF

Table 3.14 Value Patterns in Decimal, Binary, Exponential and Hexadecimal Notation

Bytes

There's not much that can be expressed with a single bit, so computers examine a collection of bits at one time. This collection of bits is called a *word*. Modern computers have 32 or 64-bit words. Traditionally, personal computers such as the Apple I have used an 8-bit word, which is why you'll often hear computers from the 1970s and early '80s referred to as "8-bit microcomputers." This 8-bit word is known as a *byte*.

Eight bits can be used to represent any letter in the alphabet, a computer instruction, a number, the color of a pixel, or data in countless other formats. A byte is not significantly harder to work with than a single bit. An 8-bit flip-flop (called a *register*), for example, is just eight flip-flops connected to the same clock (Figure 3.41).

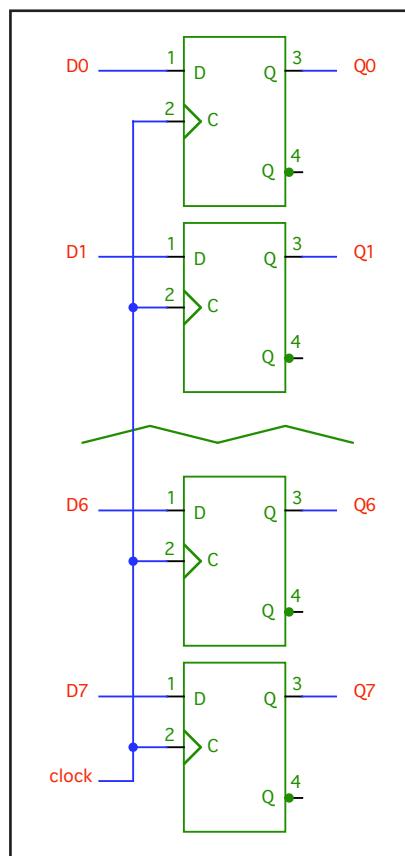


Figure 3.41 Flip-Flops in Parallel

When we work with eight bits, it can become very repetitive drawing identical objects one after the other, eight times. A shortcut is therefore adopted where we use a single (usually thicker) line to represent multiple lines. The circuit in Figure 3.42 is equivalent to that in Figure 3.41, but it takes less space and is quicker to read. If you're using schematic software such as McCAD, it may forgo the slash and merely use a thicker line.

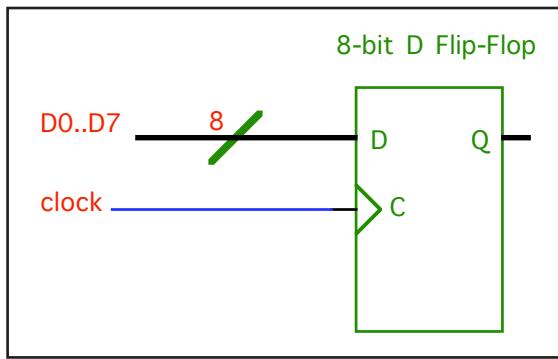


Figure 3.42 8-Bit D Flip-Flop

A selection of lines (such as D0 through D7) treated as a group is called a *bus*. Since our Apple I circuit is based on bytes, we will be using an 8-bit bus quite extensively. A bus can be either parallel or serial. If you're familiar with the parallel and serial ports used on PCs and the Apple II, you probably already have an idea of the difference between the two. In a parallel bus, which we've seen in Figures 3.41 and 3.42, every bit is on its own line and arrives at the same time. Every clock pulse, a new byte of data arrives. This bus is the simplest to use and to understand.

The other option, a serial bus, uses a single data line. All data arrives over this single line, one bit after the next. Each time the clock pulses, the next bit arrives and must be saved by the receiving device. Once the eighth bit arrives, the recipient can examine the entire byte. Upon the next clock pulse, the first bit of the next byte arrives.

Nibbles and Bits

A nibble is four bits—half a byte (get it?). The term is not often used, but is mentioned here for completeness.

ASCII and the Alphabet

A byte has 256 possible combinations, more than enough to represent the entire alphabet. The American National Standards Institute (ANSI) developed a standardized code in the 1960s to facilitate the exchange of information between different computers. Called the American Standard Code for Information Interchange (ASCII), it remains in common use today. ASCII uses seven bits to produce the 128 characters enumerated in Table 3.15.

Dec	Hex	Char
0	0	NUL (null)
1	1	SOH (start of heading)
2	2	STX (start of text)
3	3	ETX (end of text)
4	4	EOT (end of transmission)
5	5	ENQ (enquiry)
6	6	ACK (acknowledge)
7	7	BEL (bell)
8	8	BS (backspace)
9	9	TAB (horizontal tab)
10	A	LF (line feed)
11	B	VT (vertical tab)
12	C	FF (form feed)
13	D	CR (carriage return)
14	E	SO (shift out)
15	F	SI (shift in)
16	10	DLE (data link escape)
17	11	DC1 (device control 1)
18	12	DC2 (device control 2)
19	13	DC3 (device control 3)
20	14	DC4 (device control 4)
21	15	NAK (neg. acknowledge)
22	16	SYN (synchronous idle)
23	17	ETB (end of trans.)
24	18	CAN (cancel)
25	19	EM (end of medium)
26	1A	SUB (substitute)
27	1B	ESC (escape)
28	1C	FS (file separator)
29	1D	GS (group separator)
30	1E	RS (record separator)
31	1F	US (unit separator)

Dec	Hex	Char
32	20	Space
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
39	27	'
40	28	(
41	29)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Table 3.15 ASCII Chart

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
92	5D]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	
125	7D	}
126	7E	~
127	7F	DEL

Table 3.15 Continued ASCII Chart

A Few More Chips

Shift Register

A *shift register* is a collection of flip-flops hooked up in a row such that the output of one is the input to the next (Figure 3.43). This allows us to store a series of data as it comes in—one bit every clock pulse. Let’s assume that at every pulse of the clock, a new bit of data arrives on the input line, which is, in fact, usually the case. This data will be:

10110000...

Though most shift registers have a “clear” input to reset all the flip-flops to 0, ours does not. Therefore, we’re going to assume that the content of the flip-flops before we enter our data is unknown (either 0 or 1). This will be represented in our table by an “x.” We also use “t” to represent time. For example, t0 is when we begin, t1 is immediately after the first clock pulse, t2 after the second pulse, and so on.

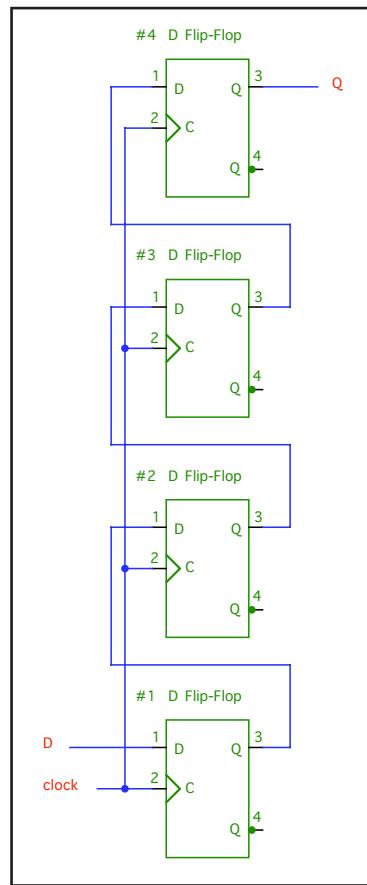


Figure 3.43 Shift Register With D Flip-Flops

At t_0 we don't know the contents of any of the flip-flops, but some time before the first clock pulse, a 1 arrives on the D input line. At t_1 , the clock pulses, and this 1 is loaded into the first flop-flop. As soon as it is loaded into that flip-flop, it becomes present on the Q output. At t_2 , the clock pulses again. At this instant, 1 is loaded into flip-flop #2 and the next 0 is loaded into flip-flop #1. You can see the full continuation of this process in Table 3.16.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
FF #1	x	1	0	1	1	0	0	0	0	0
FF #2	x		1	0	1	1	0	0	0	0
FF #3	x	x	x	1	0	1	1	0	0	0
FF #4	x	x	x	x	1	0	1	1	0	0

Table 3.16 Shift Register Output

Buffer and Tri-State Buffer

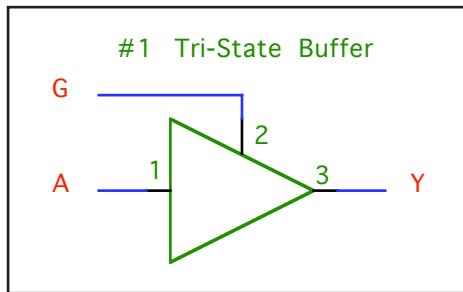
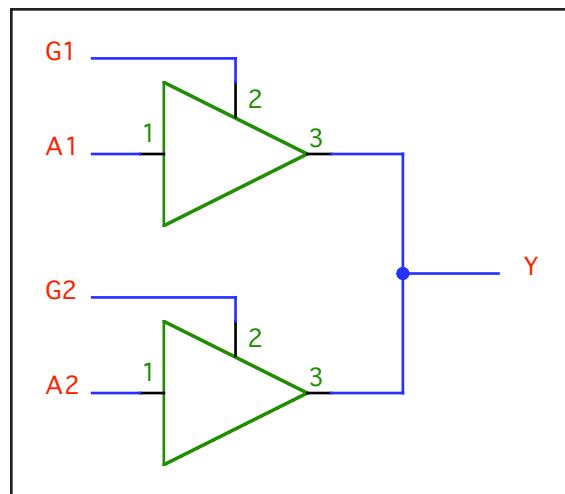
There are occasions when we want to use the output of one gate as the input to quite a few other gates, but we're limited by the fact that a TTL gate can only drive 10 other gates. To alleviate this problem, we have *buffers*. As you can see in Table 3.17, a buffer is like an inverter that doesn't invert—it just takes the signal and replicates it. Each buffer we have on an output can drive 10 more gates.

Input A	Output Y
0	0
1	1

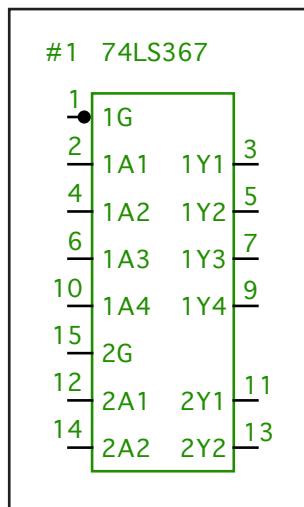
Table 3.17 Buffer Truth Table

A *tri-state buffer* is even more interesting. When we discussed OR gates, we looked at how a gate's output is either low or high, 0 or 1. Tri-state buffers add a third state—off. In this third state, “floating,” no output at all comes from the chip and it will not interfere with other signals on the line.

Examine Figure 3.44 and Table 3.18. You can think of line G as a valve that turns on the flow of electricity. As long as G is low (i.e. the flow is turned off), you can put any signal you want on the output lines without harming the chip. This allows us to create circuits such as the one in Figure 3.45. Beware, though—if both gates' outputs are turned on at the same time, you'll damage your chips. If you'd like to try wiring some of these up yourself, you can use the 74LS367 (Figure 3.46).

**Figure 3.44** Tri-State Buffer**Figure 3.45** Circuit with Tri-State Buffers

Input A	Input G	Output Y
0	0	X
0	1	0
1	0	X
1	1	1

Table 3.18 Tri-State Buffer Truth Table**Figure 3.46** 74LC367 IC

Encoders and Decoders

We have eight inputs. Line six is high. Let's say we want to send the number 6 to our computer, or to some output device such as a numeric display. The receiving device is going to expect this data in binary format. If you've paid attention to the patterns in binary digits, a *binary encoder* is not very difficult to make. To express numbers between 0 and 7 in binary, we need three bits. Examine the patterns in Table 3.14 and you'll see that the lowest bit (bit 0) alternates between 0 and 1, such that bit 0 is high for 1, 3, 5, and 7. The second lowest bit (bit 1) alternates every two lines between 0 and 1, so it's high for 2, 3, 6, and 7. Finally, bit 3 alternates every four lines, so it's high for 4, 5, 6, and 7. This circuit is expressed in Figure 3.47. The same circuit is shown in Figure 3.48 as an ordinary encoder.

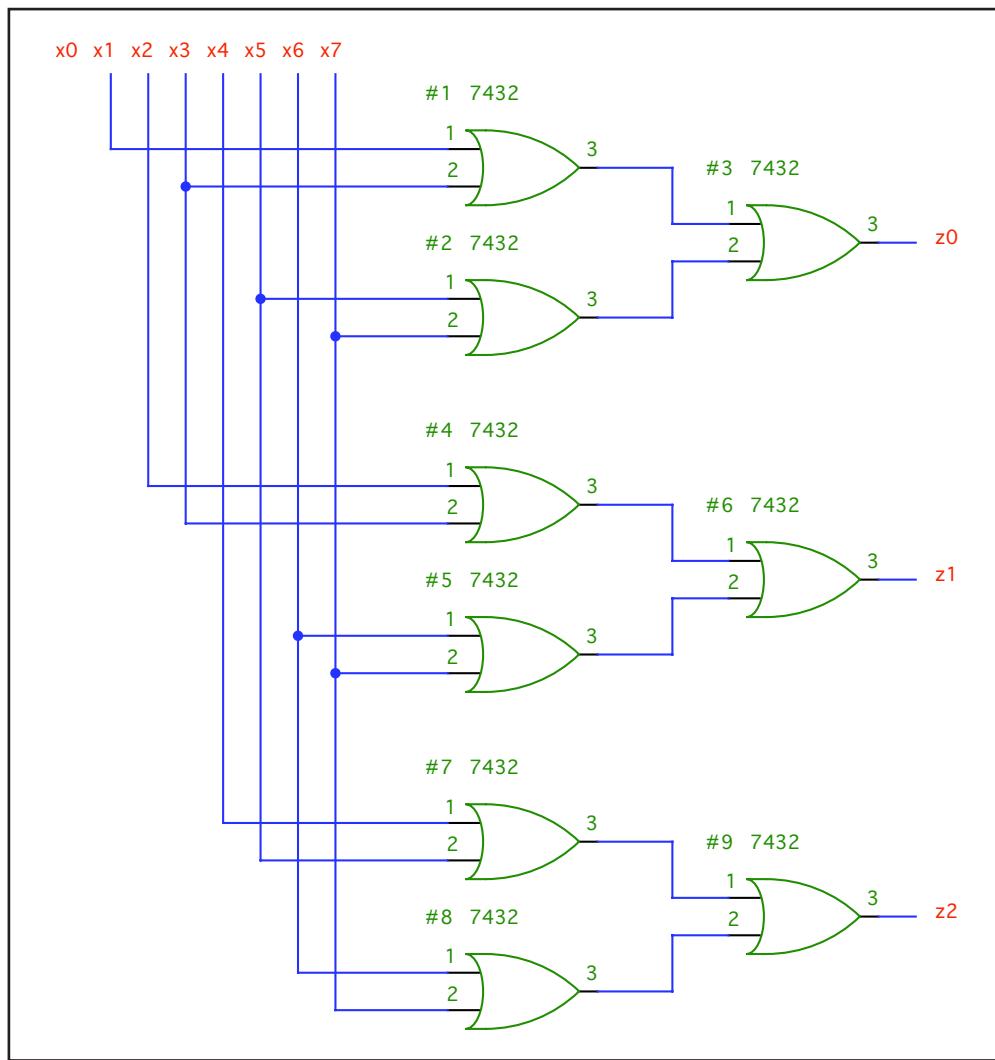
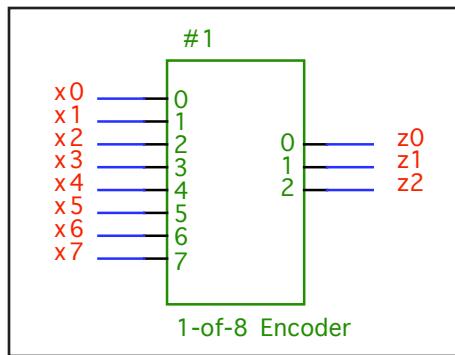
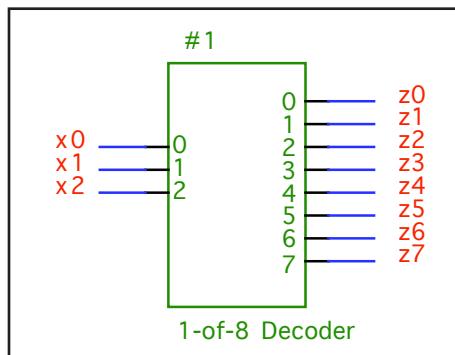


Figure 3.47 Encoder with Gates

**Figure 3.48** 1-of-8 Encoder

Is it possible to go in the reverse from binary to single-line output? Let's design a 1-of-8 *decoder* (Figure 3.49). First, fill out a truth table (Table 3.19). Consider writing this out by hand before looking ahead. Remember that your inputs (x_0, x_1, x_2) represent the binary digits and z_0 through z_7 are your outputs, so when you're filling out the table, fill in all the inputs before solving the outputs, line by line.

**Figure 3.49** 1-of-8 Decoder

x_2	x_1	x_0	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Table 3.19 Decoder Truth Table

Next, write logic statements based on this truth table. For example, z_0 is high only when x_2 is not high, x_1 is not high, and x_0 is not high; hence, we write: $z_0 = x_0' \cdot x_1' \cdot x_2'$. The complete set of logic statements is:

$$z_0 = x_0' \cdot x_1' \cdot x_2'$$

$$z_1 = x_0 \cdot x_1' \cdot x_2'$$

$$z_2 = x_0' \cdot x_1 \cdot x_2'$$

$$z_3 = x_0 \cdot x_1 \cdot x_2'$$

$$z_4 = x_0' \cdot x_1' \cdot x_2$$

$$z_5 = x_0 \cdot x_1' \cdot x_2$$

$$z_6 = x_0' \cdot x_1 \cdot x_2$$

$$z_7 = x_0 \cdot x_1 \cdot x_2$$

Finally, we draw a circuit based on these logic gates (Figure 3.50). (You might notice that this circuit uses three more inverters than it needs to. There are two ways to cut the number of inverters in half, by using DeMorgan's Laws or by simply rearranging some wires.)

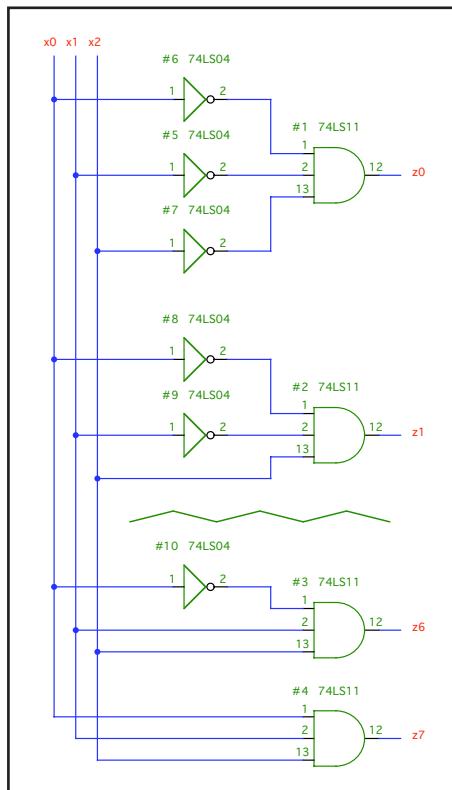


Figure 3.50 Decoder with Gates

Summary

We've covered a lot of ground in this chapter. Each layer—digital logic, microcomputer, and software—can stand on its own, so there's no need to be too concerned if you didn't grasp everything. It doesn't mean you won't be able to understand the upcoming chapters. If you'd like to learn even more about digital logic, I would encourage you to look at the following list of resources:

Books:

- *The Art of Electronics* by Horowitz and Hill (Cambridge University Press, 1989)
- *Hands-On Electronics* by Kaplan and White (Cambridge University Press, 2003)

On the Internet:

- CAL for Digital Logic: <http://www.ee.surrey.ac.uk/Projects/Labview/>
- Digital Logic: <http://www.play-hookey.com/digital/>
- Discover Circuits: <http://www.discovercircuits.com/D/digital.htm>
- How Electronic Gates Work: <http://electronics.howstuffworks.com/digital-electronics.htm>
- TTL Data Book: <http://upgrade.cntc.ac.kr/data/ttl/>

Chapter 4

Building the Replica I

In this Chapter

- Learning to Solder
- Assembling the Replica I
- Serial I/O Board
- Using McCAD EDS SE500

Introduction

Before learning microcomputer design, it helps to know how to program, and before learning how to program, it's best to have a computer with which you can program. This chapter will discuss building the Replica I with a minimum of theory. First an introduction to soldering will be provided. This will be followed by step-by-step instructions for assembling the Replica I kit sold by Vince Briel and a discussion of the Serial I/O board. Finally, the McCAD EDS package will be introduced for readers interested in having their own boards fabricated.

Learning to Solder

Soldering is a skill that takes time to learn. If you've never soldered before, I would suggest acquiring some practice time before attempting to assemble the Replica I. Test your skills on old circuit boards by removing components and putting them back in, or buy a couple of cheap parts from Radio Shack on which to practice.

The first time you use a new soldering iron, you need to tin the tip, which is done by heating the iron and applying a thin coating of solder to the tip. The surface and components you are about to solder must be clean. With a new Replica I board, this is not a concern, but if any of your parts are older or appear dirty, clean them with steel wool or alcohol.

Insert the component you wish to solder into the board; in this example we will use a resistor. Identify the correct location for the part and insert it, bending the leads as necessary to make it fit. Flip the board over and bend the resistor's leads slightly outward so that it will stay in place.

Allow your soldering iron to fully heat up before starting. Place the tip of the iron so it touches both the lead and the hole (Figure 4.1). Both must be hot for a reliable connection. Hold the solder so that it touches both the lead and the hole, but not the soldering iron. In a second or two the solder will start to flow. When the hole fills and begins to look like the solder joints you've seen on commercially-made boards, stop applying solder. This will probably take no longer than a second.

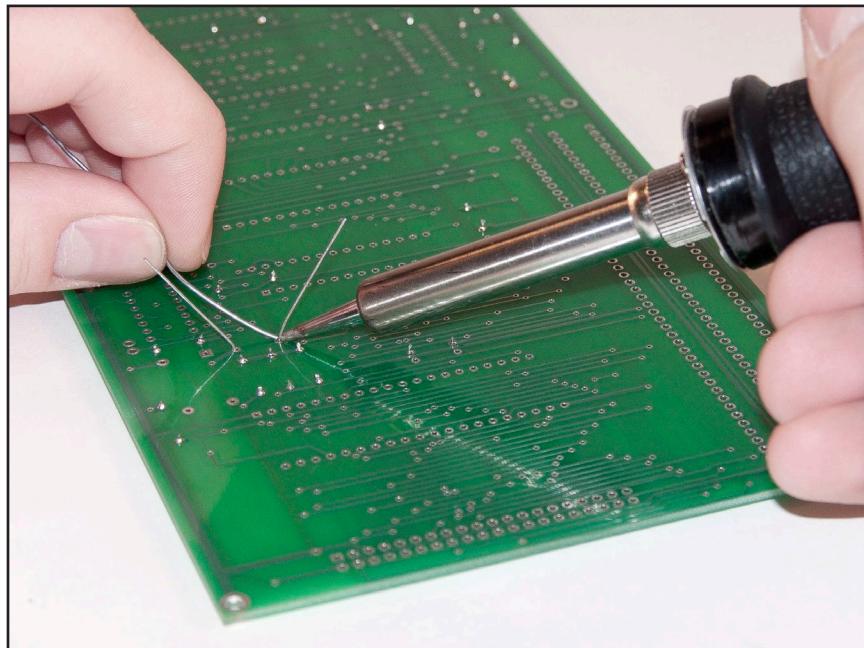


Figure 4.1 Applying Solder

Allow a few seconds for the solder to cool. Then, cut off the excess lead (Figure 4.2). Be careful not to jostle the joint before it is cool or you risk damaging the connection.

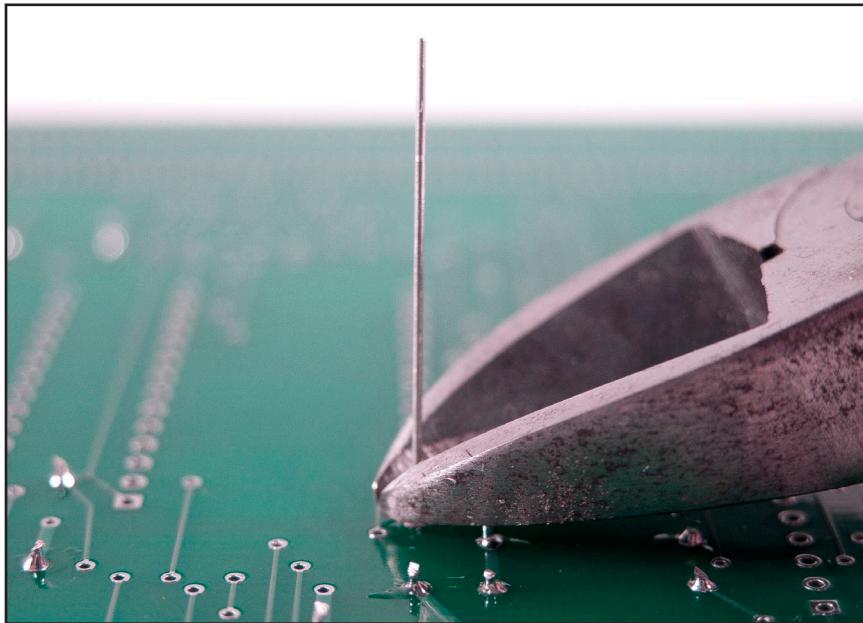


Figure 4.2 Trimming the Leads

Figure 4.3 shows some finished solder joints. Your joints should be shaped like these with a shiny and smooth finish. If your joint is dull and dirty, you have a cold solder joint – possibly because you bumped the joint while the solder was still cooling or because the joint is not getting hot enough. Though the connection may look solid, it is not electrically reliable and should be resoldered.

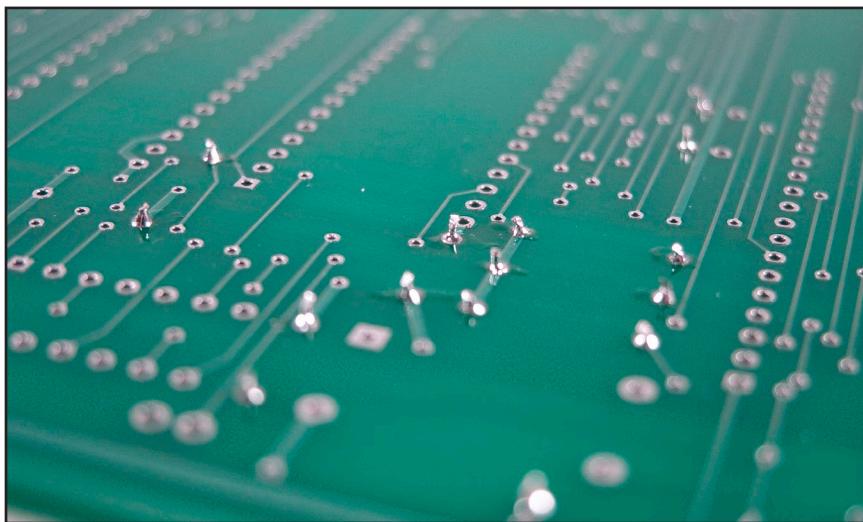


Figure 4.3 The Finished Joints

Assembling the Replica I

Figure 4.4 shows the unassembled Replica I kit available from Briel Computers. Included with the kit is all the parts you'll need to build a working Replica I. If you're a more advanced user, Briel also sells components individually, which allows you to mix and match parts or install them on your own printed circuit board, described later in this chapter. Users uncomfortable with soldering can order pre-assembled kits.

Assembling the Replica I TE

The instructions in this section describe building Briel Computers' original Replica I. This original kit has since been replaced by the Replica I TE. If you have the new TE model, you might still find this chapter useful for soldering and assembly tips, but don't use it to determine the placement of specific components. Of particular note is that the Replica I TE has an onboard serial interface, so there is no longer any need for the Serial I/O Board.



Figure 4.4 The Unassembled Briel Replica I Kit

Parts List

These are the parts included with the Replica I kit, other than the bare printed circuit board.

ICs:

- 6502 CPU 40-pin IC
- 6821 PIA 40-pin IC
- 28C64 EEPROM 28-pin IC
- 62256 SRAM 28-pin IC
- 74LS00 TTL 14-pin IC
- 74LS04 TTL 14-pin IC
- 74HC74 TTL 14-pin IC
- 74LS138 TTL 16-pin IC
- 74HC166 TTL 16-pin IC
- ATMEGA8 28-pin IC
- ATMEGA8515 40-pin IC

Crystals:

- 14.31818 MHz Crystal
- 8.0 MHz Crystal
- 1 MHz TTL Clock

Sockets:

- Qty 3 – 40-pin
- Qty 2 – 28-pin (wide)
- Qty 1 – 28-pin (narrow)
- Qty 2 – 16-pin
- Qty 1 – 16-pin (spring socket, for ASCII keyboard)
- Qty 3 – 14-pin

Connectors:

- Qty 1 – 40-pin Expansion Header
- Qty 1 – DIN PS/2 Keyboard Connector
- Qty 2 – Power Supply Connectors
- Qty 1 – Video Connector

Resistors:

- R1: 1.5 K Ohm (brown-green-red-gold)
- R2: 470 Ohm (yellow-magenta-brown-gold)
- R3: 100 Ohm (brown-black-brown-gold)
- R4, R5: 4.7 K Ohm (yellow-magenta-red-gold)
- R6-R10: 3.3 K Ohm (orange-orange-red-gold)

Capacitors:

- C1, C2, C4, C4: 18 pF
- C3: 0.1 μ F
- C6–C17: 0.1 μ F Bypass
- C18: 0.01 μ F

Diodes:

- D1, D2: 1N4148

Other:

- Qty 2 – Button
- Qty 1 – Jumper Pad

Resistors

It's easiest to install smaller components first. This way, when you turn the board over, the component will be held in place between the table and the board. Let's install the resistors first. There are 10 resistors on the Replica I board labeled R1 through R10. These labels are silk-screened onto the Replica I's printed circuit board, making it easy to determine where parts should go. In the case of resistors, the resistor value is shown beneath the label (Figure 4.5). You can check these values using the table of resistor codes in Chapter 3.

Polarity does not matter for resistors—you can install them in either direction. For the sake of aesthetics, I like to place all of mine in the same direction. With all resistors installed, your board should look like that in Figure 4.6. If you have doubts about the quality of your soldering, use a multimeter and measure across the resistors, solder joint to solder joint.

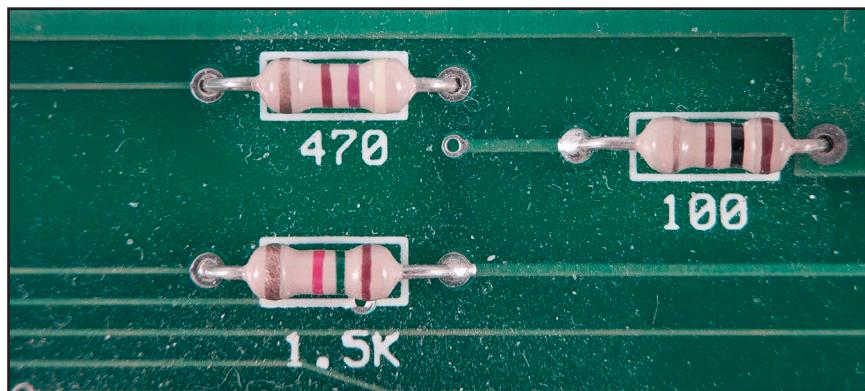


Figure 4.5 Resistors

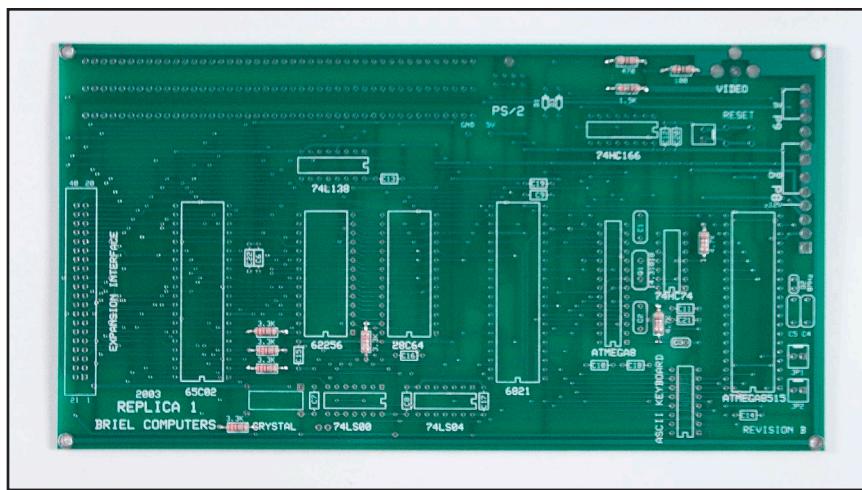


Figure 4.6 Installed Resistors

Diodes and Bypass Capacitors

Next, install the two diodes at D1 and D2. For these, polarity does matter. The negative (cathode) end of the diode is indicated by a black ring, and the negative end on the board is indicated by a square pin. Match these up when installing the diodes. You can also refer to the silk-screened image between the pins, which illustrates where the ring should be. Figure 4.7 shows the two diodes properly installed.

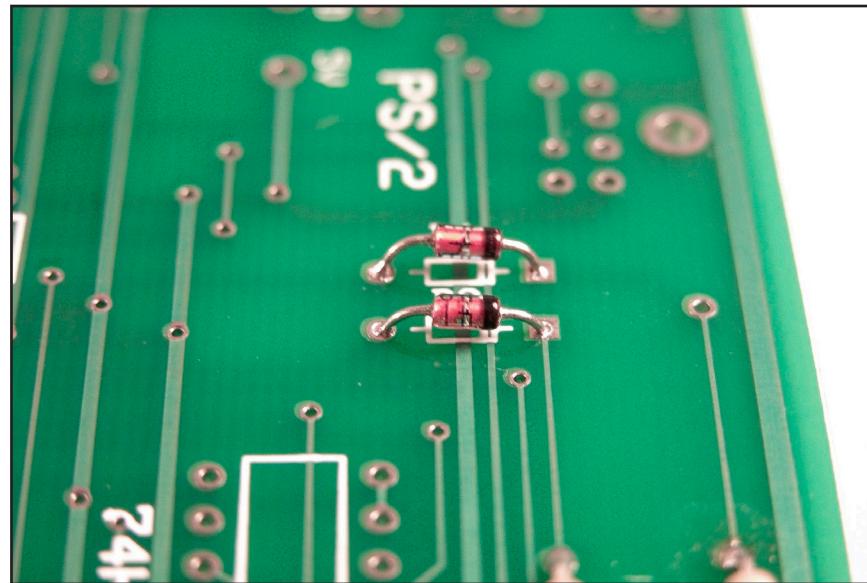


Figure 4.7 Installed Diodes

The 11 bypass capacitors (Figure 4.8) are as easy to install as the resistors. Polarity is unimportant. Bend the leads, solder them into place, and snip the extra length off.

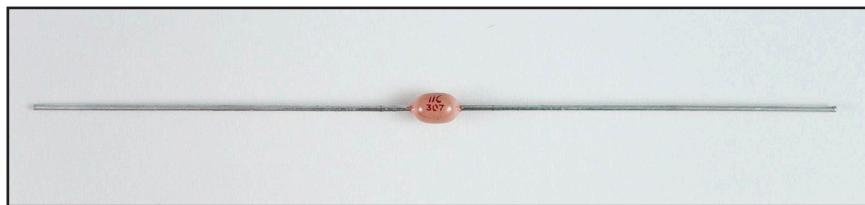


Figure 4.8 0.1µF Bypass Capacitor at C6–C17

Buttons

There are two buttons on the Replica I motherboard, one each for Reset and Clear (Figure 4.9). Polarity on these does not matter. Near each button you'll see a pair of holes for connecting an external button. If you're installing your Replica I in a case, you'll probably want to do this. The wires from the external button can be soldered directly to the board, or a socket can be used. Figure out what sort of external button you'll be using before doing anything with these.

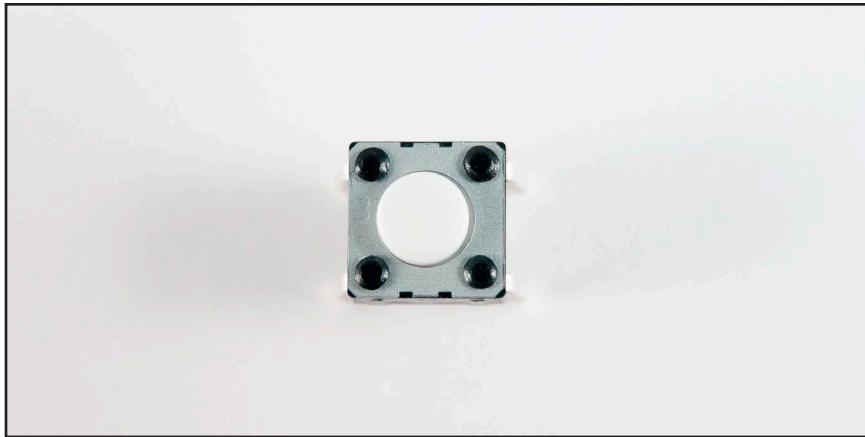


Figure 4.9 Button

Sockets

Sockets should be installed such that the notch shown in the silkscreen matches the notch in the socket (Figure 4.10). Pin 1 on the circuit board is also indicated by a square pin. If you have trouble holding the socket in place while you turn the board over to solder, try temporarily taping it down.

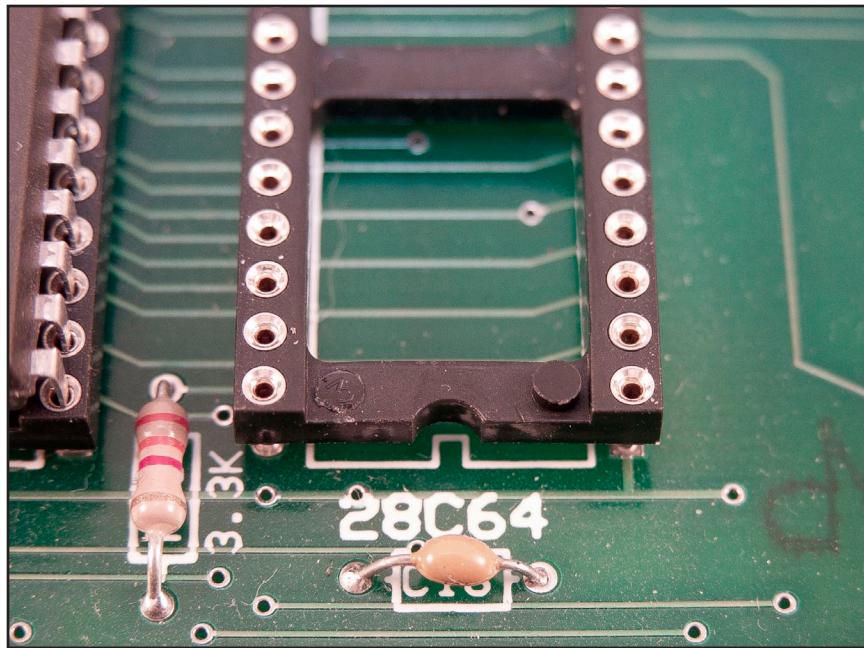


Figure 4.10 Socket Orientation

ASCII Keyboard Socket

The ASCII keyboard connects via a spring socket (Figure 4.11), which looks slightly different than the machined sockets used for the chips. A spring socket is used here because it's easy to connect and disconnect the keyboard connector from it. Machined sockets are generally of higher quality, which is why they're used for the rest of the board. If you're going to be using a PS/2 keyboard exclusively, you can skip installing this part.

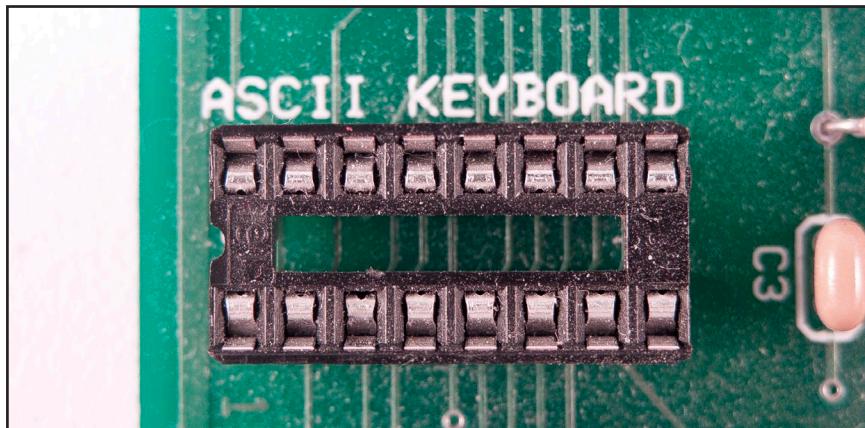


Figure 4.11 ASCII Keyboard Socket

Capacitors

The capacitors are not polarized. Installing them is trivial once you've learned to distinguish one type from another. Figure 4.12 shows the $0.1\mu\text{F}$ capacitor used in C3. The 18pF capacitor in Figure 4.13 is for locations C1 through C5. Figure 4.14 shows a $0.01\mu\text{F}$ capacitor which belongs at C18.

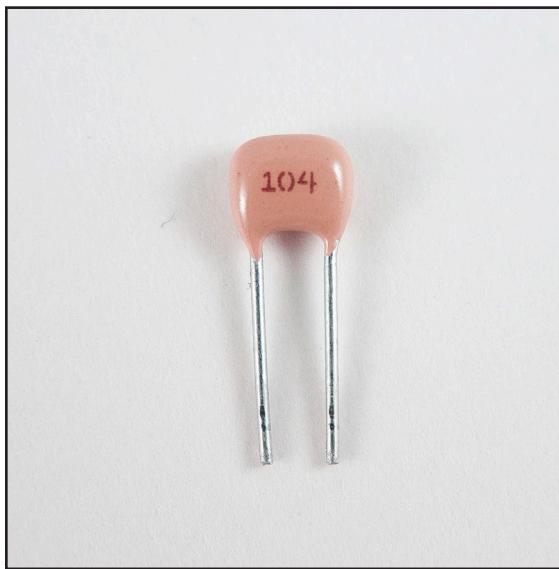


Figure 4.12 $0.1\mu\text{F}$ Capacitor for C3



Figure 4.13 18pF Capacitor for C1–C5

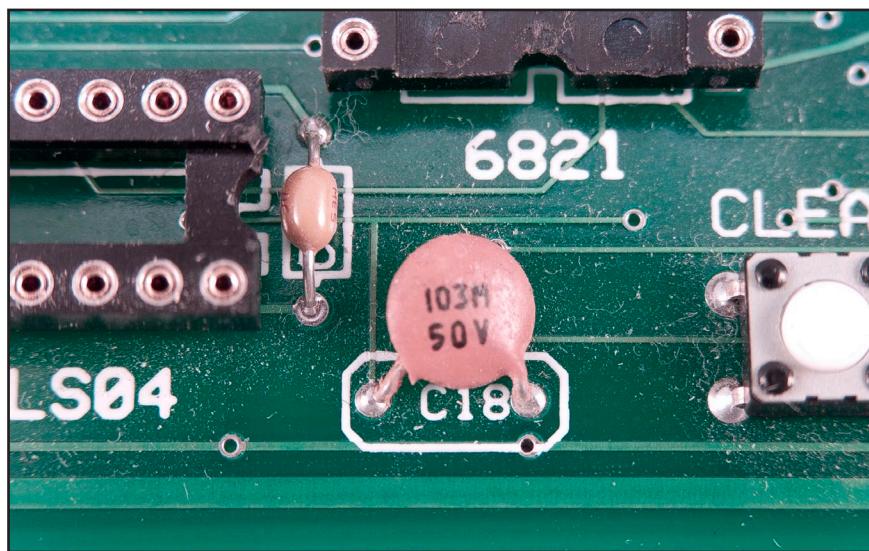


Figure 4.14 $0.01\mu\text{F}$ Capacitor for C18

Header and Jumper

The 40-pin expansion header should be installed so that the single notch faces outward, away from the board, and the double notches face inward, towards the board. The double notches are towards the top in Figure 4.15.

The jumper pad (Figure 4.16) has no polarity and should be installed at location JP1. The shorter end of the pins should be soldered to the board; the longer end should be facing up, away from the board. The jumper pad is used for switching between PS/2 and ASCII keyboard modes. If you're using an ASCII keyboard, a shorting jumper should be placed across the pins. Leave the pad unshorted in order to use a PS/2 keyboard.

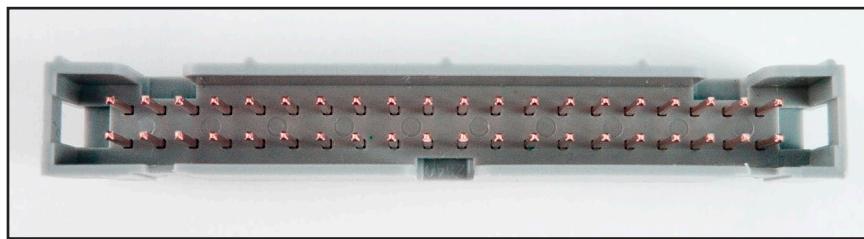


Figure 4.15 40-Pin Expansion Header

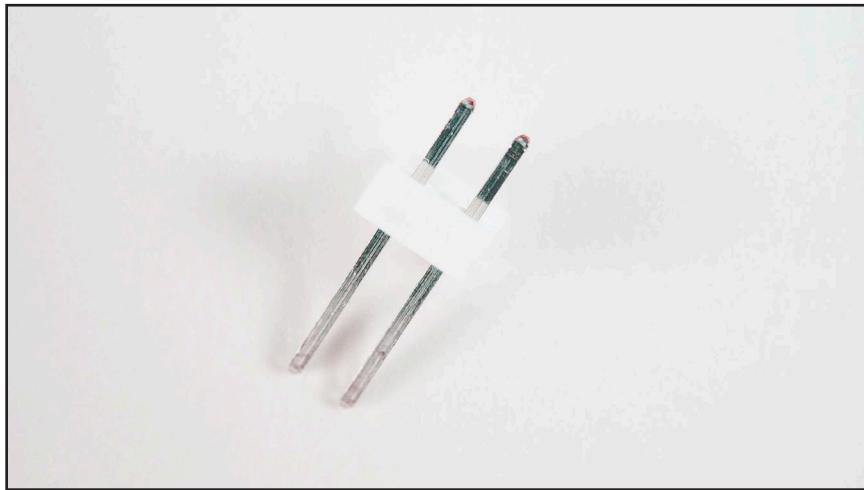


Figure 4.16 Jumper Pad

Crystals

The crystals are easy to damage and should be installed with care. The polarity of the 1MHz TTL clock (Figure 4.17) is critical. Pin 1 on the clock is indicated by a black dot and a squared corner (the rest of the corners are rounded off). On the circuit board, pin 1

is indicated by a square pin while the rest of the pins are round. The 1MHz clock should be installed at the location labeled OSC, for *oscillator*.

For the other two crystals, polarity is unimportant. The 14MHz crystal (Figure 4.18) should be installed next to the ATmega8 IC at the location labeled Q1. The 8MHz crystal (Figure 4.19) should be installed beside the ATmega515 IC at location Q2.



Figure 4.17 1MHz TTL Clock

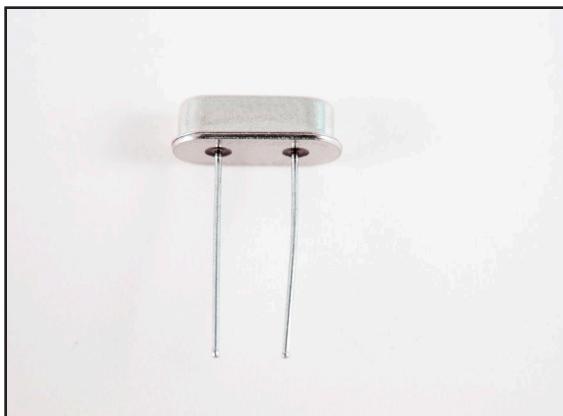


Figure 4.18 14MHz Crystal

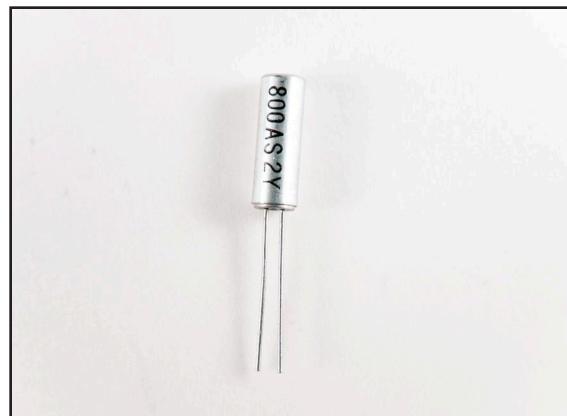
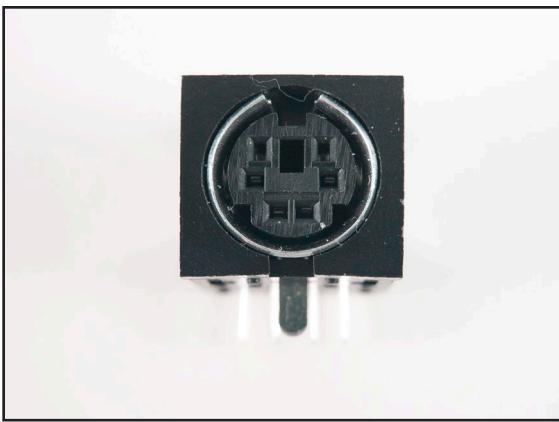
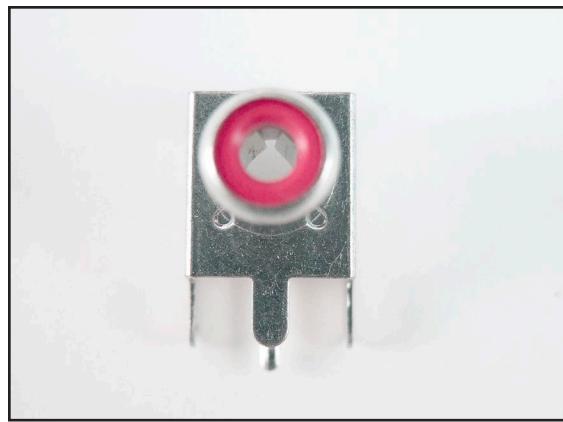


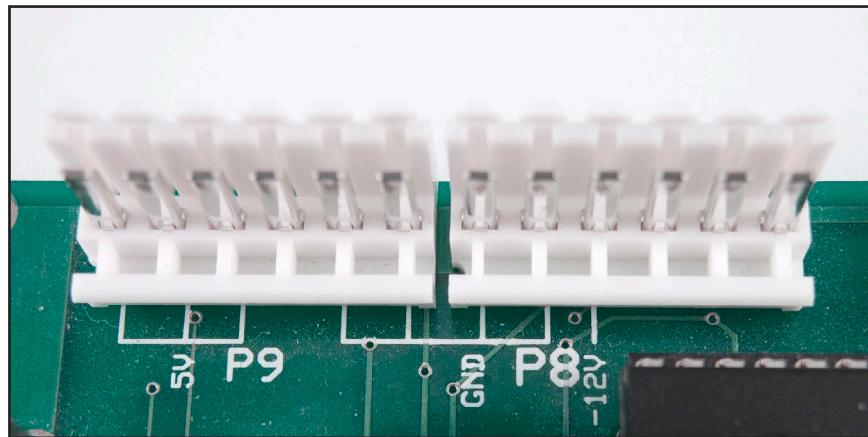
Figure 4.19 8MHz Crystal

Connectors

Almost done! The PS/2 connector (Figure 4.20) should be installed at the PS/2 label. Likewise, the video connector (Figure 4.21) should be installed at the video label. Both of these fit in only one way. The holes for the video connector are large. Be sure the entire hole is filled with solder for a secure electrical connection.

**Figure 4.20** PS/2 Connector**Figure 4.21** Video Connector

Finally, install the power connectors. The two connectors are identical and thus interchangeable, but remember that orientation is important. The flat white support edge should be against the edge of the circuit board with the pins on the inside, as shown in Figure 4.22. When soldering these into place, I recommend using tape. The holes for the power connector's pins are large and it's easy for the connectors to slide around. Your finished board will look much nicer if the two connectors are neatly lined up.

**Figure 4.22** Power Connectors

Finishing Assembly

Congratulations! You've completed all of the soldering. Now, all that remains is to install the chips. Each chip and each socket is labeled; it should be evident where the chips belong. Make sure you get the orientation correct—the notch on the chip should match the notch on the socket. Gently press the chips into their sockets and ensure that

no pins are bent out of place.

When you've completed the preceding steps, connect the power cables. The black wires should be on the inside to match the silk-screened label of GND (Figure 4.23). Connect your monitor and keyboard. Make sure the keyboard select jumper (JP1) is set correctly for your keyboard. Step back and admire your masterpiece (Figure 4.24). Turn on the power and press Reset. A \ prompt should appear on the screen, which means you're ready to move on to start programming.

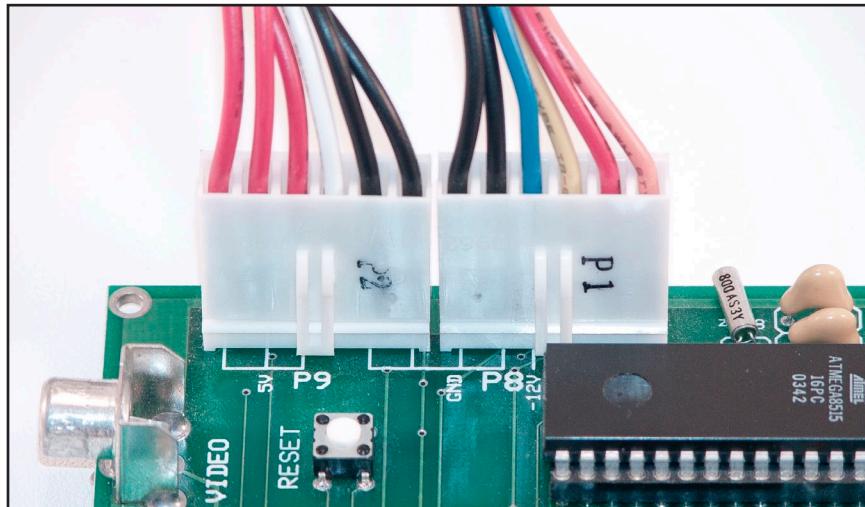


Figure 4.23 Connected Power



Figure 4.24 Completed Replica I System

Serial I/O Board

The Serial I/O board by Briel Computers (Figure 4.25) provides an easy way to interface the Apple I to a modern computer. The I/O board allows emulation of the keyboard and mouse using a terminal program on your Mac or PC. The board plugs into the 6821 socket and provides a standard nine-pin serial interface. Sending data to this serial port is the same as typing it on the keyboard. All data that is sent to the video section is also sent out the serial interface.

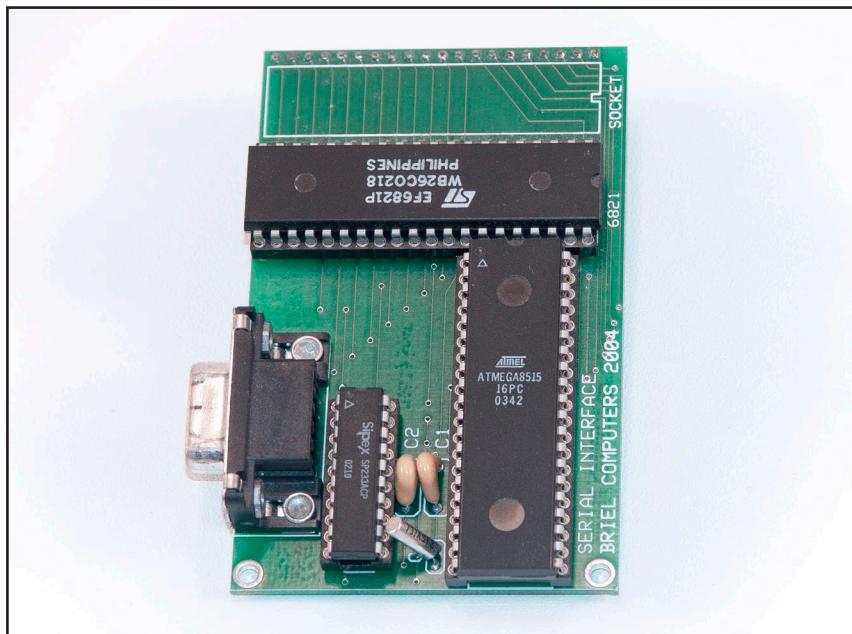


Figure 4.25 Serial I/O Board

To connect the I/O board, first disconnect all power. Next, use a small screwdriver to gently pry the 6821 out of its socket (Figure 4.26). Alternate between gently prying each side so as not to bend any pins. Do not use an IC puller to remove the chip, as this will put stress on the socket connections. Next, gently insert the Serial I/O board into the socket. In the Replica I, the nine-pin serial port should face the same direction as the video and PS/2 ports. The installed board is shown in Figure 4.27.

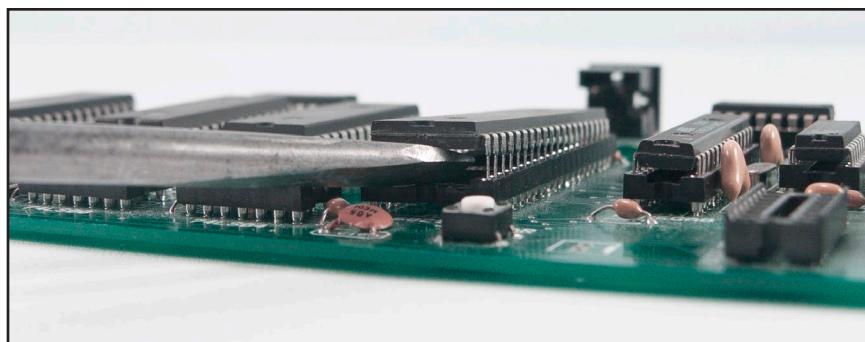


Figure 4.26 Removal of the 6821

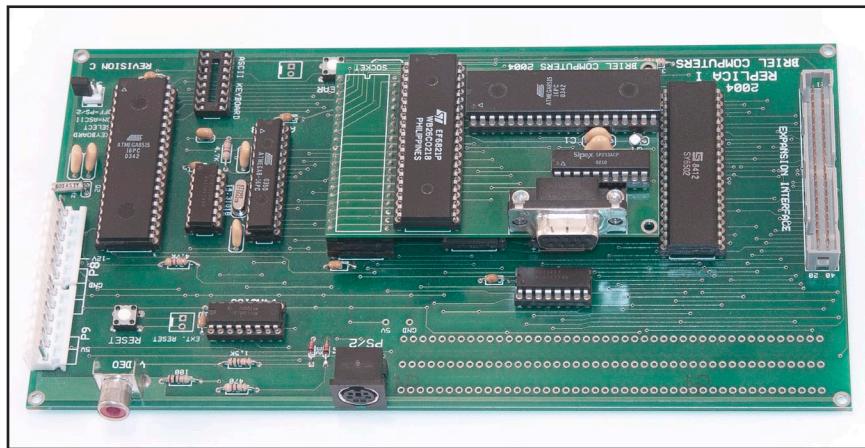


Figure 4.27 Serial I/O Board Installed

If you ever need to remove the serial board, be very careful. Its pins are brittle and will easily snap off if bent too far. It is very easy to end up with a pin stuck in your socket that will be extremely difficult to remove.

If you have an older PC or Mac, the Serial I/O board can be connected directly to your computer's serial port via a straight serial cable (not null-modem cable). Newer computers will require a USB-to-serial adapter. One such adapter is the Keyspan USB Serial Adapter (#USA-19). The communication needs of the Apple I are very basic; consequently, any USB adapter that isn't special-purpose should work.

Windows users can use HyperTerminal, which is included with the operating system. Macintosh users will want to check out ZTerm by Dave Alverson. When launching ZTerm, hold down the Shift key. This will produce a menu where you can choose which serial port to use. If you're using the internal serial port on an older machine, you'll want to select "printer" or "modem" port. With a USB adapter, the name will be something obscure such as "USA19191P1.1."

Once the program is launched, choose Connection from the Settings menu and confirm that all your settings match those in Table 4.1. All other settings can be left at

their defaults. Make sure your replica is running and has been reset. Text you now type in the terminal will be sent to the Apple I and video output will be sent to the terminal. Programs, available from the Apple I Owners Club on Applefritter, can be entered by simply using copy and paste in the terminal or by going to the File menu, selecting Send Text, and choosing the file that contains the program.

Property	Setting
Data Rate	2400 bps
Data Bits	8
Parity	None
Stop Bits	1
Echo	Off
Xon/Xoff	Off
Hardware Handshake	Off

Table 4.1 Connection Settings

Using NVRAM

Nonvolatile RAM (NVRAM) can be used in lieu of static RAM (SRAM) in your replica. NVRAM preserves the contents of memory, even when the computer is turned off. This means that programs you enter by hand will still be accessible the next time you use your replica. NVRAM is implemented either by placing a battery inside the IC to provide constant power to standard SRAM, or by using an EEPROM. The DS130Y-120 is an equivalent replacement for the 32KB SRAM IC that is used normally in the Replica I. It's available from Digi-Key and others for about \$20.

Using McCAD EDS SE500

McCADC EDS SE500 is an electronic design system that includes all the tools you need to take a design from idea to production. Available from the Mac App Store, it's a package that includes Schematics SE500, PCB-ST SE500, and Gerber Translator. Schematics SE500 is a very easy-to-use program. The schematics it generates can be used with PCB-ST SE500, a PCB layout design environment, to design a printed circuit board based on the schematic. McCADC Gerber Translator will convert the PCB-ST document to a standard Gerber photo-plotting text command file that can be read by PCB manufac-

ers anywhere.

If you are new to circuit design, the circuits shown in this section will probably look foreign to you until you read the following chapters. It will be of interest to readers wishing to modify the Replica I circuit to suit their own needs and interests. Don't be fooled by the simple appearance of the McCAD interface. This is a very capable toolset that is easy to master yet very robust in its capabilities.

Documentation for McCAD EDS in PDF can be found in the application's Help menu. You are encouraged to read through the documentation before attempting a new design or modifying the files provided in the Supplemental Software package. If you run into difficulty, visit the forums on Applefritter at www.applefritter.com/forum.

McCAD Schematics SE500

Figure 4.28 shows the Replica I circuit opened in McCAD Schematics. The circuit can be modified using the tools on the left of the screen. Components selected from the Library menu can be placed and new components can be created using the online library editor. A separate library of Apple I components (6502, 6821, etc.) was created for inclusion with this book.

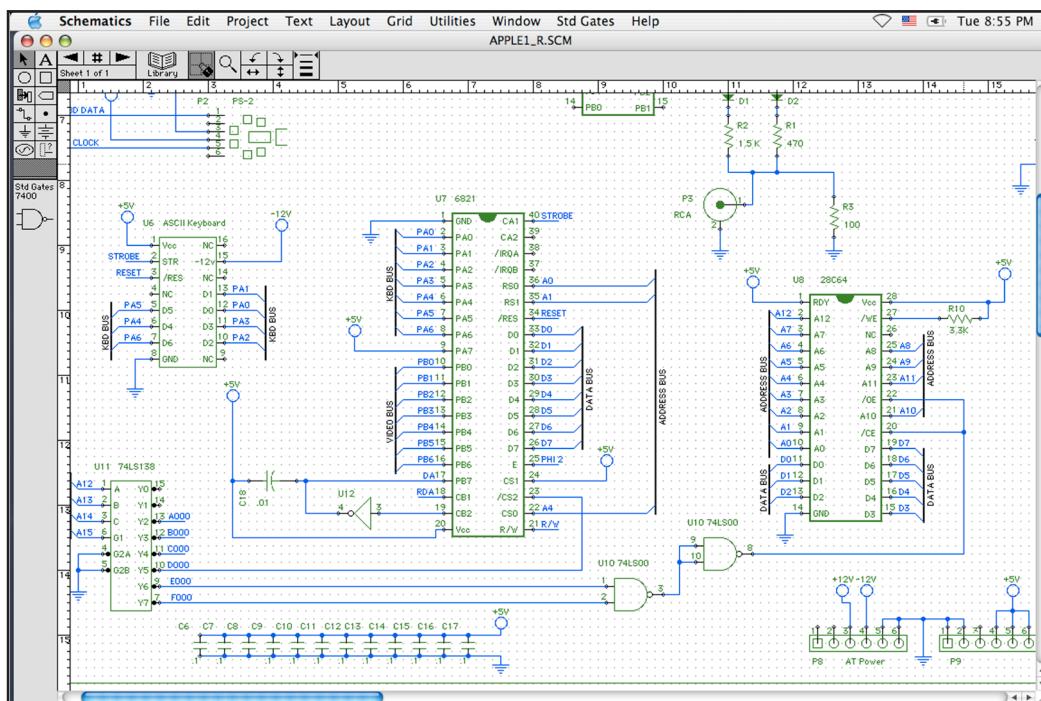


Figure 4.28 McCAD Schematics

McCADC PCB-ST SE500

Figure 4.29 shows the Replica I PCB open in McCADC PCB-ST. When you import a McCADC Schematics file to PCB-ST it will automatically collect the needed parts and allow you to place them and route the interconnecting traces. If you are happy with the Replica I circuit design but perhaps wanted to change the component layout or board shape, you would do so with this software. The standard Gerber file generated by PCB-ST can be sent to any PCB manufacturer to be fabricated.

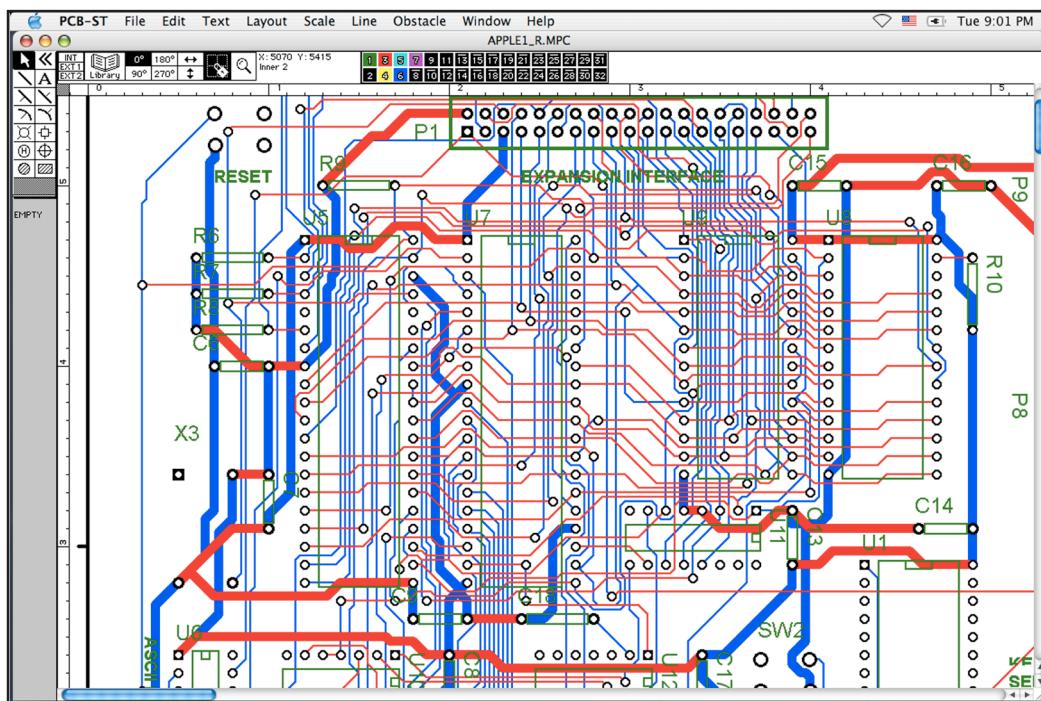


Figure 4.29 McCADC PCB-ST

Chapter 5

Programming in BASIC

In this Chapter

- Setting Up BASIC
- Hello World
- Welcome, User (Input, Variables, and Strings)
- Calculator (Math)
- Count Down (FOR/NEXT)
- Some (IF/THEN)
- The GOSUB Command
- Arrays
- Strings, In Depth
- PEEK and POKE
- The CALL Command
- *Richard III*: Interactive Fiction

Introduction

As its name suggests, Beginner’s All-Purpose Symbolic Instruction Code (BASIC) is a great high-level programming language for beginners. BASIC was developed at Dartmouth College in 1964 and became the dominant programming language on microcomputers in the 1970s and ‘80s. If you visit your local library, you’ll probably find dozens of books devoted to programming in BASIC, whether for the Apple II, Commodore 64, or IBM PC. All these variations of BASIC have the same base syntax, but they differ in more advanced features. Apple I BASIC is among the simplest of the variations. In this chapter, you’ll learn how to set up BASIC on the Apple I and write some simple programs.

Setting Up BASIC

The original Apple I did not have BASIC in ROM. Instead, users had to either load it into RAM using an optional cassette interface, or painstakingly type the entire program in by hand. If you're using a Replica I ROM, you already have BASIC ready to run. Power the system up and press Reset. At the / prompt, type:

E000R

This means "run the code, starting at address \$E000." Press Return and a > prompt will appear. You're now running BASIC.

The Apple I supports only uppercase type. However, if you're using a PS/2 keyboard, it will probably be possible for you to input lowercase text. The Replica I video circuitry will properly display lowercase, but the BASIC and Apple I software will not recognize it. Keep Caps Lock on to avoid entering any lowercase letters.

Hello World

First, we'll discuss a few steps to take when you're loading BASIC for Apple I.

The PRINT Command

We'll start at the same place at which every programming introduction starts: "Hello World." Load BASIC and type the following at the prompt:

```
10 PRINT "HELLO WORLD"  
20 END
```

Type RUN followed by pressing Return, and the program will display:

HELLO WORLD

In the aforementioned example, note the line numbers down the left column of the code. The processor executes statements in numerical order. If, later on, you came back and added the line:

```
15 PRINT "GOODBYE WORLD"
```

Line 15 would execute before line 20, even though line 20 was typed in first. If you

ran the program, the result would be:

```
HELLO WORLD  
GOODBYE WORLD
```

To see a list of all your code, type:

```
LIST
```

The program you entered will be printed to screen, line by line. If you were to type

```
10 PRINT "GOODBYE WORLD"
```

your original line 10 would be overwritten with the new line 10. This is especially useful when you discover a typo.

As you've probably guessed, the *PRINT* command prints to screen. The *END* command tells the processor when to stop executing.

Multiple strings can be printed with one *PRINT*. Separating strings with a semicolon will cause them to be printed without any space between them:

```
>PRINT "HELLO"; "WORLD"  
HELLOWORLD
```

Using commas will insert spaces so that the strings line up in columns:

```
10 PRINT "1", "2"  
20 PRINT "428", "84"  
30 END
```

```
>RUN  
1      2  
428    84
```

The TAB Command

TAB is used to insert spaces before the next *PRINT*. This is useful for formatting. For example, TAB 15 indents a line 15 spaces; TAB 20 indents 20 spaces.

```
10 TAB 15: PRINT "HELLO"  
20 TAB 20: PRINT "WORLD"  
30 END
```

```
>RUN  
      HELLO  
           WORLD
```

A colon (:) is used to place two instructions on the same line and can be used with any BASIC instruction, not just with TAB.

The GOTO Command

Suppose we want to write a really obnoxious program that says “Hello World” over and over again. This is where the GOTO command comes into play:

```
10 PRINT "HELLO WORLD"  
20 GOTO 10  
30 END
```

The program loops infinitely, printing “HELLO WORLD” each time, and never reaches line 30.

Welcome, User (Input, Variables, and Strings)

In this section, we’ll look at inputting data and storing it in variables. Variables can be one of two types, either integer strings or character strings. An integer may be named with a letter or a letter and a digit (for example, A, N, A1, B8). A character string must be named with a letter and a dollar sign (\$; for example, A\$, R\$, Z\$).

Characters are ASCII; therefore, each character in a string occupies 1 byte of memory. BASIC needs to be told how many bytes to allocate to a string, a process called dimensioning. The maximum string length is 255 characters (or bytes). For example:

```
10 PRINT "WELCOME TO THE APPLE I!"  
20 DIM N$(20)  
30 INPUT "WHAT IS YOUR NAME", N$  
40 INPUT "PICK A NUMBER", A  
50 PRINT "YOUR NAME IS "; N$  
60 PRINT "YOU PICKED "; A  
70 END
```

The output of this code will look like this:

```
>RUN  
WELCOME TO THE APPLE I!  
WHAT IS YOUR NAME?TOM  
PICK A NUMBER?5
```

```
YOUR NAME IS TOM  
YOU PICKED 5
```

```
>
```

You may notice that there is a question mark added after WHAT IS YOUR NAME. A question mark is automatically inserted after each INPUT statement. Unfortunately, there is no way to turn it off.

Let's examine the importance of quotation marks in statements. Note the difference between lines 30 and 40 in this code:

```
10 DIM N$(2)  
20 N$="HI"  
30 PRINT N$  
40 PRINT "N$"  
50 END
```

```
>RUN  
HI  
N$
```

The first line prints the string contained in the variable N\$. The second line, because of the quotes, literally prints "N\$."

Calculator (Math)

Apple I BASIC allows simple mathematical functions to be performed directly from the command line. Without writing a program, you can simply type:

```
>PRINT 8+3  
11  
>PRINT 3-7  
-4  
>PRINT (8+3)*4  
44  
>A=12  
>PRINT A-7  
5  
>B = A*3  
>PRINT B  
36
```

>

Apple I BASIC only supports integers; so, if you perform division, you'll get results like this:

```
>PRINT 38/9
4
```

You can, however, use MOD (modulus) to calculate the remainder.

```
>PRINT 38 MOD 9
2
```

From these two commands, we can ascertain that $38/9$ is equal to $4 \frac{2}{9}$.

Here's a simple program to add two numbers:

```
10 INPUT "FIRST NUM",A
20 INPUT "SECOND NUM",B
30 PRINT A+B
40 END
>RUN
FIRST NUM?9872
SECOND NUM?1111
10983
```

BASIC supports integers in the range of -32,767 to 32,767 (2 bytes). There are several mathematical functions built into BASIC (Table 5.1).

Function	Description
ABS(N)	Returns the absolute value of N. If N is positive, N will be returned. If N is negative, -N will be returned.
SGN(N)	Returns 1 if N is positive, 0 if N is 0, and -1 if N is negative.
RND(N)	Returns a random number between 0 and N - 1 if N is positive. Returns a random number between N + 1 and 0 if N is negative.

Table 5.1 BASIC's Built-In Mathematical Functions

Here's a quick example of the functions in action:

```
10 A= RND (-10000)
20 PRINT "RANDOM INT IS: ";A
30 PRINT "SIGN OF INT IS: "; SGN (A)
40 PRINT "ABSOLUTE VALUE OF INT IS: "; ABS (A)
```

```
50 END  
  
>RUN  
RANDOM INT IS: -9758  
SIGN OF INT IS: -1  
ABSOLUTE VALUE OF INT IS: 9758
```

Each time you run this program, RND will generate a new random number.

Count Down (FOR/NEXT)

The FOR and NEXT commands create a loop that runs a set number of times:

```
FOR I=a TO b STEP c  
  instructions here  
NEXT I
```

In a simple example, $a = 1$, $b = 10$, and $c = 1$. For the first iteration of the loop, $a = 0$. Then, the NEXT I instruction is reached, and a is incremented by c (in this case, 1). The program continues to loop until $b = c$. Here's an example:

```
10 INPUT "ENTER INCREMENT", I  
20 INPUT "START WHERE", S  
30 FOR V=S TO 0 STEP -I  
40 PRINT V  
50 NEXT V  
60 END  
  
>RUN  
ENTER INCREMENT?3  
START WHERE?27  
27  
24  
21  
18  
15  
12  
9  
6  
3  
0
```

If STEP is left off the FOR statement, STEP will default to +1.

Some (IF/THEN)

IF/THEN is a very powerful tool that allows the program to branch in two directions. Look at this sample:

```
10 DIM N$(20)
20 INPUT "WHAT IS YOUR NAME", N$
30 IF N$ = "TOM" THEN 100
40 PRINT "HELLO, STRANGER!"
50 END
100 PRINT "WELCOME BACK, TOM!"
110 END
```

If the user's name is Tom, the program issues a GOTO that jumps execution to line 100. If it's not Tom, the program continues executing at the next line, which is 40. IF/THEN statements can be used with an instruction. Here's a sample using PRINT:

```
10 DIM N$(20)
20 INPUT "WHAT IS YOUR NAME", N$
30 IF N$ = "SECRETAGENT" THEN PRINT "THE CROW FLIES AT MIDNIGHT."
40 PRINT "WELCOME TO THE APPLE I, "; N$; "!"
50 END

>RUN
WHAT IS YOUR NAME?TOM
WELCOME TO THE APPLE I, TOM!

>RUN
WHAT IS YOUR NAME?SECRETAGENT
THE CROW FLIES AT MIDNIGHT.
WELCOME TO THE APPLE I, SECRETAGENT!
```

If you've programmed in other languages, you're used to having an ELSE statement, such as "If *a* then *b* else *c*." Unfortunately, ELSE is not provided by Apple I BASIC.

Expressions

Apple I BASIC supports the following types of expressions:

A = B
A > B
A < B
A >= B (A is greater than or equal to B)

A <= B (A is less than or equal to B)
A <> B (A does not equal B)
A # B (same as A <> B)

Only = and # may be used with strings. These expressions evaluate to 1 if true and 0 if false. For example:

```
>PRINT 5=5  
1  
>PRINT 5=6  
0
```

The IF/THEN statement bases its operation on the resulting 1 or 0. In fact, from the command line you can even type:

```
>IF 1 THEN PRINT "HI"  
HI
```

It's possible to combine multiple statements with the AND and OR commands:

```
40 IF A=5 AND B=12 THEN PRINT "WINNER!"  
50 IF A=7 OR B=19 OR N$="TOM" OR I$="HELLO" THEN PRINT "WHAT AN  
ODD COMBINATION."
```

Parentheses can be used for nesting statements, which are useful for ordinary algebra:

```
>PRINT (5+2)*8  
56
```

More interestingly, parentheticals can also be used with AND/OR statements:

```
10 A=5  
20 B=8  
30 C=11  
40 IF A=5 AND (B=927 OR C=11) THEN PRINT "CLOSE ENOUGH!"  
50 END  
  
>RUN  
CLOSE ENOUGH!
```

Finally, we have NOT. NOT allows us to get the BASIC-equivalent of NAND and NOR logic gates. With it, we can write statements such as:

```
IF NOT (A=5 AND B=7) THEN PRINT "NOT BOTH, BUT POSSIBLY ONE"  
IF NOT (A=9 OR B=2) THEN PRINT "NEITHER"
```

The GOSUB Command

GOSUB, which is short for “go to subroutine,” is like a more advanced GOTO command. Whereas GOTO permanently branches off to a new location, GOSUB lets the user branch off, run a few lines of code, then RETURN where the program left off before branching. It works like this:

```
10 GOSUB 100
20 GOSUB 100
30 END

100 PRINT "HELLO"
110 RETURN

>RUN
HELLO
HELLO
```

Here’s another example using nested subroutines:

```
10 PRINT "START"
20 GOSUB 100
30 PRINT "FINISH"
40 END

100 PRINT "SUBROUTINE 1"
110 GOSUB 200
120 RETURN

200 PRINT "SUBROUTINE 2"
220 RETURN

>RUN
START
SUBROUTINE 1
SUBROUTINE 2
FINISH
```

When GOSUB is called, BASIC records the line number of origin. RETURN is like a GOTO, which goes back to that original point and begins executing at the next line. Up to eight subroutines can be nested. The program above uses two nested subroutines.

GOSUB is a vital tool for structuring programs. It allows us to establish subroutines, which we can repeatedly use within our program. A well-written subroutine may even be worth saving to use in future programs. In the next example, the sample program

illustrates the uses of GOSUB and GOTO. It's more complex than the previous programs we've looked at, so I've included REMs (remarks). The REM statements are not executed by BASIC. They are only displayed when the program is LISTed and exist only to provide guidance to those reading the code.

This program is similar to the battle scenes in a traditional text adventure or dungeon game:

```
10 REM DUNGEON BATTLE
20 DIM I$ (1)

30 REM HEALTH OF MONSTER M AND YOU Y
40 M = RND(1000)
45 PRINT "MONSTER'S HEALTH: ";M
50 Y = RND(1000)
55 PRINT "YOUR HEALTH: ";Y

60 REM WEAPON STRENGTHS OF MONSTER M1 AND YOU Y1
70 M1 = 250
80 Y1 = 250

100 REM USE A GOTO TO MAKE A LOOP.
101 REM IF/THEN'S ARE USED TO BREAK OUT OF THE LOOP.
120 INPUT "DO YOU WANT TO FIGHT OR RUN (F/R)", I$
130 IF I$ = "R" THEN GOTO 200
140 GOSUB 500
150 IF Y < 1 THEN GOTO 400
160 IF M < 1 THEN GOTO 300
170 GOTO 100

200 PRINT "YOU FLEE IN COWARDLY TERROR. "
210 GOTO 1000

300 PRINT "YOU HAVE VANQUISHED THE MONSTER!"
310 GOTO 1000

400 PRINT "THE MONSTER HAS EATEN YOU."
410 GOTO 1000

500 REM ATTACK
510 M = M-RND(Y1)
520 Y = Y-RND(M1)
530 PRINT "MONSTER HEALTH: "; M
540 PRINT "YOUR HEALTH: "; Y
550 RETURN

1000 END
```

Here are two sample runs of the game:

```
>RUN
MONSTERS HEALTH: 466
YOUR HEALTH: 758
DO YOU WANT TO FIGHT OR RUN (F/R)?F

MONSTER HEALTH: 335
YOUR HEALTH: 558
DO YOU WANT TO FIGHT OR RUN (F/R)?F

MONSTER HEALTH: 164
YOUR HEALTH: 474
DO YOU WANT TO FIGHT OR RUN (F/R)?F

MONSTER HEALTH: -21
YOUR HEALTH: 408
YOU HAVE VANQUISHED THE MONSTER!

>RUN
MONSTERS HEALTH: 326
YOUR HEALTH: 69
DO YOU WANT TO FIGHT OR RUN (F/R)?F

MONSTER HEALTH: 312
YOUR HEALTH: -20
THE MONSTER HAS EATEN YOU.
```

Arrays

An *array* is a collection of variables stored consecutively in memory. They are accessed by a single variable name followed by their array location in parentheses (as a subscript). Table 5.2 shows a sample array named *A*.

Location	Value
1	89
2	7
3	123
4	2
5	1015
6	42
7	123

Table 5.2 Sample Array A

In this example, 89 would be accessed with A(1), 42 with A(6), and so on. An attempt to access a value outside 1 to 7 will produce an out-of-range error (RANGE ERR).

Like a string, which is actually an array of characters, an array needs to be dimensioned with DIM. Here's a program to create an array similar to the one shown in Table 5.2 and then print all the values:

```
10 REM DETERMINE ARRAY LENGTH
20 C = RND(20)
30 DIM A(C)

100 REM LOAD VALUES
110 FOR I = 1 TO C
120 A(I) = RND(32767)
130 NEXT I

200 REM PRINT VALUES
210 FOR I=1 TO C
220 PRINT A(I)
230 NEXT I
500 END

>RUN
10411
26608
8259
785
5324
30034
32729
428
```

Arrays are a useful means of organizing data and are especially well suited to use with FOR/NEXT loops. They also provide the ability to work with a varying number of

variables. The execution of the program just shown would not be possible using ordinary variables. When each variable has its own name (for example, *A*, *B*), there is no way to loop through them, let alone declare a random number of variables.

Strings, In Depth

Apple I BASIC has considerably advanced string manipulation, given its other limitations. Extracting substrings is simple. A string's length can be retrieved with a single function call. Comparing two strings to see if they're equal is a trivial task.

Substrings

It is possible to select a substring by specifying the start and end points in a subscript. Suppose we have the string:

```
T$ = "APPLE I REPLICA CREATION"
```

This string has 24 characters, which are numbered 1 through 24. Substrings are specified in the form of T(I,J)$, where *I* is the location of the first character to be printed and *J* is the location of the last:

```
>PRINT T$(1,5)
APPLE
>PRINT T$(7,9)
I R
>PRINT T$(21,21)
T
>PRINT T$(1,24)
APPLE I REPLICA CREATION
```

It is also possible to specify only the starting point. In which case, all characters from that point on are returned:

```
>PRINT T$(11)
PLICA CREATION
```

The LEN Function

LEN returns the number of characters in a string. For example:

```
>DIM R$(20)
>R$ = "REPLICA"
>PRINT LEN(R$)
7
```

This is useful when you're using a FOR/NEXT statement to cycle through each character in a string. LEN can be used to tell the FOR/NEXT when to stop. LEN also makes it easy to combine strings, as shown in the next section.

Appending Strings

It's easy to overwrite a string, using a command such as:

```
DIM $S(20)
$S = "ORIGINAL STRING"
$S = "NEW STRING"
```

It is also possible to append one string to the end of another. To do this, specify the location where the appended string should start. For example:

```
>DIM A$(30)
>DIM B$(30)
>A$ = "ABCDEFG"
>B$ = "HIJKLMNOP"
>A$(LEN(A$)+1) = B$
>PRINT A$
ABCDEFGHIJKLMNP
```

If the location specified is before the end of the string, the rest of the string will be overwritten. Continuing from the previous example:

```
>DIM C$(30)
>C$ = "123"
>A$(4) = C$
>PRINT A$
ABC123
```

Conditionals

The = symbol can also be used to compare two strings. This works for simple comparisons of entire strings:

```
>DIM A$(30)
>DIM B$(30)
>DIM C$(30)
>A$ = "HELLO"
>B$ = "HELLO"
>C$ = "WORLD"
>PRINT (A$ = B$)
1
>PRINT (A$ = C$)
0
```

A 1 means that the strings are equal. A 0 means they are different. These comparisons can also be used in IF/THEN statements.

It is also possible to compare any two substrings:

```
>DIM A$(30)
>DIM B$(30)
>A$ = "NONETHELESS"
>B$ = "JOHANN THEILE"
>PRINT (A$(5,7) = B$(8,10))
1
```

Sample String Program

This simple program combines substrings, the LEN function, and string comparisons to reverse the order of a name. Here is a sample run of the program:

```
>RUN
ENTER YOUR FULL NAME?JAMES CONNOLLY
YOUR NAME IS: CONNOLLY, JAMES
```

The program in full is as follows:

```
10 REM FULL NAME, FIRST NAME, LAST NAME
15 DIM N$(50)
20 DIM F$(50)
25 DIM L$(50)
30 REM REVERSED NAME
35 DIM R$(50)

50 REM LOAD FULL NAME
55 INPUT "ENTER YOUR FULL NAME", N$

100 REM FIND FIRST NAME BY SEARCH FOR SPACE
105 FOR I = 1 TO LEN(N$)
```

```
110 IF N$(I,I) = " " THEN F$ = N$(1,I-1)
115 NEXT I

150 REM LAST NAME STARTS ONE CHAR AFTER THE
155 REM END OF THE FIRST NAME AND ENDS AT
160 REM END OF THE FULL NAME
165 L$ = N$(LEN(F$)+2, LEN(N$))

200 REM REARRANGE NAMES
205 R$ = L$
210 R$(LEN(R$)+1) = ", "
215 R$(LEN(R$)+1) = F$

250 PRINT "YOUR NAME IS: "; R$

300 REM WERE THOSE REALLY DIFFEENT NAMES?
305 IF F$ = L$ THEN PRINT "ARE YOU TELLING THE TRUTH?"

350 END
```

You might have noticed that the loop at lines 100–115 searches the entire string, even though there's no reason for it to continue after finding the space. Here's an alternative that breaks out of the loop once the space is found:

```
100 REM FIND FIRST NAME BY SEARCH FOR SPACE
105 FOR I = 1 TO LEN(N$)
110 IF N$(I,I) = " " THEN F$ = N$(1,I-1)
112 IF N$(I,I) = " " THEN 150
115 NEXT I
```

PEEK and POKE

PEEK and POKE are used for manually accessing memory locations. This is useful if you're interfacing with an input/output device or writing part of your program in assembly. Inconveniently, BASIC uses decimal for PEEKs and POKEs; as a result, you'll have to convert your address from hexadecimal to decimal (as discussed in Chapter 3) before POKEing or PEEKing.

To place the value 42 in location 9000, type:

```
POKE 9000,42
```

To read it back and print it to the screen, type:

```
PRINT PEEK (9000)
```

The Apple I has 65,535 memory locations. Apple I BASIC allows for integers in the range of -32,767 to 32,767. Memory locations 0 through 32767 are accessed directly. Addresses above 32,767 are accessed using Two's Complement notation (Table 5.3).

To get the Two's Complement notation for -2:

1. Convert +2 to binary.

```
0000 0000 0000 0010
```

2. Take the complement.

```
1111 1111 1111 1101
```

3. Add 1.

```
1111 1111 1111 1110
```

4. Convert to hexadecimal, for ease of use.

```
$FFFE
```

Two's Complement Notation	Hexadecimal Value
+32,767	\$7FFF
+32,766	\$7FFE
...	...
+1	\$0001
0	\$0000
-1	\$FFFF
-2	\$FFFE
...	...
-32,767	\$8001
-32,768	\$8000

Table 5.3 Two's Complement Notations and Hexadecimal Values

Since location \$8000 is represented by a value less than -32,767, it cannot be reached with BASIC.

The CALL Command

CALL is used to access a subroutine written in Assembly. It is used in the manner:

```
CALL n
```

Here, *n* is the memory location of the subroutine, which is represented in decimal format. In your Assembly subroutine, use an RTS to return control to the BASIC program.

CALL is useful if you want to write most of your program in BASIC but occasionally need a fast Assembly-language subroutine. Table 5.4 lists CALL commands and their meanings.

Command	Explanation
AUTO <i>a, b</i>	AUTO automatically numbers lines for you as they are typed. <i>a</i> is the first line number. <i>b</i> is the increment between lines. If <i>b</i> is not specified, it defaults to 10. Ctrl-D terminates AUTO mode.
CLR	CLR resets all variables, FOR loops, GOSUBs, arrays, and strings. It does not delete the program in memory.
DEL <i>a, b</i>	DEL deletes all lines of code between <i>a</i> and <i>b</i> , inclusive. If <i>b</i> is omitted, only line <i>a</i> is deleted.
LIST <i>a, b</i>	LIST displays all lines of code between <i>a</i> and <i>b</i> inclusive. If <i>b</i> is omitted, only line <i>a</i> is displayed. If both are omitted, all lines are displayed.
RUN <i>a</i>	RUN does a CLR and then begins executing code starting at line <i>a</i> . If <i>a</i> is omitted, the program begins executing at the first line.
SCR	SCR scratches (deletes) the entire program and clears memory.
LOMEM = <i>n</i>	LOMEM sets the memory floor for user programs. It is initialized to 2048. LOMEM clears all user programs and values.
HIMEM = <i>n</i>	HIMEM sets the memory ceiling for user programs. It is initialized to 4096. Replica I users can increase this value to 32767 to take advantage of the Replica I's extended memory. HIMEM clears all user programs and values.

Table 5.4 CALL Commands and Explanations

Table 5.5 lists CALL error codes and their meanings.

Error Code	Explanation
>256 ERR	A value restricted to one byte was outside the allowed range of 0 to 255.
>32767 ERR	The value entered or calculated was outside the range of (-32767, 32767).
>8 FORS ERR	There are more than eight nested FOR loops.
>8 GOSUBS ERR	There are more than eight nested subroutines.
BAD BRANCH ERR	There is an attempt to branch to a line number that does not exist.
BAD NEXT ERR	A NEXT statement is encountered without a corresponding FOR.
BAD RETURN ERR	A RETURN statement is encountered without a corresponding GOSUB.
DIM ERR	There is an attempt to dimension a character string that has already been dimensioned.
END ERR	There is no END command at the end of the program.
MEM FULL ERR	Occurs when the size of an array exceeds the amount of space available in memory.
RANGE ERR	The requested value is smaller than 1 or larger than the maximum value of the array or string.
RETYPE LINE	Data entered for an INPUT is not of the correct type.
STR OVFL ERR	The maximum length of the string is exceeded.
STRING ERR	The attempted string operation was illegal.
SYNTAX ERR	The line is not properly formatted or it contains some typo.
TOO LONG ERR	There are too many nested parenthesis in a statement.

Table 5.5 CALL Error Codes and Explanations

Richard III: Interactive Fiction

Richard III is a piece of interactive fiction based on Shakespeare's play of the same name. It is presented here to give you an example of how a small program might be constructed. The story was written by Sarah McMenomy.

Walkthrough

Before writing any code, it's necessary to figure out the basics of how the game is going to work. McMenomy wrote a walkthrough of *Richard III*. Let's look at her original design:

Intro:

King Richard: Is thy name Tyrrel?
Tyrrel: James Tyrrel, and your most obedient subject.
King Richard: Art thou indeed?
Tyrrel: Prove me, my gracious lord.
King Richard: Dar'st' thou resolve to kill a friend of mine?
Tyrrel: Please you;
But I had rather kill two enemies.
King Richard: Why, then thou hast it! Two deep enemies,
Foes to my rest and my sweet sleep's disturbers,
Are they that I would have thee deal upon:
Tyrrel, I mean those bastards in the Tower.
Tyrrel: Let me have open means to come to them,
And soon I'll rid you from the fear of them.
King Richard: Thou sing'st sweet music. Hark, come hither, Tyrrel.
Go, by this token. Rise, and lend thine ear.
There is no more but so: say it is done,
And I will love thee and prefer thee for it.
Tyrrel: I will dispatch it straight.

Outside Tower.

Ravens are flying around your head as you stand outside the tower.
There are two spigots on the wall; one says "Haute," and the other,
"Caulde." "Caulde" is on.

>N

The Ravens swoop down and peck at you, blocking your entrance.

>Turn Haute on
"Haute" is now on.

>Turn Caulde off
"Caulde" is now off. The ravens are happily bathing in the hot wa-
ter.

>N
You enter the tower.

Bottom of tower.
You are at the bottom of the tower. There is a dagger on the
floor, and
a slip of paper underneath it. There is a guard on the stairs.

>Take dagger and paper
Taken.

>Read paper
The paper says "1483."

>Up
The guard blocks your ascent.

>Kill/stab guard
The guard crumples to the floor, falling on the dagger.

>Up
You ascend the stairs.

Middle of Tower.
You are in the middle of the tower. There is a key in the corner.
There
are four number dials on the door to the north.

>Take key
Taken.

>N
The door is closed.

>Open door
You can't.

>Read dials

The dials read "0000."

>Turn first dial to 1
The dial clicks.

>Turn second dial to 4
The dial makes a whir.

>Turn third dial to 8
The dial squeaks.

>Turn fourth dial to 3
The dial whines.

>Open door
The door opens.

>N
You are so close...

Top of tower.
You are at the top of the tower. There is a door to the north.

>Unlock door
You push the key into the door and it swings open; you can see
shapes
silhouetted in the moonlight.

>N
You enter the chamber of the sleeping princes...

Tyrrel: The tyrannous and bloody act is done,
The most arch deed of piteous massacre
That ever yet this land was guilty of.
Dighton and Forrest, who I did suborn
To do this piece of ruthless butchery,
Albeit they were fleshed villains, bloody dogs,
Melted with tenderness and mild compassion,
Wept like to children in their deaths' sad story.
'O, thus,' quoth Dighton, 'lay the gentle babes.'
'Thus, thus,' quoth Forrest, 'girdling one another
Within their alabaster innocent arms.
Their lips were four red roses on a stalk,
And in their summer beauty kissed each other.
A book of prayers on their pillow lay,
Which once,' quoth Forrest, 'almost changed my mind;

But, O! The devil' - there the villain stopped;
When Dighton thus told on - 'We smothered
The most replenished sweet work of nature
That from the prime creation e'er she framed.'
Hence both are gone with conscience and remorse:
They could not speak; and so I left them both,
To bear this tidings to the bloody king.

Your villainous task is done; long live King Richard!
You have won

I originally wanted to implement a command parser, but the complexity of writing this in BASIC would have made for a very poor beginner's example. To simplify things, I decided to make the options multiple-choice. The game is less interesting this way, but it also makes for an easier demonstration.

Structure

Now that we know what the game should look like from the user's perspective, let's examine the structures needed to implement it.

Each room will be a subroutine; consequently, we can allocate each a range of lines (Table 5.6).

Room	Line Allocation
Outside Tower	1,000–1,999
Bottom of Tower	2,000–2,999
Middle of Tower	3,000–3,999
Top of Tower	4,000–4,999

Table 5.6 Line Allocations for Interactive Richard III

Most of the game will take place within the code allocated for the rooms shown in Table 5.6. However, we can make things simpler by using a few other subroutines. The Intro and Conclusion speeches can each be their own subroutines. Likewise, we can use a single subroutine to print the room descriptions. The description will be stored in a string by the room and then used by the subroutine. This will allow us to change the layout for all descriptions by simply modifying one subroutine (Table 5.7).

Subroutine	Line Allocation
Introduction	30,000–30,499
Conclusion	30,500–30,999
Initialize	31,000–31,999
PRINT Room	32,000–32,499

Table 5.7 Subroutines for Interactive Richard III

Variables

Next, let's look at the variables we'll need. Ideally, each string of text would be stored in its own variable, making it easy to update and access data. Unfortunately, Apple I BASIC only allows 26 string variables, so the less commonly used strings will have to be stored directly in PRINT statements (Tables 5.8–5.12).

Object	Variable Name	Description
Room title	R\$	These variables will be overwritten at the beginning of each room and then used by the Print Room subroutine to display the room's information.
Room description	D\$	
Temporary	T\$	A temporary variable, which can be used for any purpose.
Choice	Y	The user's input at the multiple choice.
Increment variable	I	This variable is used by the FOR/NEXT loops.

Table 5.8 Variables for All Rooms

Object	Variable Name	Description
Haute spigot	H1	The haute and caulde spigots are either on (1) or off (0). By default, haute is off and caulde is on.
Caulde spigot	C1	

Table 5.9 Variables for Haute and Caulde Spigots

Object	Variable Name	Description
Dagger	D2	The dagger is either in the player's possession (1) or not (0).
Paper	P2	The paper is either in the player's possession (1) or not (0).
Guard	G2	The guard is either alive (1) or dead (0).

Table 5.10 Variables for Bottom of Tower

Object	Variable Name	Description
Key	K3	The key is either in the player's possession (1) or not (0).
Door	D3	The door is either in the open (1) or closed (0) position.
Lock	L3	An array of four integers holds the lock combinations. Each value is initialized to 0.

Table 5.11 Variables for Middle of Tower

Object	Variable Name	Description
Door	D4	The door is either unlocked (1) or locked (0).

Table 5.12 Variables for Top of Tower

Skeleton

Each phase of the game is its own subroutine. Rooms can easily be rearranged. Adding another room simply requires adding another subroutine. Italicized names are used in place of line numbers. Omitting line numbers in the program's rough draft will make it easier to make changes along the way.

```
GOSUB initialize
GOSUB introduction
```

```
GOSUB outside_tower
GOSUB bottom_tower
GOSUB middle_tower
GOSUB top_tower
GOSUB conclusion
END
```

Initialization

The program should start by initializing all the variables we're going to use.

```
initialize:
DIM R$(255)
R$ = "X" : REM ROOM TITLE
DIM D$(255)
D$ = "X" : REM ROOM DESCRIPTION
DIM T$(255)
T$ = "X" : REM TEMPORARY VARIABLE
DIM L3(4) : REM LOCK COMBINATION
FOR I = 1 TO 4
L3(I) = 0 : REM LOCK COMBINATION SET TO 0000
NEXT I
H1 = 0 : REM HAUTE IS OFF
C1 = 1 : REM CAULDE IS ON
D2 = 0 : REM NO DAGGER
P2 = 0 : REM NO PAPER
G2 = 1 : REM GUARD ALIVE
K3 = 0 : REM NO KEY
D3 = 0 : REM DOOR CLOSED
D4 = 0 : REM DOOR LOCKED

RETURN
```

Introduction and Conclusion Subroutines

We can address the subroutines for the introduction and conclusion next because they are extremely basic. These two methods consist of nothing more than a series of PRINT statements followed by a RETURN to give control back to the calling routine.

An INPUT line halfway through pauses the scrolling text so that the user has time to read it. The user need not enter any data, but only presses keys for the program to continue.

When a line of poetry exceeds 40 characters, it is manually divided into two lines. The second line is indented four spaces using TABs.

```
PRINT "KING RICHARD: IS THY NAME TYRREL?"  
PRINT "TYRREL: JAMES TYRREL, AND YOUR MOST"  
TAB 4: PRINT "OBEDIENT SUBJECT."  
PRINT "KING RICHARD: ART THOU INDEED?"  
PRINT "TYRREL: PROVE ME, MY GRACIOUS LORD."  
PRINT "KING RICHARD: DAR'ST' THOU RESOLVE TO"  
TAB 4: PRINT "KILL A FRIEND OF MINE?"  
PRINT "TYRREL: PLEASE YOU;"  
TAB 4: PRINT "BUT I HAD RATHER KILL TWO ENEMIES."  
PRINT "KING RICHARD: WHY, THEN THOU HAST IT!"  
TAB 4: PRINT "TWO DEEP ENEMIES,"  
PRINT "FOES TO MY REST AND MY SWEET SLEEP'S"  
TAB 4: PRINT "DISTURBERS,"  
PRINT "ARE THEY THAT I WOULD HAVE THEE DEAL"  
TAB 4: PRINT "UPON:"  
PRINT "TYRREL, I MEAN THOSE BASTARDS IN THE"  
TAB 4: PRINT "TOWER."  
PRINT "TYRREL: LET ME HAVE OPEN MEANS TO COME"  
TAB 4: PRINT "TO THEM,"  
PRINT "AND SOON I'LL RID YOU FROM THE FEAR OF"  
TAB 4: PRINT "THEM."  
PRINT "KING RICHARD: THOU SING'ST SWEET MUSIC."  
TAB 4: PRINT "HARK, COME HITHER, TYRREL."  
INPUT "CONTINUE", TS  
PRINT "GO, BY THIS TOKEN. RISE, AND LEND THINE"  
TAB 4: PRINT "EAR."  
PRINT "THERE IS NO MORE BUT SO: SAY IT IS DONE,"  
PRINT "AND I WILL LOVE THEE AND PREFER THEE FOR"  
TAB 4: PRINT "IT."  
PRINT "TYRREL: I WILL DISPATCH IT STRAIGHT."  
RETURN
```

The conclusion is nearly identical:

```
PRINT "TYRREL: THE TYRANNOUS AND BLOODY ACT IS"  
TAB 4: PRINT "DONE,"  
PRINT "THE MOST ARCH DEED OF PITEOUS MASSACRE"  
PRINT "THAT EVER YET THIS LAND WAS GUILTY OF."  
PRINT "DIGHTON AND FORREST, WHO I DID SUBORN"  
PRINT "TO DO THIS PIECE OF RUTHLESS BUTCHERY,"  
PRINT "ALBEIT THEY WERE FLESHED VILLAINS,"  
TAB 4: PRINT "BLOODY DOGS,"  
PRINT "MELTED WITH TENDERNESS AND MILD"  
TAB 4: PRINT "COMPASSION,"
```

```
PRINT "WEPT LIKE TO CHILDREN IN THEIR DEATHS"
TAB 4: PRINT "SAD STORY."
PRINT "'O, THUS,' QUOTH DIGHTON, 'LAY THE'
TAB 4: PRINT "GENTLE BABES.'"
PRINT "'THUS, THUS,' QUOTH FORREST,"
TAB 4: PRINT "'GIRDLING ONE ANOTHER"
PRINT "WITHIN THEIR ALABASTER INNOCENT ARMS."
PRINT "THEIR LIPS WERE FOUR RED ROSES ON A"
TAB 4: PRINT "STALK,"
INPUT "CONTINUE", T$
PRINT "AND IN THEIR SUMMER BEAUTY KISSED EACH"
TAB 4: PRINT "OTHER."
PRINT "A BOOK OF PRAYERS ON THEIR PILLOW LAY,"
PRINT "WHICH ONCE,' QUOTH FORREST, 'ALMOST"
TAB 4: PRINT "CHANGED MY MIND;"
PRINT "BUT, O! THE DEVIL' - THERE THE"
TAB 4: PRINT "VILLAIN STOPPED;"
PRINT "WHEN DIGHTON THUS TOLD ON - 'WE"
TAB 4: PRINT "SMOTHERED"
PRINT "THE MOST REPLENISHED SWEET WORK OF NATURE"
PRINT "THAT FROM THE PRIME CREATION E'ER SHE"
TAB 4: PRINT "FRAMED.'"
PRINT "HENCE BOTH ARE GONE WITH CONSCIENCE AND"
TAB 4: PRINT "REMORSE:"
PRINT "THEY COULD NOT SPEAK; AND SO I LEFT THEM"
TAB 4: PRINT "BOTH,"
PRINT "TO BEAR THIS TIDINGS TO THE BLOODY KING."
INPUT "CONTINUE", T$
PRINT "YOUR VILLAINOUS TASK IS DONE."
PRINT "LONG LIVE KING RICHARD!"
PRINT
PRINT "***YOU HAVE WON***"
RETURN
```

PRINT Room Subroutine

PRINT Room is so simple it barely deserves its own subroutine. I wrote it this way to allow for customization. Only one line of code has to be changed to modify the display of every room title or description.

```
TAB 10: PRINT T$
PRINT D$
INPUT "CONTINUE", T$
RETURN
```

Outside Tower

Let's examine the first room. The title and description variables are set and the PRINT Room subroutine is called. Next, the program enters the CHOICE loop. The options are printed and IF statements are used to choose where to go next. Some options will print a message and return the user to the loop. Others will enter a subroutine. The Haute and Caulde subroutines allow the user to use the H1 and C1 variables. When H1 and C1 are set correctly, the CHOICE loop jumps to NEXT_LEVEL and then returns from the subroutine.

The Haute and Caulde subroutines take their variables, H1 and C1 respectively, and invert their values from 0 to 1 or 1 to 0. (Though they are technically integers, we're using them as Boolean—true or false—variables.) In some languages there is an instruction for this, but in Apple I BASIC there is not. Instead, the Haute and Caulde subroutines demonstrate two different methods for inverting a Boolean variable.

```
T$ = "OUTSIDE THE TOWER"
D$ = " RAVENS ARE FLYING AROUND YOUR HEAD AS YOU STAND OUTSIDE
THE TOWER.
THERE ARE TWO SPIGOTS ON THE WALL; ONE SAYS 'HAUTE,' AND THE
OTHER,
'CAULDE.' 'CAULDE' IS ON."

GOSUB print_room

choose:
PRINT "1) ENTER THE TOWER"
PRINT "2) TURN HAUTE SPIGOT"
PRINT "3) TURN CAULDE SPIGOT"

INPUT "CHOICE", Y
IF Y = 1 AND H1 = 1 AND C1 = 0 THEN next_level
IF Y = 1 THEN PRINT "THE RAVENS SWOOP DOWN AND PECK AT YOU,
BLOCKING YOUR ENTRANCE."
IF Y = 2 THEN GOSUB haute
IF Y = 3 THEN GOSUB caulde
INPUT "CONTINUE", T$
GOTO choose

next_level:
PRINT "YOU ENTER THE TOWER."
RETURN

haute:
H1 = H1 + 1
IF H1 = 2 THEN H1 = 0
```

```

IF H1 = 0 THEN PRINT "HAUTE IS NOW OFF."
IF H1 = 1 THEN PRINT "HAUTE IS NOW ON."
RETURN

caulde:
C1 = 1 - C1
IF C1 = 0 THEN PRINT "CAULDE IS NOW OFF."
IF C1 = 1 THEN PRINT "CAULDE IS NOW ON."
RETURN

```

Bottom of the Tower

The most interesting feature in this room is the behavior of the paper. The option READ PAPER is only displayed, and only works, if the paper is in the player's possession.

```

T$ = "BOTTOM OF THE TOWER"
D$ = "YOU ARE AT THE BOTTOM OF THE TOWER. THERE IS A DAGGER ON
THE FLOOR, AND A SLIP OF PAPER UNDERNEATH IT. THERE IS A GUARD ON
THE STAIRS."

```

print_room

```

choose:
PRINT 1) "ATTACK GUARD"
PRINT 2) "GO UP STAIRS"
PRINT 3) "PICK UP PAPER"
PRINT 4) "PICK UP DAGGER"
IF P2 = 1 THEN PRINT "5) READ PAPER"

INPUT "CHOICE", Y
IF Y = 1 THEN GOSUB attack_guard
IF Y = 2 AND G2 = 0 THEN GOTO next_level
IF Y = 2 THEN PRINT "THE GUARD BLOCKS YOUR ASCENT."
IF Y = 3 THEN GOSUB get_paper
IF Y = 4 THEN GOSUB get_dagger
IF Y = 5 AND P2 = 1 THEN PRINT "THE PAPER SAYS '1482'."
INPUT "CONTINUE", T$
GOTO choose

```

```

next_level:
PRINT "YOU ASCEND THE STAIRS."
RETURN

```

get_paper:

```

IF P2 = 1 THEN PRINT "YOU ALREADY HAVE THE PAPER."
IF P2 = 1 THEN RETURN

P2 = 1
PRINT "YOU TAKE THE PAPER."
RETURN

get_dagger:
IF D2 = 1 THEN PRINT "YOU ALREADY HAVE THE DAGGER."
IF D2 = 1 THEN RETURN

D2 = 1
PRINT "YOU TAKE THE DAGGER."
RETURN

attack_guard:
IF D2 = 1 THEN PRINT "THE GUARD KNOCKS YOU TO THE GROUND."
IF D2 = 1 THEN RETURN

G2 = 0
PRINT "THE GUARD CRUMPLES TO THE FLOOR, FALLING ON THE DAGGER."
RETURN

```

Middle of the Tower

The “combination lock” in this room is a four-integer array. Only when all four values are correctly set via the ENTER_COMBINATION subroutine can the player continue to the next room.

```

T$ = "MIDDLE OF THE TOWER"
D$ = "YOU ARE IN THE MIDDLE OF THE TOWER. THERE IS A KEY IN THE
CORNER. THERE ARE FOUR NUMBER DIALS ON THE DOOR TO THE NORTH."

```

print_room

```

choose:
PRINT "1) PICK UP KEY"
PRINT "2) OPEN DOOR"
PRINT "3) ENTER COMBINATION"
PRINT "4) ENTER ROOM"

INPUT "CHOICE", Y
IF Y = 1 THEN GOSUB get_key
IF Y = 2 AND L3(1) = 1 AND L3(2) = 4 AND L3(3) = 8 AND L3(4) = 3
THEN GOSUB open_door

```

```
IF Y = 2 THEN PRINT "THE COMBINATION IS NOT CORRECT."
IF Y = 3 THEN GOSUB enter_combination
IF Y = 4 AND D3 = 1 THEN next_level
IF Y = 4 THEN PRINT "THE DOOR IS CLOSED."
INPUT "CONTINUE", T$
GOTO choose

next_level:
PRINT "YOU ARE SO CLOSE..."
RETURN

get_key:
IF K3 = 1 THEN PRINT "YOU ALREADY HAVE THE KEY."
IF K3 = 1 THEN RETURN

K3 = 1
PRINT "YOU TAKE THE KEY."
RETURN

enter_combination:
INPUT "TURN FIRST DIAL TO", L3(1)
PRINT "THE DIAL CLICKS."
INPUT "TURN SECOND DIAL TO", L3(2)
PRINT "THE DIAL WHIRRS."
INPUT "TURN THE THIRD DIAL TO", L3(3)
PRINT "THE DIAL SQUEAKS."
INPUT "TURN THE FOURTH DIAL TO", L3(4)
PRINT "THE DIAL WHINES. "
RETURN

open_door:
IF D3 = 0 THEN PRINT "THE DOOR IS LOCKED."
IF D3 = 0 THEN RETURN
PRINT "THE DOOR OPENS."
D3 = 1
RETURN
```

Top of the Tower

The final room is trivial. The door is unlocked with the key from the previous room—but only if the player picked it up! If you’re considering to expand this game, the option to move freely between rooms, picking up and dropping objects, would surely be a welcome feature.

```
T$ = "TOP OF THE TOWER"
D$ = "YOU ARE AT THE TOP OF THE TOWER. THERE IS A DOOR TO THE
NORTH."
print_room

choose:
PRINT "1) UNLOCK THE DOOR"
PRINT "2) ENTER THE CHAMBER."

INPUT "CHOICE", Y
IF Y = 1 THEN GOSUB unlock_door
IF Y = 2 AND D4 = 1 THEN next_level
IF Y = 2 THEN PRINT "THE DOOR IS LOCKED."
INPUT "CONTINUE", T$
GOTO choose

next_level:
PRINT "YOU ENTER THE CHAMBER OF THE SLEEPING PRINCES... "
RETURN

unlock_door:
IF D4 = 1 THEN PRINT "THE DOOR IS ALREADY UNLOCKED"
IF D4 = 1 THEN RETURN

D4 = 1
PRINT "YOU PUSH THE KEY INTO THE DOOR AND IT SWINGS OPEN; YOU CAN
SEE SHAPES SILHOUETTED IN THE MOONLIGHT."
RETURN
```

Richard III Code

Here is the complete code for *Richard III*. It can also be found in the Supplemental Software package, so those with a serial I/O board from Briel Computers will be able to transfer it to their replica via the terminal.

The italicized headers from earlier sections have been replaced with REM lines to allow them to be stored in the program. Before you begin entering lines, be sure to increase HIMEM to 32767. This program is too large to run in the standard 8K.

```
10 REM RICHARD III: (BARELY) INTERACTIVE FICTION
30 GOSUB 31000 : REM INITIALIZE
40 GOSUB 30000 : REM INTRODUCTION
50 GOSUB 1000 : REM OUTSIDE OF TOWER
60 GOSUB 2000 : REM BOTTOM OF TOWER
70 GOSUB 3000 : REM MIDDLE OF TOWER
80 GOSUB 4000 : REM TOP OF TOWER
90 GOSUB 30500 : REM CONCLUSION
100 END
```

```
31000 REM INITIALIZE
31010 DIM R$(255)
31015 R$ = "X" : REM ROOM TITLE
31020 DIM D$(255)
31025 D$ = "X" : REM ROOM DESCRIPTION
31030 DIM T$(255)
31035 T$ = "X" : REM TEMPORARY VARIABLE
31040 DIM L3(4) : REM LOCK COMBINATION
31045 FOR I = 1 TO 4
31050 L3(I) = 0 : REM LOCK COMBINATION SET TO 0000
31055 NEXT I
31060 H1 = 0 : REM HAUTE IS OFF
31065 C1 = 1 : REM CAULDE IS ON
31070 D2 = 0 : REM NO DAGGER
31075 P2 = 0 : REM NO PAPER
31080 G2 = 1 : REM GUARD ALIVE
31085 K3 = 0 : REM NO KEY
31090 D3 = 0 : REM DOOR CLOSED
31095 D4 = 0 : REM DOOR LOCKED
31100 RETURN
```

```
30000 REM INTRODUCTION
30010 PRINT "KING RICHARD: IS THY NAME TYRREL?"
30015 PRINT "TYRREL: JAMES TYRREL, AND YOUR MOST"
30020 TAB 4: PRINT "OBEDIENT SUBJECT."
30025 PRINT "KING RICHARD: ART THOU INDEED?"
30030 PRINT "TYRREL: PROVE ME, MY GRACIOUS LORD."
30035 PRINT "KING RICHARD: DAR'ST' THOU RESOLVE TO"
30040 TAB 4: PRINT "KILL A FRIEND OF MINE?"
30045 PRINT "TYRREL: PLEASE YOU;"
30050 TAB 4: PRINT "BUT I HAD RATHER KILL TWO ENEMIES."
30055 PRINT "KING RICHARD: WHY, THEN THOU HAST IT!"
30060 TAB 4: PRINT "TWO DEEP ENEMIES,"
```

```
30065 PRINT "FOES TO MY REST AND MY SWEET SLEEP'S"
30070 TAB 4: PRINT "DISTURBERS,"
30075 PRINT "ARE THEY THAT I WOULD HAVE THEE DEAL"
30080 TAB 4: PRINT "UPON:"
30085 PRINT "TYRREL, I MEAN THOSE BASTARDS IN THE"
30090 TAB 4: PRINT "TOWER."
30095 PRINT "TYRREL: LET ME HAVE OPEN MEANS TO COME"
30100 TAB 4: PRINT "TO THEM,"
30105 PRINT "AND SOON I'LL RID YOU FROM THE FEAR OF"
30110 TAB 4: PRINT "THEM."
30115 PRINT "KING RICHARD: THOU SING'ST SWEET MUSIC."
30120 TAB 4: PRINT "HARK, COME HITHER, TYRREL."
30125 INPUT "CONTINUE", T$
30130 PRINT "GO, BY THIS TOKEN. RISE, AND LEND THINE"
30135 TAB 4: PRINT "EAR."
30140 PRINT "THERE IS NO MORE BUT SO: SAY IT IS DONE,"
30145 PRINT "AND I WILL LOVE THEE AND PREFER THEE FOR"
30150 TAB 4: PRINT "IT."
30155 PRINT "TYRREL: I WILL DISPATCH IT STRAIGHT."
30160 INPUT "CONTINUE", T$
30165 RETURN
```

```
30500 REM CONCLUSION
30505 PRINT "TYRREL: THE TYRANNOUS AND BLOODY ACT IS"
30510 TAB 4: PRINT "DONE,"
30515 PRINT "THE MOST ARCH DEED OF PITEOUS MASSACRE"
30520 PRINT "THAT EVER YET THIS LAND WAS GUILTY OF."
30525 PRINT "DIGHTON AND FORREST, WHO I DID SUBORN"
30530 PRINT "TO DO THIS PIECE OF RUTHLESS BUTCHERY,"
30535 PRINT "ALBEIT THEY WERE FLESHED VILLAINS,"
30540 TAB 4: PRINT "BLOODY DOGS,"
30545 PRINT "MELTED WITH TENDERNESS AND MILD"
30550 TAB 4: PRINT "COMPASSION,"
30555 PRINT "WEPT LIKE TO CHILDREN IN THEIR DEATHS'"
30560 TAB 4: PRINT "SAD STORY."
30565 PRINT "'O, THUS,' QUOTH DIGHTON, 'LAY THE"
30570 TAB 4: PRINT "GENTLE BABES.'"
30575 PRINT "'THUS, THUS,' QUOTH FORREST,"
30580 TAB 4: PRINT "'GIRDLING ONE ANOTHER"
30585 PRINT "WITHIN THEIR ALABASTER INNOCENT ARMS."
30590 PRINT "THEIR LIPS WERE FOUR RED ROSES ON A"
30595 TAB 4: PRINT "STALK,"
30600 INPUT "CONTINUE", T$
30605 PRINT "AND IN THEIR SUMMER BEAUTY KISSED EACH"
```

```
30610 TAB 4: PRINT "OTHER."
30615 PRINT "A BOOK OF PRAYERS ON THEIR PILLOW LAY,"
30620 PRINT "WHICH ONCE,' QUOTH FORREST, 'ALMOST"
30625 TAB 4: PRINT "CHANGED MY MIND;"
30630 PRINT "BUT, O! THE DEVIL' - THERE THE"
30635 TAB 4: PRINT "VILLAIN STOPPED;"
30640 PRINT "WHEN DIGHTON THUS TOLD ON - 'WE"
30645 TAB 4: PRINT "SMOTHERED"
30650 PRINT "THE MOST REPLENISHED SWEET WORK OF NATURE"
30655 PRINT "THAT FROM THE PRIME CREATION E'ER SHE"
30660 TAB 4: PRINT "FRAMED.'"
30665 PRINT "HENCE BOTH ARE GONE WITH CONSCIENCE AND"
30670 TAB 4: PRINT "REMORSE:"
30675 PRINT "THEY COULD NOT SPEAK; AND SO I LEFT THEM"
30680 TAB 4: PRINT "BOTH,"
30685 PRINT "TO BEAR THIS TIDINGS TO THE BLOODY KING."
30690 INPUT "CONTINUE", T$
30700 PRINT "YOUR VILLAINOUS TASK IS DONE."
30705 PRINT "LONG LIVE KING RICHARD!"
30710 PRINT
30715 PRINT "****YOU HAVE WON****"
30720 INPUT "CONTINUE", T$
30725 RETURN

32000 REM PRINT ROOM
32010 TAB 10: PRINT T$
32015 PRINT D$
32020 INPUT "CONTINUE", T$
32025 RETURN

1000 REM OUTSIDE THE TOWER
1010 T$ = "OUTSIDE THE TOWER"
1015 D$ = "RAVENS ARE FLYING AROUND YOUR HEAD AS YOU STAND
OUTSIDE THE TOWER. THERE ARE TWO SPIGOTS ON THE WALL; "
1016 D$(LEN(D$)+1) = "ONE SAYS 'HAUTE,' AND THE OTHER, 'CAULDE.'
'CAULDE' IS ON."
1020 GOSUB 32000

1025 PRINT "1) ENTER THE TOWER"
1030 PRINT "2) TURN HAUTE SPIGOT"
1035 PRINT "3) TURN CAULDE SPIGOT"

1040 INPUT "CHOICE", Y
1045 IF Y = 1 AND H1 = 1 AND C1 = 0 THEN 1200
1050 IF Y = 1 THEN PRINT "THE RAVENS SWOOP DOWN AND PECK AT YOU,"
```

```
BLOCKING YOUR ENTRANCE."
1060 IF Y = 2 THEN GOSUB 1300
1065 IF Y = 3 THEN GOSUB 1400
1070 INPUT "CONTINUE",T$
1075 GOTO 1025

1200 REM NEXT LEVEL
1205 PRINT "YOU ENTER THE TOWER."
1210 RETURN

1300 REM HAUTE
1305 H1 = H1 + 1
1310 IF H1 = 2 THEN H1 = 0
1315 IF H1 = 0 THEN PRINT "HAUTE IS NOW OFF."
1320 IF H1 = 1 THEN PRINT "HAUTE IS NOW ON."
1325 RETURN

1400 REM CAULDE
1405 C1 = C1 - 1
1410 C1 = ABS(C1)
1415 IF C1 = 0 THEN PRINT "CAULDE IS NOW OFF."
1420 IF C1 = 1 THEN PRINT "CAULDE IS NOW ON."
1425 RETURN

2000 REM BOTTOM OF THE TOWER
2010 T$ = "BOTTOM OF THE TOWER"
2015 D$ = "YOU ARE AT THE BOTTOM OF THE TOWER. THERE IS A DAGGER
ON THE FLOOR, AND A SLIP OF PAPER UNDERNEATH IT."
2016 D$(LEN(D$)+1) = "THERE IS A GUARD ON THE STAIRS."

2020 GOSUB 32000

2025 PRINT "1) ATTACK GUARD"
2030 PRINT "2) GO UP STAIRS"
2035 PRINT "3) PICK UP PAPER"
2040 PRINT "4) PICK UP DAGGER"
2045 IF P2 = 1 THEN PRINT "5) READ PAPER"

2050 INPUT "CHOICE", Y
2055 IF Y = 1 THEN GOSUB 2500
2060 IF Y = 2 AND G2 = 0 THEN GOTO 2200
2065 IF Y = 2 THEN PRINT "THE GUARD BLOCKS YOUR ASCENT."
2070 IF Y = 3 THEN GOSUB 2300
2075 IF Y = 4 THEN GOSUB 2400
2080 IF Y = 5 AND P2 = 1 THEN PRINT "THE PAPER SAYS '1482' ."
2085 INPUT "CONTINUE",T$
```

```
2090 GOTO 2025

2200 REM NEXT LEVEL
2205 PRINT "YOU ASCEND THE STAIRS."
2210 RETURN

2300 REM GET PAPER
2305 IF P2 = 1 THEN PRINT "YOU ALREADY HAVE THE PAPER."
2310 IF P2 = 1 THEN RETURN
2315 P2 = 1
2320 PRINT "YOU TAKE THE PAPER."
2325 RETURN

2400 REM GET DAGGER
2405 IF D2 = 1 THEN PRINT "YOU ALREADY HAVE THE DAGGER."
2410 IF D2 = 1 THEN RETURN
2415 D2 = 1
2420 PRINT "YOU TAKE THE DAGGER."
2425 RETURN

2500 REM ATTACK GUARD
2505 IF D2 = 0 THEN PRINT "THE GUARD KNOCKS YOU TO THE GROUND."
2510 IF D2 = 0 THEN RETURN
2515 G2 = 0
2520 PRINT "THE GUARD CRUMPLES TO THE FLOOR, FALLING ON THE DAG-
GER."
2525 RETURN

3000 REM MIDDLE OF THE TOWER
3010 T$ = "MIDDLE OF THE TOWER"
3015 D$ = "YOU ARE IN THE MIDDLE OF THE TOWER. THERE IS A KEY IN
THE CORNER."
3016 D$(LEN(D$)+1) = "THERE ARE FOUR NUMBER DIALS ON THE DOOR TO
THE NORTH."

3020 GOSUB 32000

3025 PRINT "1) PICK UP KEY"
3030 PRINT "2) OPEN DOOR"
3035 PRINT "3) ENTER COMBINATION"
3040 PRINT "4) ENTER ROOM"

3045 INPUT "CHOICE", Y
3050 IF Y = 1 THEN GOSUB 3200
3055 IF Y = 2 AND L3(1) = 1 AND L3(2) = 4 AND L3(3) = 8 AND L3(4)
= 3 THEN GOSUB 3400
```

```
3060 IF Y = 2 AND D3 = 0 THEN PRINT "THE COMBINATION IS NOT COR-
RECT."
3065 IF Y = 3 THEN GOSUB 3300
3070 IF Y = 4 AND D3 = 1 THEN 3100
3075 IF Y = 4 THEN PRINT "THE DOOR IS CLOSED."
3080 INPUT "CONTINUE", T$
3085 GOTO 3025

3100 REM NEXT LEVEL
3105 PRINT "YOU ARE SO CLOSE..."
3110 RETURN

3200 REM GET KEY
3205 IF K3 = 1 THEN PRINT "YOU ALREADY HAVE THE KEY."
3210 IF K3 = 1 THEN RETURN
3215 K3 = 1
3220 PRINT "YOU TAKE THE KEY."
3225 RETURN

3300 REM ENTER COMBINATION
3305 INPUT "TURN FIRST DIAL TO", L3(1)
3310 PRINT "THE DIAL CLICKS."
3315 INPUT "TURN SECOND DIAL TO", L3(2)
3320 PRINT "THE DIAL WHIRRS."
3325 INPUT "TURN THE THIRD DIAL TO", L3(3)
3330 PRINT "THE DIAL SQUEAKS."
3335 INPUT "TURN THE FOURTH DIAL TO", L3(4)
3340 PRINT "THE DIAL WHINES."
3345 RETURN

3400 REM OPEN DOOR
3405 PRINT "THE DOOR OPENS."
3410 D3 = 1
3415 RETURN

4000 REM TOP OF THE TOWER
4010 T$ = "TOP OF THE TOWER"
4015 D$ = "YOU ARE AT THE TOP OF THE TOWER. THERE IS A DOOR TO
THE NORTH."

4020 GOSUB 32000

4025 PRINT "1) UNLOCK THE DOOR"
4030 PRINT "2) ENTER THE CHAMBER." 4035 INPUT "CHOICE", Y
4040 IF Y = 1 THEN GOSUB 4200
4045 IF Y = 2 AND D4 = 1 THEN 4100
```

```
4050 IF Y = 2 THEN PRINT "THE DOOR IS LOCKED."
4055 INPUT "CONTINUE",T$
4060 GOTO 4025

4100 REM NEXT LEVEL
4105 PRINT "YOU ENTER THE CHAMBER OF THE SLEEPING PRINCES..."
4110 RETURN

4200 REM UNLOCK DOOR
4205 IF K3 = 0 THEN PRINT "YOU DO NOT HAVE A KEY"
4210 IF K3 = 0 THEN RETURN
4215 IF D4 = 1 THEN PRINT "THE DOOR IS ALREADY UNLOCKED"
4220 IF D4 = 1 THEN RETURN
4225 D4 = 1
4230 PRINT "YOU PUSH THE KEY INTO THE DOOR AND IT SWINGS OPEN;
YOU CAN SEE SHAPES SILHOUETTED IN THE MOONLIGHT."
4235 RETURN
```

Summary

This chapter has attempted to provide a reasonable introduction to programming the Apple I in BASIC. Particular emphasis has been placed on the peculiarities of the Apple I's version of BASIC. For more sample programs written in Apple I BASIC, visit the Apple I Owners Club on Applefritter. If you'd like a more in-depth guide to programming BASIC in general, most public libraries will have countless books on the subject.

Chapter 6

Programming in Assembly

In this Chapter

- Using the Monitor
- Setting Up the Assembler
- Registers
- Hello World
- TV Typewriter
- X and Y
- Memory Addressing
- Interacting with Memory
- Printing Strings
- String Subroutine
- Bit Representation
- Using the Stack
- Bit Manipulation
- Math

Introduction

Programming in Assembly, a low-level programming language, can be difficult. However, it's also a very rewarding process that allows us to get to the core of how the processor interacts with other components. The programmer issues arcane instructions to the processor and those instructions are exactly what the processor does. BASIC, by contrast, is a high-level language. Instructions in BASIC are translated into a series of Assembly instructions. Thus with BASIC, it's difficult to know precisely what the processor is doing, whereas with Assembly, we control the processor directly.

Using the Monitor

Power on your Apple I replica and hit Reset. The `\` prompt means you've entered the Monitor. In the Monitor, it's possible to read and write data to and from memory. Everything is presented in hexadecimal format. To view the contents of location `$FFEF`, for example, type:

```
FFEF
```

The output will be:

```
FFEF: 2C
```

It is also possible to view a range of addresses by specifying the beginning and end address with a period in between, such as:

```
FFF0.FFFF
```

```
FFF0: 12 D0 30 FB 8D 12 D0 60  
FFF8: 00 00 00 0F 00 FF 00 01
```

Note that only the address for the first value of each line is given. `FFF0` contains `$12`, `FFF1` contains `$D0`, etc. Each line displays the contents of eight bytes.

Non-consecutive memory locations can be viewed by separating each address by a space:

```
EFA1 FFEF FFFF
```

```
FFA1: CA  
FFEF: 2C  
FFFF: 01
```

All the addresses we've been looking at are in the `$FFxx` range. This range was chosen because `$FFxx` contains the Monitor, which guarantees we'll be looking at known values. From here onward, we'll be working in main memory. If you're using your own replica with the traditional 8 KB RAM, you have 4 KB of main memory, located at `$0000.0FFF`. If you're using a Replica I, you have `$0000.7FFF` (32 KB) available.

We can edit data in a similar manner to how we view it. To insert `$AA` in `$0000`, type:

```
0000:AA
```

The output will be the previous content of `$0000`:

```
0000: 02
```

Now, examine \$0000 to view the new contents:

```
0000
```

```
0000: AA
```

It is also possible to enter multiple bytes of data at once. For example:

```
0000: 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14
```

```
0000: AA
```

```
0000.0017
```

```
0000: 01 02 03 04 05 06 07 08
```

```
0008: 09 0A 0B 0C 0D 0E 0F 10
```

```
0010: 11 12 13 14 68 FF 02 FF
```

Finally, to begin running a program, specify the line number and append an R for RUN. To run the program we just reviewed, for example, you would type:

```
0000R
```

If you try this right now, the system will hang because the program is meaningless. Hit Reset to recover. You should now have a basic working understanding of the Monitor, which will allow you to enter programs. If you'd like to learn more, see the Apple I Operations Manual found at the Apple I Owners Club.

Set Up the Assembler

Assembly code is written using mnemonics such as LDA and JMP. An *assembler* translates these mnemonics into hexadecimal codes that can be understood by the processor. The conversion can be done by hand. Appendix B shows the conversions to opcodes. Assemble a program by hand, and you can type it directly into the Apple I Monitor and run it. The conversion is simple, but very tedious. A much easier method is to use an assembler program such as xa65 by Andre Fachat and Cameron Kaiser.

xa65 is released under the GNU General Public License. Source code is included in the Supplemental Software package as well as a compiled version for Mac OS X. If you're running OS X, copy the application's directory to your hard drive. Open Terminal (in Applications/Utilities) and navigate to the xa65 folder. Your command will look something like this, depending upon to where you copied the folder:

```
cd /Users/owad/xa-2.1.4h
```

“cd” stands for “change directory.” To run the program, type:

```
./xa
```

This will present the help screen:

```
Cross-Assembler 65xx V2.1.4h 12dec1998 (c) 1989-98 by A.Fachat
usage : xa { option | sourcefile }
options:
  -v      = verbose output
  -x      = old filename behaviour (overrides -o, -e, -l)
  -C      = no CMOS-opcodes
  -B      = show lines with block open/close
  -c      = produce o65 object instead of executable files (i.e. do
not link)
  -o filename = sets output filename, default is 'a.o65'
            A filename of '-' sets stdout as output file
  -e filename = sets errorlog filename, default is none
  -l filename = sets labellist filename, default is none
  -r      = adds crossreference list to labellist (if -l given)
  -M      = allow ":" to appear in comments, for MASM compatibility
  -R      = start assembler in relocating mode
  -Llabel = defines 'label' as absolute, undefined label even when
linking
  -b? adr  = set segment base address to integer value adr.
            '?' stands for t(ext), d(ata), b(ss) and z(ero) segment
            (address can be given more than once, latest is taken)
  -A adr  = make text segment start at an address that when the
_file_
            starts at adr, relocation is not necessary. Overrides -bt
            Other segments have to be take care of with -b?
  -G      = suppress list of exported globals
  -DDEF=TEXT = defines a preprocessor replacement
  -Idir  = add directory 'dir' to include path (before XAINPUT)
Environment:
  XAINPUT = include file path; components divided by ','
  XAOOUTPUT= output file path
```

There are only a couple of options we’re worried about at the moment: -v, -o, -bt, and -C. -v turns on verbose output. This provides more error messages, more detailed descriptions, and a better idea of what’s going on. If something’s not working right, turn on this option to get more details on the problem. -o lets us specify what the output file will be called. -bt allows us to specify the starting address of the code. Note that to avoid modifying blocks used by the Monitor, our programs will start at location \$0280 (decimal 640). -C turns off support for opcodes that were added in the 65c02. Since we won’t

be using these opcodes, this isn't necessary at the present time, but if you're assembling someone else's code, turning on this option will warn you if they've used any opcodes that your processor doesn't support.

Our typical assemble command will look like this:

```
./xa -o samplefile.hex -bt640 samplefile
```

samplefile is the input; samplefile.hex is the output. Hex files, such as samplefile.hex, can be viewed using HexEdit, which is included in the Supplemental Software package.

Reserved Addresses

The Apple I Monitor uses memory locations \$0024 through \$002B as index pointers and locations \$0200 through \$027F as input buffer storage. The Monitor will not let you edit these locations. It will let you try and it won't give you any error message, but when you go to examine the contents you'll find it unchanged.

Registers

You're going to find that moving data around consumes a massive portion of the Apple I processor's time. There are three general-purpose *registers* in the 6502, each of which hold one byte of data. These are the *Accumulator*, *X*, and *Y*. The Accumulator is the register that handles all arithmetic operations and is the one through which most data passes. *X* and *Y* are generic registers for temporarily storing data that you want to keep close to the processor. Additional registers will be discussed in Chapter 7.

Hello World

Our first assembly program, like our first BASIC program, is HELLO WORLD. We'll need three instructions for this: LDA, JSR, and JMP.

JMP is the mnemonic for "jump." It's the equivalent of the GOTO instruction in BASIC—though instead of specifying line numbers, we specify memory addresses. When our program is finished running we want to return control to the Monitor. We do this by jumping to location \$FF1F. To jump to the Monitor you would type:

```
JMP $FF1F
```

The xa65 assembler allows us to use labels to represent physical addresses, so that at the start of the file you could state:

```
Monitor = $FF1F
```

And then later on in the code, you should include the instruction:

```
JMP Monitor
```

JSR stands for “jump subroutine”. It is the equivalent of BASIC’s GOSUB instruction. The Apple I Monitor has a subroutine called Echo which takes whatever value is in the Accumulator and sends it to the display. First, we load an ASCII value (see chart in Appendix A). Then we call the Echo subroutine. The character is printed and control returns to the main program. To jump to the Echo subroutine you would type:

```
JSR $FFEF
```

LDA is the mnemonic for “load Accumulator”. The # symbol is used to indicate “Immediate Addressing” mode. This means that the actual value specified—as opposed to the value stored at that address in memory—is loaded into the Accumulator. For example, to load the value \$3B into the Accumulator, you would type:

```
LDA #$3B
```

Our “Hello World” program should load ‘H’ into the accumulator, jump to the Echo subroutine, and then move on to the next letter. When it’s done, control should be returned to the Monitor. Since all these letters could get a bit tedious, let’s make it “HI W!” instead of “HELLO WORLD”:

```
Echo = $FFEF
Monitor = $FF1F

LDA #$48 ; H in ASCII is $48
JSR Echo
LDA #$49 ; I
JSR Echo
LDA #$20 ; SPC
JSR Echo
LDA #$57 ; W
JSR Echo
LDA #$21 ; !
JSR Echo

JMP Monitor ; Return to Monitor
```

Save this file in the same directory as xa65 with the name “hiw”. In Terminal, navi-

gate to that directory and run the line:

```
./xa -o hiw.hex -bt640 hiw
```

Now open `hiw.hex` in HexEdit (Figure 6.1). If you're using a Serial I/O card on your Apple I, select all the text and copy it into your text editor. In your text editor (such as SubEthaEdit or BBEdit), select the linebreak and copy it into the clipboard. Use the editor's Find/Replace function to replace all linebreaks with a single space. This will place the entire program on a single line, which is what we'll need when we attempt to send it to the Apple I. If you don't have a Serial I/O card, you'll have to type the program in by hand.

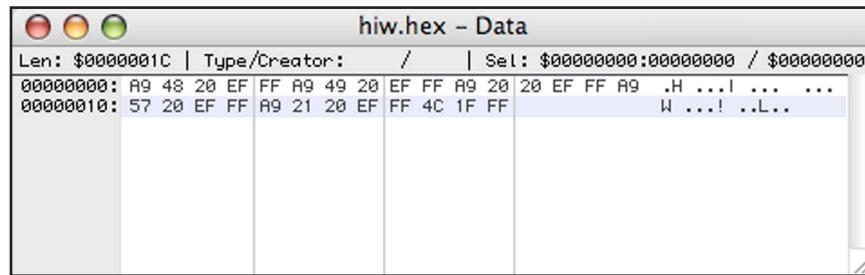


Figure 6.1 Hexadecimal Codes in HexEdit

The assembled machine code for `hiw` is:

```
A9 48 20 EF FF A9 49 20 EF FF A9 20 20 EF FF A9 57 20 EF FF A9 21
20 EF FF 4C 1F FF
```

We want to load this into the memory, starting at location \$0280; so therefore, we need to enter the line:

```
0280: A9 48 20 EF FF A9 49 20 EF FF A9 20 20 EF FF A9 57 20 EF FF
A9 21 20 EF FF 4C 1F FF
```

Type R to begin running at the most recently examined location (\$0280), or specify a line specifically:

```
0280R
```

The program will output:

```
HI W!
```

and return control to the Monitor. If the program doesn't run correctly, examine it in memory. You should get:

```
0280.029D
```

```

0280: A9 48 20 EF FF A9 49 20
0288: EF FF A9 20 20 EF FF A9
0290: 57 20 EF FF A9 21 20 EF
0298: FF 4C 1F FF E4 FF

```

How do those Assembly instructions translate into machine code? Here's a line-by-line comparison (Table 6.1).

Assembly Code	Machine Code
LDA #\$48	A9 48
JSR \$FFEF	20 EF FF
LDA #\$49	A9 49
JSR \$FFEF	20 EF FF
LDA #\$20	A9 20
JSR \$FFEF	20 EF FF
LDA #\$57	A9 57
JSR \$FFEF	20 EF FF
LDA #\$21	A9 21
JSR \$FFEF	20 EF FF
JMP \$FF1F	4C 1F FF

Table 6.1 Machine Code

From this table, we can ascertain that \$A9 is the opcode for LDA (when using Immediate Addressing). \$A9 is followed by the value being loaded into the Accumulator. The opcode for JSR is \$20. It's followed by the address of the subroutine in reverse order—low byte first, then high byte. JMP's opcode is \$4C and uses the same addressing method as JSR.

TV Typewriter

Hello World demonstrated how to output text to the screen. This program will illustrate how to read it from the keyboard. First, we need to introduce two new Assembly instructions: RTS and BPL.

RTS is short for “Return from Subroutine.” It is the equivalent of RETURN in BASIC. JSR (jump to subroutine) is used to enter a subroutine; RTS is used to return from it.

BPL is the mnemonic for “Branch on Plus.” It’s similar to an IF and GOTO statement

in BASIC. BPL appears in the form:

```
BPL addr
```

If the value in the Accumulator is positive or zero, BPL branches to the location specified by addr. If the value in the Accumulator is negative, BPL does not branch, but continues execution at the next line.

There are also two new memory addresses we need to acquaint ourselves with. They are KbdRdy (\$D011) and Kbd (\$D010). These addresses are both on the 6821 PIA. The value at KbdRdy is negative when there is data waiting to be read from the keyboard. When there is no data waiting, it is positive. The location Kbd contains the value of the character read in from the keyboard.

A subroutine to read data from the keyboard should loop (wait) until a character is available and then load that character into the Accumulator and return:

```
KbdRdy = $D011
Kbd = $D010

GetChar:
    LDA KbdRdy    ; Key ready?
    BPL GetChar   ; Loop until ready
    LDA Kbd       ; Load character
    RTS          ; Return
```

Note the use of the GetChar label in the above example. BPL branches to whatever address at which GetChar is. GetChar is whatever address LDA KbdRdy begins at. The assembler will figure out what these actual addresses are and insert numbers.

Let's configure our TV Typewriter program so that it reads a character from the keyboard, echoes that character to the screen, and repeats this loop indefinitely:

```
Begin:
    JSR GetChar    ; Read character from keyboard and store in accumulator
    JSR Echo       ; Echo value in accumulator to screen
    JMP Begin     ; Loop forever
```

The completed program might look like this:

```
Echo = $FFEF
KbdRdy = $D011
Kbd = $D010

Begin:
    JSR GetChar    ; Read character from keyboard and store in accumulator
```

```

JSR Echo      ; Echo value in accumulator to screen
JMP Begin     ; Loop forever

GetChar:
LDA KbdRdy   ; Key ready?
BPL GetChar  ; Loop until ready
LDA Kbd       ; Load character
RTS          ; Return

```

Assemble the program with the instruction:

```
./xa -o tvtypewriter.hex -bt640 tvtypewriter
```

This will produce the machine code:

```
20 89 02 20 EF FF 4C 80 02 AD 11 D0 10 FB AD 10 D0 60
```

Precede that with the starting memory location (0280:) and copy the whole string into the Apple I Monitor, just as we did for Hello World. Hit R and Return to run the program. Your Apple I will now allow you to display whatever you can type. To clear the screen without halting the program, press the Clear button on the motherboard. To halt execution, press Reset.

X and Y

The X and Y registers are very similar to the Accumulator, but without the additional mathematical abilities discussed later. One mathematical function X and Y can perform, though, is increment. The INX and INY instructions increment X and Y by one, respectively. The converse is also available: DEX and DEY are used to decrement.

X and Y support the basic memory addressing operations supported by the Accumulator (see Appendix C for specific addressing modes). Several instructions allow us to transfer values between the Accumulator and the X and Y registers (Table 6.2).

Instruction	Description
TAX	Transfer Accumulator to X
TAY	Transfer Accumulator to Y
TXA	Transfer X to Accumulator
TYA	Transfer Y to Accumulator

Table 6.2 Transfer Instructions

Note that it is not possible to transfer directly between X and Y without first going through the Accumulator.

In the sample program below, \$00 is loaded into X. X is incremented and then transferred to the Accumulator. The Accumulator is echoed to screen, then the loop is repeated. This will print every value from 0 through 255 and then loop back to the beginning. Since our display interprets all values as ASCII codes, you won't see the numbers 0 through 255. Instead, you'll see all the symbols those values represent.

```
Echo = $FFEF

Begin:
LDX #$00 ; Load 0 into X

Loop:
INX      ; Increment X
TXA      ; Transfer X to A
JSR Echo ; Echo A to screen
JMP Loop ; Repeat loop
```

To run the program, assemble it with xa65 and copy the hexadecimal code to the Apple I Monitor, as we did in the Hello World and TV Typewriter examples.

Memory Addressing

The 6502 has a myriad of addressing techniques used for moving data between memory and the processor's registers. The most basic of these are used constantly. Some of the more complex modes will seem almost esoteric, but they're invaluable under the right circumstances. The examples in this section refer to the data in Table 6.3.

Location	Contents
X (register)	\$02
Y (register)	\$03
PC (register)	\$C100
\$0000	
\$0001	
\$0002	\$B3
\$0003	\$5A
\$0004	\$EF
\$0017	\$10
\$0018	\$D0
\$002A	\$35
\$002B	\$C2
\$D010	\$33
\$C238	\$2F

Table 6.3 Memory

Accumulator: A

The Accumulator is implied as the operand in Accumulator addressing and thus no address needs to be specified. An example is the ASL (Arithmetic Shift Left) instruction, covered later in this chapter. It is understood that the value in the Accumulator is always the value being shifted left.

Implied: *i*

With *Implied addressing*, the operand is implied and therefore does not need to be specified. An example is the TXA instruction.

Immediate:

Immediate addressing loads the operand directly into the Accumulator. In this example:

```
LDA #$22
```

The Accumulator now contains the value \$22.

Absolute: *a*

Absolute addressing specifies a full 16-bit address. For example:

```
LDA $D010
```

The Accumulator now contains \$33, the value stored at location \$D010.

Zero Page: *zp*

Zero Page addressing uses a single byte to specify an address on the first page of memory (\$00xx). Thus, in the example:

```
LDA $02
```

The value \$B3 at location \$0002 is loaded into the Accumulator.

Relative: *r*

Relative addressing specifies memory locations relative to the present address. The current address is stored in the Program Counter (PC). Relative addressing adds the address in PC to the specified offset. In this example:

```
BPL $2D
```

PC is \$C100 and the offset is \$2D, so the new address is \$C12D.

Absolute Indexed with X: *a,x*

With *Absolute Indexed addressing with X*, the value stored in X is added to the specified address. In this example:

```
LDA $0001,X
```

The value in X is \$02. The sum of \$0001 and \$02 is \$0003; therefore, the value at loca-

tion \$0003, which is \$5A, is loaded into the Accumulator.

Absolute Indexed with Y: a,y

Absolute Indexed addressing with Y works the same as it does with X. In this example:

```
LDA $0001,Y
```

The value in Y is \$03. The sum of \$0001 and \$03 is \$0003, so the value at location \$0004, which is \$EF, is loaded into the Accumulator.

Zero Page Indexed with X: zp,x

Zero Page Indexed addressing is like Absolute Indexed addressing, but limited to the zero page. In the example:

```
LDA $01,X
```

X contains \$02, which is added to \$01 to present the zero page address \$03, and is equivalent to the absolute address \$0003. \$0003 contains \$5A, which is loaded into the Accumulator.

Zero Page Indexed with Y: zp,y

Zero Page Indexed addressing with Y works the same as it does with X. In the example:

```
LDA $01,Y
```

Y contains \$03, which is added to \$01 to present the zero page address \$04, and is equivalent to the absolute address \$0004. \$0004 contains \$EF, which is loaded into the Accumulator.

Absolute Indexed Indirect: (a,x)

With *Absolute Indexed Indirect addressing*, the value stored in X is added to the specified address. In this example:

```
JMP ($0001,X)
```

The value of X is \$02; so, the processor loads the address stored in locations \$0003 and \$0004 and jumps to that address. This mode of addressing is only supported by the JMP instruction.

Zero Page Indexed Indirect: (*zp,x*)

Zero Page Indexed Indirect addressing first adds X to the zero page address, such as in this example:

```
LDA ($15,X)
```

X contains \$02; so, the address \$0017 is generated. Zero Page Indexed Indirect addressing next goes to that address and the following address (\$0017 and \$0018), and loads the values contained there as an address. \$0017 contains \$10 and \$0018 contains \$D0; consequently, the address \$D010 is loaded. Finally, the processor goes to this address and loads the value it contains, \$33, into the Accumulator.

Zero Page Indirect Indexed with Y: (*zp,y*)

Zero Page Indirect Indexed addressing with Y begins by fetching the value at the zero page address and the zero page address plus one. In our example:

```
LDA ($2A),Y
```

The values at locations \$2A and \$2B would be fetched. \$2A contains \$35 and \$2B contains \$C2. This gives us the address \$C235. Next, the value of Y, which is \$03, is added to this address to create the address \$C238. Location \$C238 contains the value \$2F, which is loaded into the Accumulator.

Interacting with Memory

We used LDA as our sample instruction for memory addressing to load the Accumulator, but the same can be done with LDX and LDY to load the X and Y registers. Data can be transferred from the registers to memory using the store instructions STA, STX, and STY. A few of the more esoteric addressing modes are only supported by particular instructions. See Appendix C for details.

This program demonstrates some simple memory interactions, in which we store three variables in memory and then later retrieve them. Tables 6.4 and 6.5 show the contents of the registers before and after running the program, respectively.

Register	Value
A	\$73
X	\$E2
Y	\$90

Table 6.4 Register Contents Before Running

```
STA $05    ; Store $73 at $0005  
STX $1023  ; Store $E2 at $1023  
STY $00FF  ; Store $90 at $00FF  
  
LDA $1023  ; Load $E2 from $1023  
LDX $FF    ; Load $90 from $00FF  
LDY $0005  ; Load $73 from $0005
```

Register	Value
A	\$E2
X	\$90
Y	\$73

Table 6.5 Register Contents After Running

Printing Strings

The program in this section will introduce two new functions: the ability to assemble strings of characters and the ability to branch on a condition. ASCII characters can be stored in memory just like instructions, and by using the ASCII table in Appendix A we could manually enter the hexadecimal values into the Apple I's memory, just as we entered the instructions. The xa65 assembler provides a pseudo-opcode, .asc, which automatically converts a string of characters to hexadecimal. The statement:

```
.asc "HELLO WORLD"
```

will enter "HELLO WORLD" as ASCII, wherever the statement is placed.

The Branch on Result Zero instruction (BEQ) branches (jumps) to a new memory location whenever the Z (Zero) flag is high. BEQ uses relative addressing, so the value following the BEQ instruction is added to the current address (in the program counter) to get the new address.

The Z flag can be set using the Compare (CMP) instruction. CMP compares the value in the Accumulator to the value specified after CMP. If the value in the Accumulator is less than the value in memory, then N equals 1. If they are equal, then Z and C equal 1. If the value in the Accumulator is greater than the value in memory, then C equals 1. In the following example, \$AA is compared to the value in the Accumulator:

```
0000: CMP #$AA  
0002: BEQ $05  
0004: INX
```

If the Accumulator does not contain \$AA, then the Z flag is reset to 0 and BEQ does not cause a branch. If the Accumulator does contain \$AA, then Z is set to 1 and BEQ causes a branch to \$05 + program counter. The program counter is at \$0004 (the next instruction); therefore, the program branches to \$0009 and continues execution at that location.

CPX and CPY work the same as CMP, but use the X and Y registers instead of the Accumulator.

The program displayed later uses branching to print an array of characters. The characters begin at the label *string*, such that the instruction

```
LDA string
```

will load the first value at *string*, in this case 'H', represented by an ASCII \$48. The instruction:

```
LDA string + 1
```

will load the second value in the array, which is 'E', or \$45 in ASCII.

With this strategy, our program can use Absolute Indexed addressing to specify the offset (i.e. which character to load).

```
LDA string,X
```

The above instruction loads the value at *string* + X into the Accumulator. By starting X at zero and repeatedly incrementing it, every character in the array can be reached.

How do we know when to stop incrementing? The NULL character, \$00, can be appended to the end of the string:

```
.asc "HELLO WORLD", $00
```

After loading the character into the Accumulator, we can now compare it to the NULL character. If it is the NULL character, we know we have reached the end of the string and can branch to done:

```
CMP #$00      ; Compare char to NULL ($00)
BEQ done      ; If NULL break out of loop
```

The following program uses the above methods to print to screen a character array of undetermined length:

```
Echo = $FFEF
Monitor = $FF1F

Begin:
LDX #$00      ; Our loop expects X to start at 0

print:
LDA string,X  ; Load the next character
CMP #$00      ; Compare char to NULL ($00)
BEQ done      ; If NULL break out of loop
JSR Echo      ; If not NULL, print the character
INX          ; Increment X, so we can get at the next character
JMP print     ; Loop again

done:
JMP Monitor   ; Return to Monitor

string:
.asc "HELLO WORLD", $00  ; ASCII string, $00 is NULL
```

Assemble this program and examine it in HexEdit. You will see your string displayed in ASCII on the rightside of the window. Highlight the string and the corresponding ASCII values will be boxed (Figure 6.2). You can see NULL (\$00) following

immediately after the string. Preceding the character string is the rest of the program.

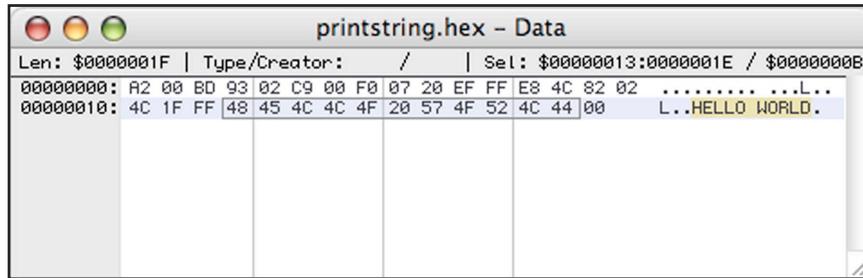


Figure 6.2 ASCII in HexEdit

String Subroutine

The objective in this section is to develop a subroutine that can be used by any program to print any string. This subroutine draws heavily from the program developed in the previous section. The primary difference is that instead of hard-coding the location of the string, we want it to be able to load characters from any address. Since the Accumulator, X, and Y registers are only 8-bit, passing a 16-bit address is rather tricky.

One possible workaround is to store the string's address in a pre-defined memory location, and then have the subroutine load it from that location. This can be done with Zero Page Indirect Indexed addressing. This addressing mode is only supported by the Y register; as a result, all our previous uses of the X register must be changed to Y.

If we load the string's address into \$00 and \$01, then we can access the characters in the string with the command:

```
LDY ($00),Y
```

The following program and subroutine demonstrates this method. This subroutine could be assembled separately and stored in ROM for use with any program. If you want to use this subroutine frequently but don't have the tools to program a ROM, you can load it into a high location of memory. Since memory is preserved through resets, you'll only lose the code when power is turned off.

```

Echo = $FFEF
Monitor = $FF1F

Begin:
  LDA #<welcome ; Load low byte of string's addr (indicated by
  <)
  STA $00 ; Save addr to $00

```

```

LDA #>welcome      ; Load high byte of string's addr (indicated by
>)
STA $01           ; Save addr to $01
JSR printsub      ; Print the string

LDA #<goodbye     ; Load low byte of string's addr
STA $00           ; Save addr to $00
LDA #>goodbye     ; Load high byte of string's addr
STA $01           ; Save addr to $01
JSR printsub      ; Print the string

JMP Monitor       ; Return to Monitor

/* Strings */

welcome:
.asc "Welcome to the Apple I", $0D, $00    ; $0D is carriage return
                                              ; $00 is NULL

goodbye:
.asc "That was a short demo.", $00

/*
Name:          PrintSub
Preconditions: Address of string to print is stored at $00,$01.
Postconditions: The string is printed to screen.
Destroys:      A, Y, Flags.
Description:   Prints the string at the address at $00,$01.
*/
printsub:
LDY #$00        ; Our loop expects Y to start at 0
print:
LDA ($00),Y      ; Load the next character
CMP #$00        ; Compare char to NULL ($00)
BEQ done        ; If NULL break out of loop
JSR Echo         ; If not NULL, print the character
INY             ; Increment X, so we can get at the next character
JMP print       ; Loop again
done:
RTS

```

This program exceeds the Monitor's maximum line length, so the simplest way to transfer it is to enter one line at a time. Each line in HexEdit contains 16 (\$10) bytes, so

you'll need to increment the memory address by \$10 with each line you copy. Entry will look like this:

```
280: A9 99 85 00 A9 02 85 01 20 C8 02 A9 B1 85 00 A9
290: 02 85 01 20 C8 02 4C 1F FF 57 65 6C 63 6F 6D 65
2A0: 20 74 6F 20 74 68 65 20 41 70 70 6C 65 20 49 0D
2B0: 00 54 68 61 74 20 77 61 73 20 61 20 73 68 6F 72
2C0: 74 20 64 65 6D 6F 2E 00 A0 00 B1 00 C9 00 F0 07
2D0: 20 EF FF C8 4C CA 02 60
```

Bit Representation

Our video circuitry can only print ASCII characters, but the processor instructions are geared primarily towards integer operations. The subroutine presented in this section will print the Accumulator's contents in its binary representation. For example, if \$05 is in the Accumulator, this subroutine will print "00000101".

There are many possible ways to approach writing this subroutine. One such way is to use the negative (N) flag. This flag is used in Two's Complement notation to indicate that the highest bit (bit 7) is a 1. By rotating the byte so that each value has its turn in the bit 7 position, N will be reset after each rotate. By branching whenever N is 1 to print an ASCII '1', and not branching whenever it is 0 to print an ASCII '0', we can print a 1 or 0 for each bit.

Three new instructions are used in this subroutine: CPY, ROL, ROR, BMI, and BNE.

Compare Y (CPY) is the equivalent of CMP for the Y register. The instruction:

```
CPY #$08
```

compares the value in the Y register to the value \$08. If they are equal, the Zero (Z) flag is set. Branch on Result Not Zero (BNE) branches whenever Z equals 0. We can use this combination of instructions to create a loop:

```
LDY #$00      ; Initialize Y to 0
loop:
    INY        ; Increment Y
    CPY #$08    ; Compare Y to $08
    BNE loop    ; If Y != $08 then loop
```

The above program initializes Y to \$00 and then loops, incrementing Y until it equals \$08.

Branch on Minus (BMI) branches when N equals 1. The N flag is set whenever bit 7 is high, by numerous instructions (see Appendix D), such as LDA and CMP. For example:

```
loop:
    LDA #$80
    BMI loop
```

will load the value \$80 (1000 0000_b) into the Accumulator and set N to 1 since bit 7 is high. Because N is high, BMI will branch to loop, and the program will run indefinitely.

Rotate Right (ROR) and Rotate Left (ROL) rotate the contents of the Accumulator or a memory address. The 'Carry' (C) is included in the rotation, such that if you did a ROL on the value:

0000 1111 C=1

the result would be:

0001 1111 C=0

It takes nine rotates to return the memory or Accumulator contents to their original values. Each time a rotate is done, the N flag in responds to the new bit 7.

The PrintBin subroutine below uses these new instructions to print the binary representation of whatever value is passed to it in the Accumulator:

```
Echo = $FFEF
Monitor = $FF1F

Begin:
    LDA #$00
    JSR printbin ; Print $00 in binary
    LDA #$FF
    JSR printbin ; Print $FF in binary
    LDA #$AA
    JSR printbin ; Print $AA in binary
    JMP Monitor

/*
Name:      PrintBin
Preconditions: Byte to print is in Accumulator.
Postconditions: The string is printed to screen.
Destroys:   A, X, Y, Flags.
Description: Prints the binary representation of the value in
A.
*/
printbin:
    LDY #$00      ; Initialize counter to 0
    ROR          ; ROR once so initial setup will mesh with the loop
```

```
TAX      ; Loop expects value to be in X register

begin:
TXA      ; Restore value to Accumulator
ROL      ; Rotate next bit into bit 7's location
BMI neg   ; If that bit is a 1, then branch to neg
TAX      ; The bit is 0; save byte to X
LDA #$30  ; Load ASCII '0' for printing
JSR Echo   ; Print 0
JMP continue ; Skip over code for printing a 1

neg:
TAX      ; Save byte to X
LDA #$31  ; Load ASCII '1' for printing
JSR Echo   ; Print 1

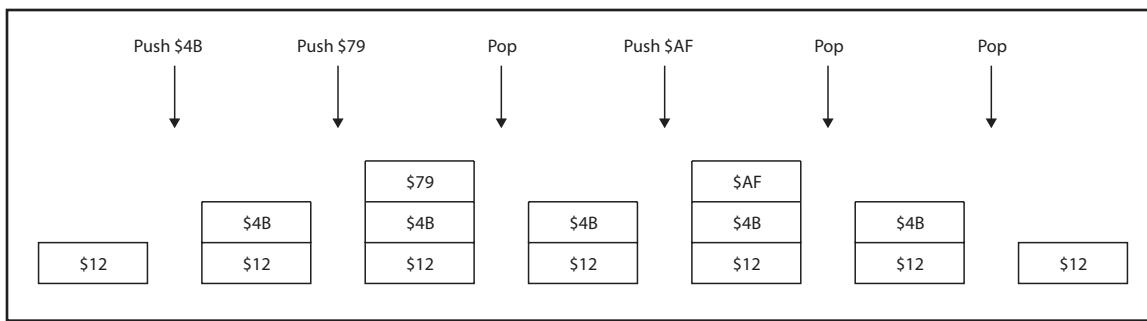
continue:
INY      ; Increment counter
CPY #$08  ; Compare counter to 8 (number of bits to print)
BNE begin  ; If counter != 8, then not all bits have been
printed,
           ; so loop again, otherwise continue

LDA #$0D  ; Load ASCII Carriage Return (CR)
JSR Echo   ; Print CR to screen
RTS      ; Exit subroutine
```

Using the Stack

A great feature of the Echo subroutine is that it doesn't destroy the contents of any registers—you can call Echo and pick up right where you left off without worrying about the contents of the Accumulator or the X and Y registers being any different. The PrintBin subroutine, by contrast, destroys the contents of all three. We can preserve the contents of these registers by saving them to the stack.

The *stack* is a data structure that only allows data to be added or removed at the top-most location. Data is either *pushed* onto the top of the stack or *popped* (sometimes called *pulled*) off the top of the stack. Figure 6.3 illustrates data being pushed and popped.

**Figure 6.3** Pushing and Popping

The 6502 uses the stack to store the return address when jumping to a subroutine (JSR). When returning from a subroutine (RTS), the address is popped from the stack and loaded into the Program Counter so the processor can continue operation where it left off. The stack is allocated the addresses between \$0100 and \$01FF. Overwrite data within this range and you risk corrupting the contents of the stack.

We can also use the stack for our own purposes, using the commands PHA, PLA, PHP, PLP, TSX, and TXS.

Push Accumulator on Stack (PHA) pushes the value in the Accumulator onto the top of the stack. *Pull Accumulator from Stack* (PLA) pops the top value off the stack and places it in the Accumulator. If we were to put the actions in Figure 6.3 into code, it would look like this:

```

LDA #$4B
PHA ; Push $4B
LDA #$79
PHA ; Push $79
PLA ; Pull $79
LDA #$AF
PHA ; Push $AF
PLA ; Pull $AF
PLA ; Pull $4B
  
```

In this example, every time a PLA instruction is issued, the contents of the Accumulator are overwritten. Normally, something would be done with the data after popping it.

The *stack pointer* is the address of the top of the stack. The value is stored in its own register, S. The address changes every time you push or pop a byte. If you want to manipulate this address, you can copy it into the X register using the *Transfer Stack Pointer to Index X* (TSX) instruction. You can copy your own stack pointer into S using the *Transfer Index X to Stack Register* (TSX) instruction.

Push Processor Status on Stack (PHP) and *Pull Processor Status from Stack* (PLP) are

used to preserve the contents' processor flags (e.g. N, V, etc.). This is particularly useful when you don't want a subroutine to destroy your flags. Using PHP before the subroutine is called and PLP after it returns will ensure that they are not lost. This is a concept we can use to preserve all our register contents when calling a subroutine.

Suppose we want to print the binary representation of every integer from 0 to 255. This is best done with a loop:

```
LDX #$00      ; Initialize counter to 0
JMP start     ; Skip over increment first time through the loop

next:
    INX          ; Increment the counter
start:
    TXA          ; printbin2 requires the value be in the accumulator
    JSR printbin2 ; print the value as binary
    CPX #$FF     ; If we've reached 255 ($FF), we're done
    BNE next     ; If not, loop again

JMP Monitor
```

There's just one problem – the X register is overwritten each time the PrintBin subroutine is called, destroying our counter. A possible fix for this is to push X onto the stack before calling the subroutine, then pop it after the subroutine returns, which could be done like this:

```
LDX #$00      ; Initialize counter to 0
JMP start     ; Skip over increment first time through the loop

next:
    INX          ; Increment the counter
start:
    TXA          ; printbin2 requires the value be in the accumulator
    PHA          ; Save X (already in the accumulator), before subroutine
    JSR printbin ; print the value as binary
    PLA          ; Retreive X from Stack, place it in the accumulator
    TAX          ; Copy X from accumulator to X register
    CPX #$FF     ; If we've reached 255 ($FF), we're done
    BNE next     ; If not, loop again

JMP Monitor
```

Another possibility is to modify the PrintBin subroutine so that it does not destroy the registers. This can be done by pushing all registers onto the stack at the start of the program and popping them at the end:

```

/*
Name:          PrintBin2
Preconditions: Byte to print is in Accumulator.
Postconditions: The string is printed to screen.
Destroys:      Flags
Description:   Prints the binary representation of the value in
A.
*/
printbin2:

STA temp      ; Save Accumulator for later
PHA           ; Push Accumulator
TXA
PHA           ; Push X (via A)
TYA           ; Push Y (via Y)
PHA
LDA temp      ; Restore the Accumulator

LDY #$00      ; Initialize counter to 0
ROR           ; ROR once so initial setup will mesh with the loop
TAX           ; Loop expects value to be in X register

JSR PrintBin  ; Now that registers are saved, call the ordinary
               ; PrintBin subroutine

PLA
TAY           ; Recover Y
PLA
TAX           ; Recover X
PLA           ; Recover Accumulator

RTS           ; Exit subroutine

temp:
.asc $00       ; Temporary storage location for Accumulator value

```

Which method is better? It depends on the application. If it doesn't matter that your subroutine destroys the registers, then there's no point in wasting time and space in the subroutine to preserve them. On the rare occasions you do want to preserve the registers, it would be more efficient to do so in the calling program. On the other hand, there are some subroutines such as Echo that get called very frequently. Imagine having to first push and then pop the contents of all the registers each time Echo is called. In a case such as this, it's much less frustrating and more space-efficient to put the stack operations inside the subroutine.

Bit Manipulation

The instructions *AND Memory with Accumulator* (AND), *OR Memory with Accumulator* (ORA), and *Exclusive-OR Memory with Accumulator* (EOR) are used to perform bit-by-bit operations on data. These are the same operations discussed in Chapter 2, but while in that chapter they are applied to single bits, here they are applied to bytes on a bit-by-bit basis. To AND \$AC with \$E6, for example, produces \$A5:

```
1010 1100 = $AC
AND 1110 0110 = $E6
      1010 0101 = $A5
```

In an AND operation, each resulting bit is high only if both bits in that column are high. This is very useful for masking. Suppose you want to isolate the lower four bits from the upper—that is, you want to change \$B6 into just \$06. Use \$0F as a mask:

```
1011 0110 = $B6
AND 0000 1111 = $0F
      0000 0110 = $06
```

To turn \$B6 into just \$0B, use \$F0 as a mask and then ROR four times.

Suppose that, by some terrible calamity, you come upon a string of text that contains lower-case characters. Masking can be used to make these upper-case. Upper-case ASCII characters have hex values in the range of \$41 (0100 0001b) for ‘A’ to \$5A (0101 1010b) for ‘Z’. Lower-case characters are in the range \$61 (0110 0001b) for ‘a’ to \$7A (0111 1010b) for ‘z’. Note that the values for ‘A’ and ‘a’ are identical, except for bit 5. The same applies to ‘Z’ and ‘z’, as well as the other letters. To change a character from lower- to upper-case, you only need to change bit 5 from 1 to 0.

This can be done programmatically by masking the ASCII value with 1101 1111b. This passes through every bit except bit 5, which is forced to a 0. Here are some examples:

```
0110 0001 = $61 = 'a'
AND 1101 1111 = $DF
      0100 0001 = $41 = 'A'
```

```
0100 0001 = $41 = 'A'
AND 1101 1111 = $DF
      0100 0001 = $41 = 'A'
```

```
0111 0011 = $73 = 's'
AND 1101 1111 = $DF
      0101 0011 = $53 = 'S'
```

The following program uses a mask with \$DF to convert all letters in a string from lower- to upper-case; it will leave upper-case letters untouched, but will affect a few non-letter characters above \$5A:

```

Echo = $FFEF
Monitor = $FF1F

LDY #$00      ; Initialize counter to 0

caps:
    LDA string,Y ; Load in first character
    CMP #$00      ; Is this character null?
    BEQ done      ; If so, we're done converting
    CMP #$41      ; Is this character a letter?
    BMI skip      ; If not, skip it
    AND #$DF      ; Mask with $DF to convert to uppercase
    STA string,Y ; Overwrite original character with the new
skip:
    INY          ; Increment the counter
    JMP caps      ; Loop to the next character

done:
    LDA #<string ; Save memory address of string for access by
PrintSub
    STA $00
    LDA #>string
    STA $01
    JSR printsub ; Print the new uppercase string

    JMP Monitor

string:
    .asc "lowercase string, MOSTLY", $00

/* Append PrintSub subroutine here */

```

While AND is useful for clearing bits to 0, ORA can be used to set them to 1. Suppose that contrary to the last example, you now want to convert all letters from upper- to lower-case. Bit 5 would then need to be set high. This can be done with an ORA operation:

0100 0001 = \$41 = 'A'
ORA 0010 0000 = \$20
0110 0001 = \$61 = 'a'

```

0110 0001 = $61 = 'a'
ORA 0010 0000 = $20
0110 0001 = $61 = 'a'

0101 0011 = $53 = 'S'
ORA 0010 0000 = $20
0111 0011 = $73 = 's'

```

To modify the above program so that it converts upper- to lower-case instead of vice-versa, it is only necessary to change one line:

```
AND #$DF ; Mask with $DF to convert to uppercase
```

so that it reads

```
ORA #$20 ; Set bit 5 to convert to lowercase
```

Finally, the Exclusive-OR (EOR) command is useful for flipping bits:

```

0010 1010 = $2A
EOR 1111 1111 = $FF
1101 0101 = $D5

1111 0000 = $F0
EOR 1111 1111 = $FF
0000 1111 = $0F

```

One particularly interesting feature of EOR is that it can be used to swap two values without using a temporary variable. Under normal circumstances, if we were trying to swap two values stored in memory, the code would look something like this:

```

LDA $00 ; Load both values into registers
LDX $01
STA $01 ; Store both values to memory
STX $00

```

But what if only one register is available? The following code will swap the contents of \$00 and \$01 using only the Accumulator:

```

LDA $00 ; Load v1
EOR $01 ; v1 xor v2 = v1
STA $00 ; Store in v1
EOR $01 ; v1 xor v2 = v2
STA $01 ; Store in v2
EOR $00 ; v1 xor v2 = v1
STA $00 ; Store in v1

```

AND, ORA and EOR are powerful tools for bit manipulation. Most of the examples

in this chapter deal with ASCII manipulation where they are of limited usefulness, but for other applications, such as interacting with peripherals, they are invaluable.

Math

The 6502 contains instructions for addition and subtraction, and can work in both binary and decimal modes. The examples in this section will use binary mode, but you can find many examples of decimal mode and Binary Coded Decimal (BCD) on the Internet.

Add Memory to Accumulator with Carry (ADC) is the instruction for addition. The *carry bit* is a flag in the Processor Status Register that indicates there is a bit to carry beyond the 8th place. Here are a few examples of simple arithmetic, with carry first reset to 0:

$$\begin{array}{r} 0000\ 0001 = \$01 \\ + \underline{0000\ 0001} = \$01 \\ 0000\ 0010 = \$02 \end{array}$$

$$\begin{array}{r} 0000\ 0001 = \$01 \\ + \underline{0000\ 0011} = \$03 \\ 0000\ 0100 = \$04 \end{array}$$

$$\begin{array}{r} 0000\ 0011 = \$03 \\ + \underline{0000\ 0011} = \$03 \\ 0000\ 0110 = \$06 \end{array}$$

$$\begin{array}{r} 0000\ 1111 = \$0F \\ + \underline{0000\ 1111} = \$0F \\ 0001\ 1110 = \$1E \end{array}$$

$$\begin{array}{r} 0110\ 1110 = \$6E \\ + \underline{0001\ 1111} = \$1F \\ 1000\ 1101 = \$8D \end{array}$$

$$\begin{array}{r} 1001\ 1101 = \$9D \\ + \underline{0110\ 0001} = \$61 \\ 1111\ 1110 = \$FE \end{array}$$

This last example came dangerously close to overflowing the available 8 bits. The largest value a single byte can hold is 255. To calculate values larger than this, the carry flag is needed. The carry flag allows us to continue our operation into the next byte, with a carry, so that no part of the value is lost. For example:

$$\begin{array}{r}
 C \quad 1 \\
 \begin{array}{r} 0000 \ 0000 = \$00 \\ + \underline{0000 \ 0000 = \$00} \\ \hline 0000 \ 0001 = \$01 \end{array} \quad \begin{array}{r} 1000 \ 0100 = \$84 \\ + \underline{1001 \ 0010 = \$92} \\ \hline C=1 \quad 0001 \ 0110 = \$16 \end{array}
 \end{array}$$

The sum of \$84 and \$92 exceeds \$FF and our byte overflows. This overflow sets the carry flag and we can continue our addition in the next byte with a sum of \$01. Putting the bytes together, we get the grand sum of \$0116.

By using two bytes we can address values up to 65,535. Suppose we were to add \$42A5 to \$15D2. The same layout can be used:

$$\begin{array}{r}
 C \quad 1 \\
 \begin{array}{r} 0100 \ 0010 = \$42 \\ + \underline{0001 \ 0101 = \$15} \\ \hline 0101 \ 1000 = \$58 \end{array} \quad \begin{array}{r} 1010 \ 0101 = \$A5 \\ + \underline{1101 \ 0010 = \$D2} \\ \hline C=1 \quad 0111 \ 0111 = \$77 \end{array}
 \end{array}$$

The method is equally effective when no carry flag is necessary, such as in the case of adding \$3E1F to \$1A73:

$$\begin{array}{r}
 C \quad 0 \\
 \begin{array}{r} 0011 \ 1110 = \$3E \\ + \underline{0001 \ 1010 = \$1A} \\ \hline 0101 \ 1000 = \$58 \end{array} \quad \begin{array}{r} 0001 \ 1111 = \$1F \\ + \underline{0111 \ 0011 = \$73} \\ \hline C=0 \quad 1001 \ 0010 = \$92 \end{array}
 \end{array}$$

The carry flag can be manually set or cleared using the *Set Carry Flag* (SEC) and *Clear Carry Flag* (CLC) instructions, respectively. Many instructions set the carry flag; subsequently, it is important to clear the flag before performing addition. Below is the code for adding \$61 to \$9D:

```

LDA #$9D      ; $9D + ...
CLC          ; Make sure Carry is not already set
ADC #$61      ; $9D + $61
JSR printbin  ; Print the result in binary

```

Sixteen-bit addition is only slightly more complex. Suppose we want to add the value in the locations \$00,\$01 to \$02,\$03 and store the result in \$04,\$05. First the low bits are added and then the high bits, with carry:

```

CLC          ; Make sure Carry is not already set
LDA $00
ADC $02      ; Add the low bytes
STA $04      ; And save sum to memory
LDA $01
ADC $03      ; Add the high bytes (note that Carry was NOT
cleared)
STA $05      ; And save sum to memory

```

```

LDA $04
JSR printbin ; Print low byte
LDA $05
JSR printbin ; Print high byte
JMP Monitor

```

Subtraction is handled just like addition, except it expects the carry flag to be high when it starts. Before issuing the first SBC instruction, it's always necessary to set the carry flag using SEC.

You may have noticed by now that there is no instruction for multiplication. There are many techniques for multiplying two numbers. The simplest is to place the first value in the Accumulator and the second in Y, then decrement Y each time you add the value in the Accumulator to its original value. In this manner, 5×4 becomes $5+5+5+5$.

Since we're working in base-2, multiplying or dividing by powers of two stands out as particularly easy. To multiply by two, simply shift once to the left; to multiply by four, shift twice to the left, and so on. The same technique applies to division. Shift once to the right to divide by two, twice to the right to divide by four, and so on. Here are some examples:

```

0110 1100 = $6C
x     2      (shift left 1 bit)
11011000 = $D8

0000 0110 = $06
x     16     (shift left 4 bits)
0110 0000 = $60

1110 1000 = $E8
,     8      (shift right 3 bits)
0001 1101 = $1D

```

These multiplication and division operations can be performed with the *Accumulator Shift Left* (ASL) and *Logical Shift Right* (LSR) instructions. ASL shifts the value in the Accumulator (or memory) left by one bit. Bit 7 is shifted into the carry flag. A zero is shifted into bit 0. LSR shifts the value in the Accumulator (or memory) right by one bit. A zero is shifted into bit 7 and bit 0 is shifted into the carry flag. The following program multiplies the value in the Accumulator by 8:

```

ASL
ASL
ASL

```

This program divides the value in the Accumulator by 4:

```

LSR
LSR

```

Obviously, this method has its limitations. It is easy when multiplying to exceed 255 and when dividing to generate a fraction. Checking the carry flag after each shift will alert you if this occurs.

It is possible to use ASL and LSR to multiply by values that are not powers of two. To multiply by 10, for example, multiply by two, then multiply by eight, and add the results together.

Summary

In this chapter I've attempted to give a basic overview of the instructions for programming in assembly and their practical applications. Some instructions, which are similar to those already covered here, have been passed over. For a complete list of instructions and their uses, see appendices B, C, and D. For a more thorough look at programming in assembly, visit the Apple I Owners Club's web site on www.applefritter.com, or Mike Naberezny's web site at www.6502.org. There, you'll find a wealth of data sheets, tutorials, and source code.

Chapter 7

Understanding the Apple I

In this Chapter

- Overview
- The Data Bus
- The Clock
- The Processor
- Memory
- I/O with the 6821
- Keyboard Input
- Video Output

Overview

The design of the Apple I is as simple and straightforward as any computer you'll find. Look at the block diagram in Figure 7.1. At the center of the design is the processor, the brains of the operation. All data passes through it, and it controls which data that is, and when. The ROM contains the Monitor, which, as you learned in the previous chapter, is essentially a very simple operating system. When the computer is first turned on, the processor immediately goes to the ROM and runs the Monitor program. The RAM is used to store user programs and data. You'll note that there is no mass storage device (hard drive, floppy drive, etc.) in this diagram or in the Apple I, though it would be possible to add one.

Once the computer boots, it outputs a command line prompt (a slash) to the screen. The processor sends the character to the Input/Output (I/O) chip. The I/O chip waits until the video section is ready for the next character, then sends it off. The video section converts the character from digital data to an analog signal suitable for a video monitor. Input comes from the keyboard, where it is stored in the I/O chip and then transferred to the processor. It's very important to understand that the processor does all the work and directs all operations.

In this chapter, we'll discuss the workings of the processor, RAM, ROM, and I/O. The video section will be addressed briefly, but we will not cover the complexities of adapting the data for output to a CRT. Our emphasis will be on the digital workings of the computer.

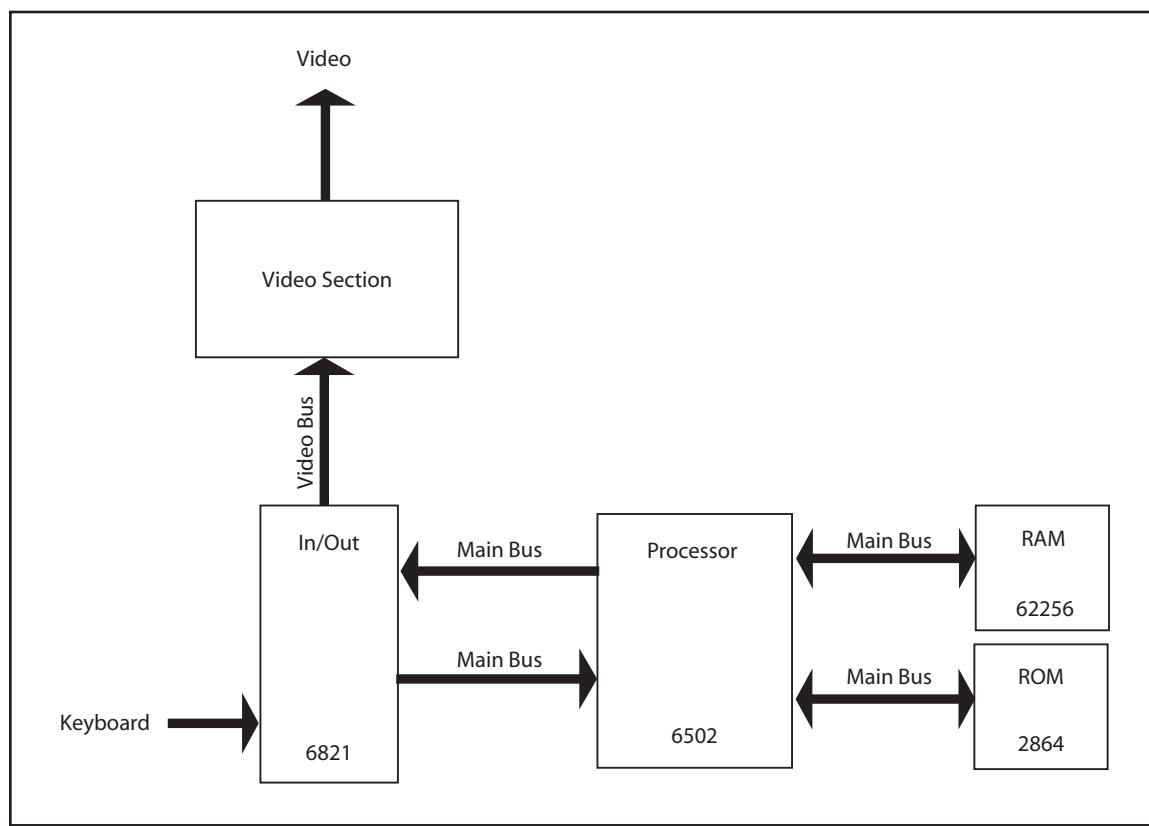


Figure 7.1 Apple I Block Diagram

The Data Bus

The Apple I's main components are all attached to the 8-bit data bus (Figure 7.2). All the data the processor sends or receives travels over this bus. The individual components cannot communicate with one another directly—all data must pass through the processor.

Let's say we have a character (we'll make it 'J') in RAM and we want to send it to the display. An ASCII character is a single byte, so it fits perfectly on our 8-bit data bus. Our first step is to get the data out of RAM and load it into the processor. We know the location, or *address*, of the character, so we put that location on the address bus. If the address were \$2100, for example, the instruction would be LDA \$2100.

Each device on the bus sees that address. The ROM and I/O chips look at the address and see that it's not for them. The RAM chip recognizes the address as its own and knows to which location in its memory that the address refers. The RAM chip looks at the Read/Write line to see whether the processor wants to read a byte from this address, or write a byte to it (that is, whether to load the 'J' into the processor or to overwrite the 'J' with some other letter). The Read line is high, so the RAM chip places the 'J' on the data bus, where it's then loaded into the processor.

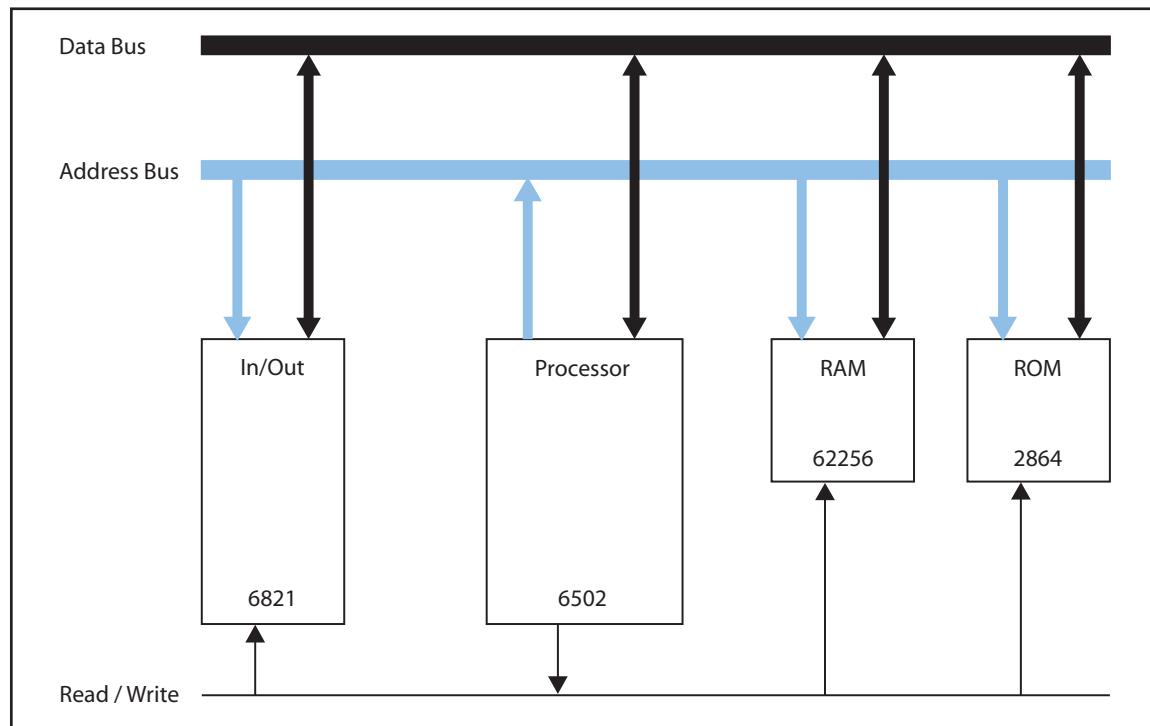


Figure 7.2 Data Flow

Next, we want to send the data from the processor to the video output via the I/O chip. A specific address on the I/O chip is associated with sending data to video-out. The processor puts that address on the address bus. The I/O chip recognizes that address as its own, the one intended for video-out data. The other chips don't recognize the address and thus ignore it. The Read line goes low (which means write), so the In/Out chip knows it will be receiving data and the character 'J' is placed on the data bus. At the clock pulse, the I/O chip reads the data off the data bus. Then the I/O chip sends it off to the video circuitry.

The Address Bus

The address bus is 16 bits wide, which means we have enough addresses for 65,536 (64 KB, or 2^{16}) unique locations. Usually, these locations are memory addresses (RAM), but they could range anywhere from light emitting diodes (LEDs) to sensors or I/O chips. Let's say we have a 32-kilobyte RAM chip. Half of our available addresses fit within 32 KB, so let's make this chip occupy the lower half of the address space, addresses 0 through 32,767. Whenever the address is 32,767 or less, we want to enable the chip. How can we check for numbers in this range?

One very complex possibility is to check each possible combination. For example, to check for 1, which is within our range, we would write:

```
Enabled for 1 = A15' • A14' • A13' • A12' • A11' • A10' • A9' •  
A8' • A7' • A6' • A5' • A4' • A3' • A2' • A1' • A0
```

Or to check for 14,287, which is also within our desired range, we would write:

```
Enabled for 14287 = A15' • A14' • A13 • A12 • A11' • A10 • A9 •  
A8 • A7 • A6 • A5' • A4' • A3 • A2 • A1 • A0
```

And we would end up doing this 32,767 times, once for each address. This approach would certainly work, but the amount of work and time needed to complete the task would be so enormous that it would be terribly impractical. Let's try to find a simpler method by examining the relevance of each bit. The lowest bit, A0, alternates high and low with each number, so it's no good for determining whether a number is larger or smaller than 32 KB. The same applies to bit A1, which alternates high and low with every second number. Let's look at the range of numbers we have to cover (Table 7.1):

Decimal	Binary
0	0000 0000 0000 0000b
32,767	0111 1111 1111 1111b

Table 7.1 32 KB Address Range

Notice from this table is that although bits A14 through A0 change, bit A15 is always 0. Not only that, but our range includes every number where A15 is 0. If A15 is low, we know the address is within our 0 to 32 KB range. Our circuit, shown in Figure 7.3, is now trivial—whenever A15 is low, enable the chip.

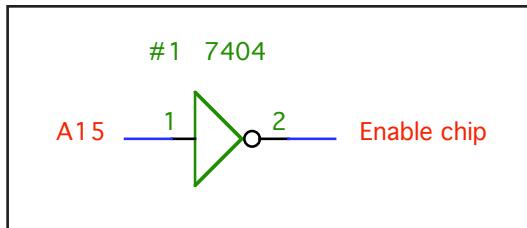


Figure 7.3 32 KB Enable

You might be thinking that this example was contrived to work out easily, and it certainly was, but that doesn't make it unrealistic. Any memory chip you find is going to be based on a power of two, and when you're designing a computer, it certainly behooves you to use start and end points that are easily defined in logic.

Let's take a harder example. Suppose we have 16 kilobytes of RAM and we want to locate this memory at addresses 32 KB through 48 KB. Our table would then look like Table 7.2:

Decimal	Binary
32,768	1000 0000 0000 0000b
49,151	1011 1111 1111 1111b

Table 7.2 16 KB Address Range

Bits A0 through A13 will vary, so we can disregard those for now. A14 must be 0, because if it were 1, the value would be larger than 48 KB. A15 must be 1, because if it were 0, the value would be smaller than 32 KB. Thus, we want to enable our chip whenever A15 is high and A14 is low (Figure 7.4).

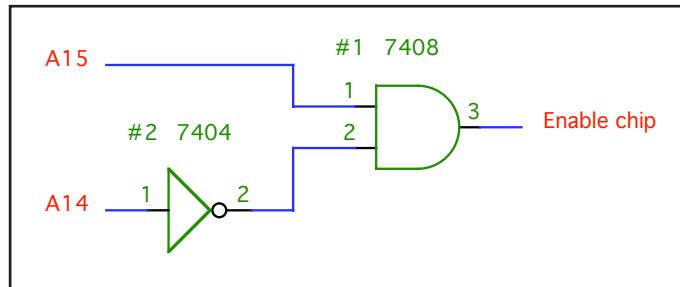


Figure 7.4 16 KB Enable

Now that we've enabled the chip we want to use, we need to identify the location on that chip. This is where the rest of the address lines on the address bus come into play. Bits A0 through A13 can express our entire range of locations. These lines are connected directly to the address lines on the chip so that it can find the internal location of the requested data.

Bidirectional Buses

The data bus is *bidirectional*, meaning that data can be both sent and received over it. The Read/Write line is used to determine which direction the data is going. The address bus, by contrast, is *unidirectional*. Only the processor can send data to it. The rest of the chips (such as RAM, ROM, and others) can only read what's on the address bus.

The Clock

Everything happens at the clock pulse. New values are waiting at the register inputs. When the clock pulses, the new values are loaded into the registers over the course of a couple nanoseconds, then become present at the output of the registers. The output signal propagates through the circuit, each logic gate taking a few more nanoseconds. Eventually, all is updated and the lines stabilize. At this point, the clock pulses and the cycle repeats. Figure 7.5 shows a 1MHz TTL crystal oscillator, which we will use for our clock.



Figure 7.5 1MHz TTL Crystal Oscillator

Figure 7.6 shows a timing diagram for the clock pulse. At each rising edge, indicated by an arrow, the circuit updates. The Apple I uses a 1MHz clock. Compare that to the multi-gigahertz clocks in today's computers, and it's glacially slow. What would happen if we put a 1GHz clock in the Apple I? At 1GHz, the circuit would not have time to fully propagate after each clock pulse, leaving the circuit partially updated and in an uncertain state.

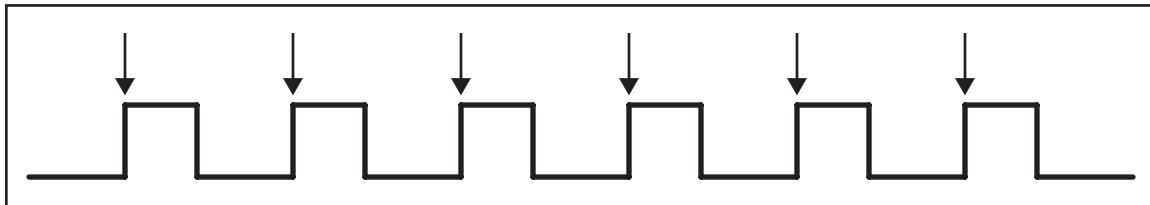


Figure 7.6 Timing Diagram

Figure 7.7 shows the pinout for a 1MHz TTL crystal oscillator. Pin 1 is not used. Pins 7 and 14 are ground and power, respectively. Pin 8 is our clock output. The output is low for 500KHz, then high for 500KHz. Our circuitry triggers on the rising edge.

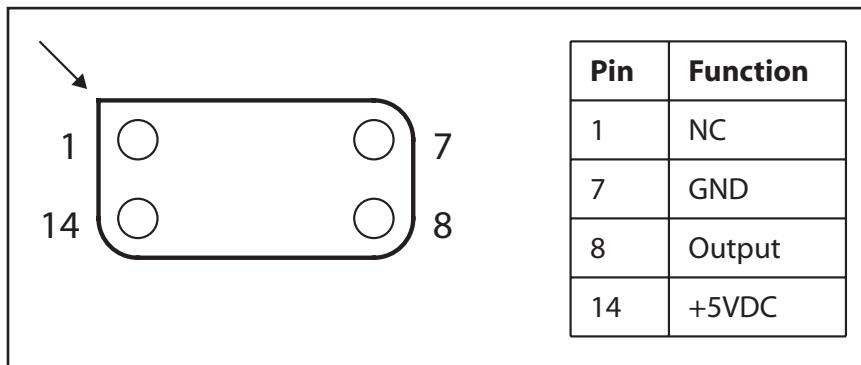


Figure 7.7 1MHz TTL Crystal Oscillator Pinout

The Processor

Fully understanding the operations of the 6502 processor (Figure 7.8) sounds daunting, and it certainly could be. However, we're going to concentrate on understanding the processor from the user's perspective, which simplifies our cause. You won't learn how to build a processor from scratch (that's another book in itself), but you will learn how to design basic circuits that interact with the processor. In Chapter 6, we discussed how to write programs in machine language that can be directly understood by the 6502. This chapter will go into more depth, looking at what happens when you run those programs.

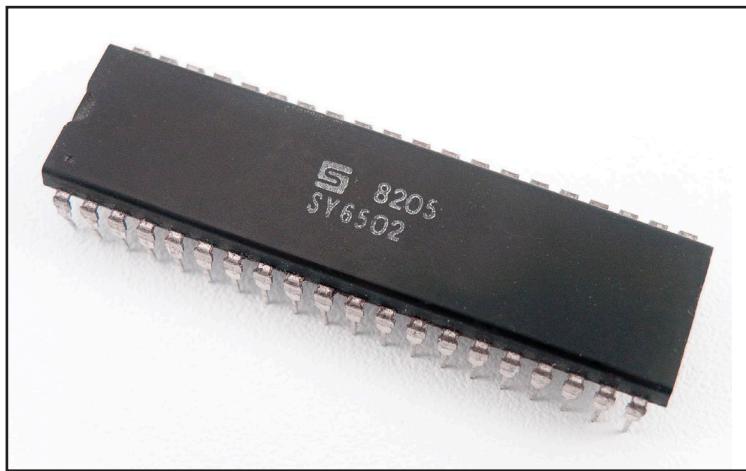


Figure 7.8 6502 Microprocessor

Pins and Descriptions

The 6502 (Figure 7.9) comes in a 40-pin dual in-line package (DIP). The function of each pin is described later in this chapter. If you're using the slightly newer and more advanced 65c02 chip, you'll have a few extra features available. These features aren't used by the Apple I, and the 65c02 is backwards compatible with the 6502, so you can safely think of your 65c02 as an ordinary 6502 processor.

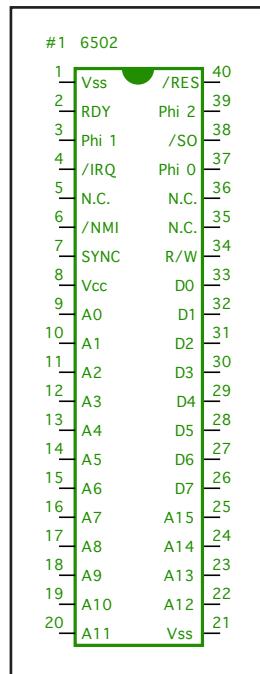


Figure 7.9 6502 Pinout

Address Bus (A0–A15)

The address bus is a 16-bit bus capable of addressing 64 kilobytes. Those bytes could be locations in memory or on input/output devices.

Clock (ϕ_0 , ϕ_1 , ϕ_2)

The symbol ϕ_0 (or Phi 0) denotes the oscillator input. Connect the output of your 1MHz clock to this line. ϕ_1 and ϕ_2 are outputs that are offset from this clock line. ϕ_1 is not used by the Apple I. ϕ_2 is used when writing to the 6821 PIA and the RAM.

Data Bus (D0–D8)

The eight-bit data bus allows bidirectional communication between the processor and memory or other peripheral devices.

Interrupt Request (IRQ)

This line is not used by the Apple I. A pulse on the interrupt line tells the processor to stop what it's doing and to process data from another location.

No Connection (NC)

There is no signal on this line; as a result, it can be left disconnected. If you're using the later 65c02 instead of the original 6502, you'll find some of the NC pins have been replaced with actual signals. The 65c02 is backward-compatible, so you can just leave these lines disconnected as you would on the 6502.

Non-Maskable Interrupt (NMI)

This line is not used by the Apple I. Whereas a normal interrupt (IRQ) can be blocked by software, a non-maskable interrupt will always succeed in getting through and interrupting the system.

Ready (RDY)

This line is not used by the Apple I. The Ready line allows you to halt or single-step the processor. This can be especially handy if you're trying to troubleshoot a design and want to see what the system is doing, one step at a time.

Reset (RES)

When power is first applied to the processor, it's in an unknown state. Pressing the Reset button tells it to load the program at the address contained at \$FFFC and \$FFFD and to start executing.

Read/Write (R/W)

The Read/Write line is used in conjunction with the data bus. When Read is high, the processor is reading data from the data bus. When Read is low, the processor is writing data to the data bus.

Set Overflow Flag (SO)

This line is not used by the Apple I. Its function is to manually set the overflow flag in the processor. The overflow flag is used to indicate that a number is larger than can be contained.

Sync

This line is not used by the Apple I. Sync can be used in combination with RDY to single-step the processor.

Voltage Common Collector (Vcc)

Vcc is the positive voltage line. It provides power for the chip and connects to the five-volt signal.

Voltage Source (Vss)

Vss is the ground line. The 6502 has two Vss lines, probably for additional current-handling.

With this knowledge, we can draw the schematic for the processor (Figure 7.10). The address lines go to the address bus, and the data lines to the data bus. The Read/Write, Phi 0, and Phi 2 pins are all wired to their respective lines. /SO is disabled by being tied to high. RDY, /IRQ, and /NMI are tied to high with current-limiting resistors. Reset is connected to the reset circuitry, which is high until the switch is depressed.

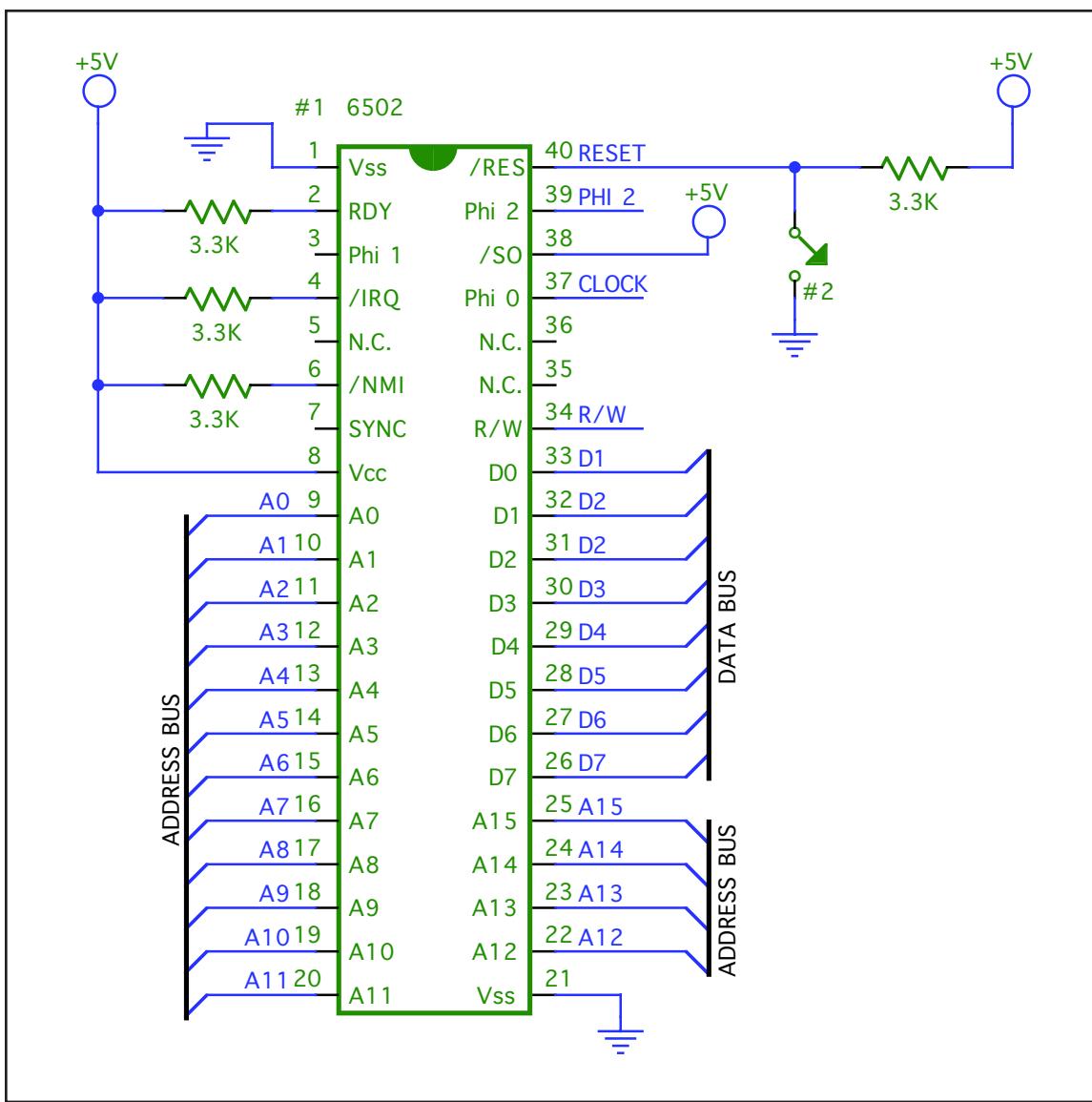


Figure 7.10 Processor Schematic

Registers

Registers, discussed in Chapter 5, are a fundamental part of the microprocessor. Each data register in the 6502 has a capacity of eight bits, and can therefore store an entire byte of data. The Program Counter (PC) can hold 16 bits. Registers allow the processor easy access to the data it needs to operate. Figure 7.11 shows the five registers most important to us.

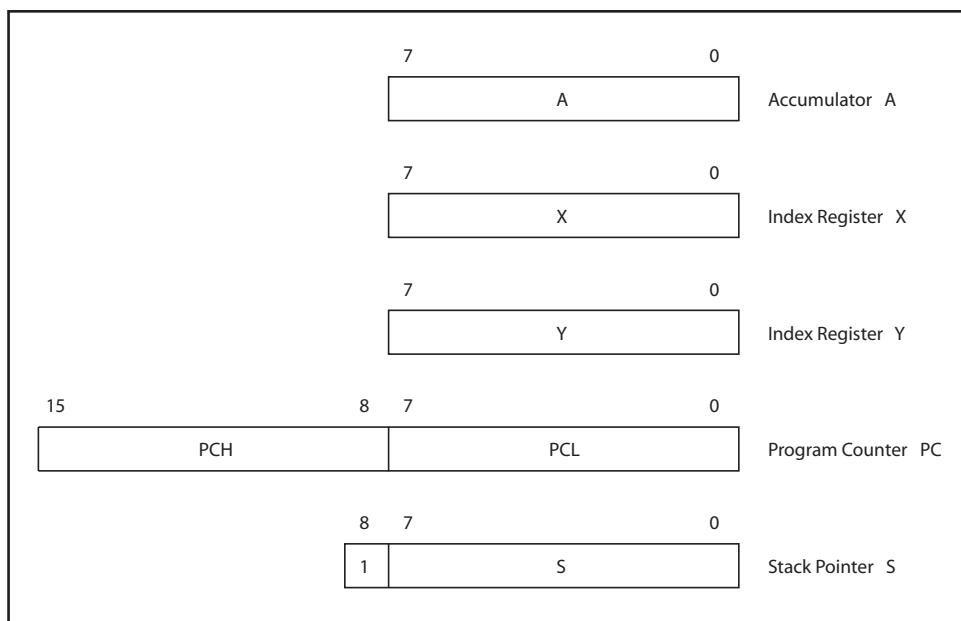


Figure 7.11 6502 Registers

The Accumulator

The Accumulator is used for mathematical and logical operations. If you have a number on which you want to perform addition or subtraction, you first load it into the Accumulator.

Index Registers X and Y

X and Y are generic registers. If you have a byte of data that you just want to keep nearby or are moving from one peripheral to another, the X and Y registers are convenient nearby storage places. It's also very easy to move data between various registers, so if you have a byte in X that you want to do a mathematical operation upon, you can easily transfer it to the Accumulator.

Program Counter (PC)

The PC is a 16-bit register used for keeping track of the address of the current instruction. It is incremented each time an instruction or data is fetched from memory. The processor looks at the program counter to see where it should fetch its next instruction.

Stack Pointer

The stack pointer points to (gives the address of) the current top of the stack. This ad-

address will change every time a byte is added or removed from the stack. The function of the stack is described later in this chapter.

Processor Status Register

The processor status register (Figure 7.12) is a collection of individual flags that serve as indicators for various instructions. For example, if you perform a subtraction and the answer is negative, the negative flag will be set. Then the next instruction can check that flag, see that the answer was negative, and report back to your program.

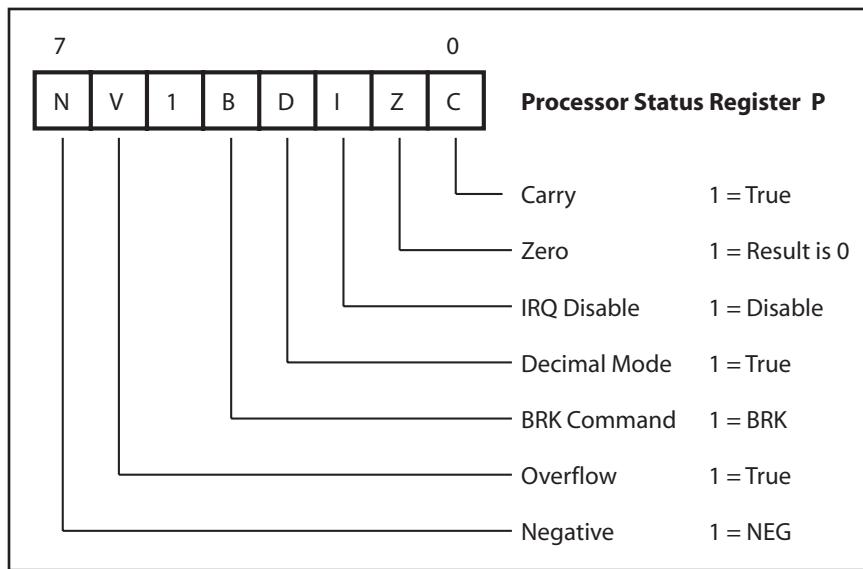


Figure 7.12 Processor Status Register

The Arithmetic and Logic Unit

The Arithmetic and Logic Unit (ALU) is at the heart of the microprocessor. The ALU can be thought of as a processing machine. Put the numbers you want to work on at the entry, choose the function you want, and let it run. The answer comes out the other end and is stored in the Accumulator.

The Stack

The stack lets us save data for future use. Access to data on the stack is limited. Only the topmost value can be read. Data is either “pushed” onto the stack or “popped” off of it. The stack pointer remembers the current location of the top of the stack. When data is

pushed, the stack grows and the stack pointer is incremented. When data is popped, the stack shrinks and the stack pointer is decremented. Chapter 6 discusses how to program with the stack.

The stack uses memory from \$0100 to \$01FF (figure 7.13). \$01FF is the base address. As data is pushed onto the stack, the stack grows towards \$0100. The size of the stack cannot exceed the 255 bytes allocated to it.

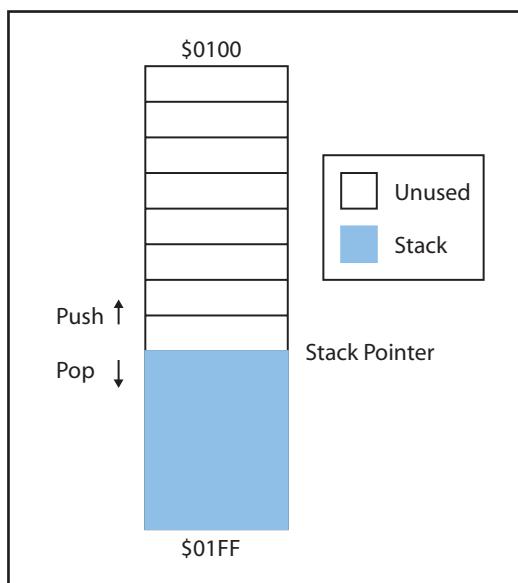


Figure 7.13 6502 Stack

Memory

Let's take a look at the way memory is used in the Apple I.

Where is It?

The Apple I has 64 KB of addressable space. The Monitor ROM occupies 256 bytes. The original Apple I also had 8 KB of RAM. Of the 64 K addresses (\$0000 through \$FFFF), where should this memory be located? Deciding on a location for the ROM is easy. The 6502 microprocessor requires that a memory address be stored at addresses \$FFFE and \$FFFF (the address is 16-bit, so it takes two bytes). When the 6502 is reset, it immediately loads the address at \$FFEF and \$FFFF into its program counter and begins executing that address. Thus, you need to have a ROM at \$FFFE-\$FFFF, and if you need to have it there, you might as well keep the ROM in a solid block. The Monitor ROM,

therefore, starts at \$FF00 and ends at \$FFFF.

The RAM, by contrast, is divided into two parts. Four kilobytes are for system and user access. These 4 KB are located at \$000-\$0FFF. The other 4 KB, which are intended for Apple's Integer BASIC, are located at \$E000-\$EFFF. You'll note that \$E000-\$EFFF are RAM, not ROM. Every time the original Apple I was powered up, the code for BASIC would have to be either loaded from cassette or typed in by hand. If the user did not need BASIC, he or she could refrain from loading it and just use that memory as generic memory space.

Figure 7.14 shows the memory map for the Apple I, as designed by Steve Wozniak.

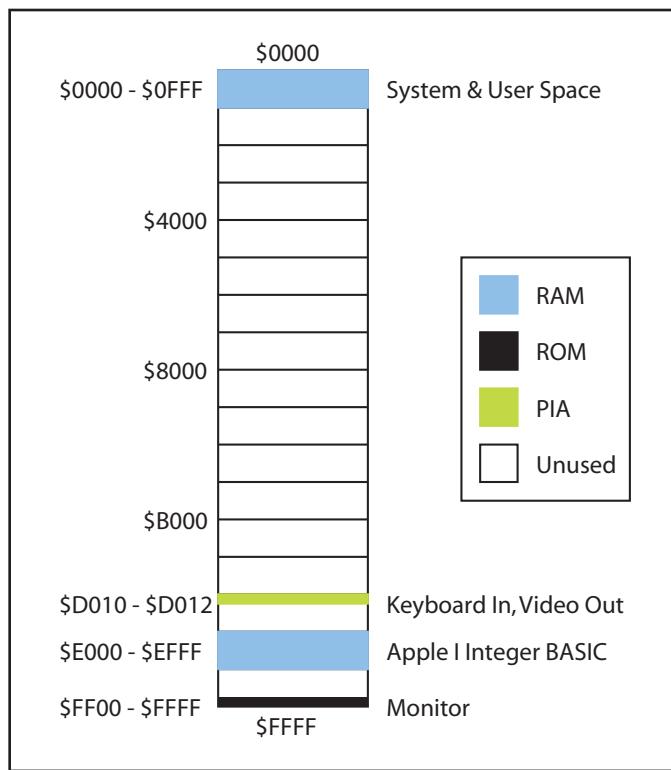


Figure 7.14 Apple I Memory Map

Note that the vast majority of this address space is unused. If the user wanted, he or she could expand the RAM in some of these blocks or use it for other peripheral devices. Joe Torzewski, who started the Apple I Owners Club in 1977, upgraded his Apple I to 16 KB. It would be possible to fill the entire unused memory with additional RAM.

Why did Apple include so little memory? Back in 1976, that 8 KB of RAM was prohibitively expensive. Today, by contrast, the price of RAM has plummeted, and the difference in cost between 8 KB and a hundred times that is negligible. Unless you're going strictly for historical accuracy, you'll probably want to add more RAM to your replica design. Figure 7.15 shows the memory map for Briel Computers' Replica I.

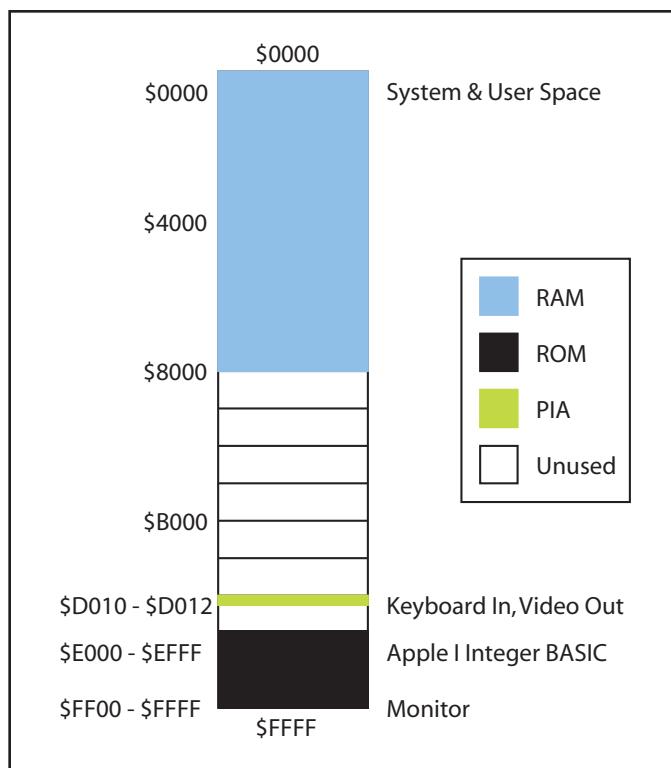


Figure 7.15 Replica I Memory Map

The first thing you'll notice about Briel's design is all the extra RAM. The first 32 KB (\$0000–\$8000) are all dedicated to system and user space. This is memory you're free to play around with. This is eight times the amount of system and user space in the original design, and that kind of expansion can make a big difference. Next, look at locations \$E000 to \$EFFF. Instead of being in RAM, Integer BASIC is now in ROM. This means BASIC is permanently stored on the Apple I and is immediately available when the system is turned on.

If you're a hardy assembly language programmer who scorns BASIC, this may be superfluous, but for everybody else, it saves a tremendous amount of time and frustration. Finally, note that addresses \$F000 to \$FE99 are mapped to ROM, but aren't labeled. If there's a program you use very frequently, you could potentially put it in ROM here, but right now, the space is wasted. Dedicating the entire 4KB block, \$F000–\$FFFF, to ROM makes addressing easier and decreases the number of chips we need to place on the circuit board.

Implementing 8 KB RAM

The original Apple I used 16 MK4096 ICs to get 8 KB RAM. We can do the same with a single Cypress CY6264 (Figure 7.16). Let's briefly take a look at the function of the more interesting pins.

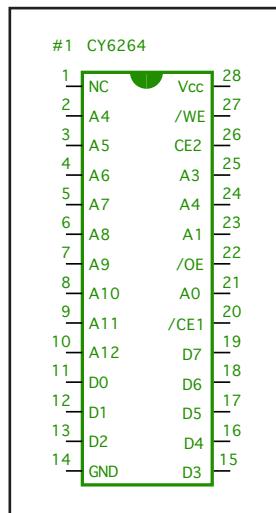


Figure 7.16 CY6264 Pinout

Address Lines (A0–A12)

You'll see that there are 13 address lines, which is enough to address 2^{13} , or 8,192, locations—conveniently, though not unintentionally, 8 KB. For the most part, these pins will be connected directly to the address bus, but there are some tricks we can play to divide the RAM into two separate blocks.

Data Lines (D0–D7)

All eight data lines are connected directly to the data bus.

Chip Enable (/CE1, CE2)

When Chip Enable 1 is low and Chip Enable 2 is high, the chip is “turned on” and will attempt to either read data from the bus or write data to it.

Write Enable (/WE)

When the chip is enabled and Write Enable is low, the data on the data bus is written

to memory.

Output Enable (/OE)

When the chip is enabled and Output Enable is low, data from memory is output onto the data bus.

The difficulty in wiring up the RAM comes from having to divide it into two blocks, one at \$0000–\$0FFF and one at \$E000–\$EFFF. We must enable the chip only when the address on the address bus is within those ranges. The CY6264 has two chip enables. Both must be enabled to enable the chip. For simplicity, we're just going to use one (CE2) and tie /CE1 to ground so that it is always enabled. Since CE2 should be high for addresses in either range \$0000–\$0FFF or \$E000–\$EFFF, we can write the statement:

$$\text{CE2} = (\$0000 \text{ to } \$0FFF) + (\$E00 \text{ to } \$EFFF)$$

Keep in mind that the + here means *OR*. Since the lowest 12 bits in each part of the statement can change without having CE2, we can rewrite the equation:

$$\text{CE2} = (\$0xxx) + (\$Exxx)$$

Now we just need to look at the leftmost four bits, which represent pins A15, A14, A13, and A12. Converting 0 and E to binary, we get:

$$\begin{aligned}\$0 &= 0000b \\ \$E &= 1110b\end{aligned}$$

Given that those binary digits represent pins A15 through A12, we can write the statements:

$$\begin{aligned}\$0xxx &= A15' \cdot A14' \cdot A13' \cdot A12' \\ \$Exxx &= A15 \cdot A14 \cdot A13 \cdot A12'\end{aligned}$$

Then we put them together to realize our expression for CE2:

$$\text{CE2} = (A15' \cdot A14' \cdot A13' \cdot A12') + (A15 \cdot A14 \cdot A13 \cdot A12')$$

The circuit created from this expression can be seen in Figure 7.17. The implementation of the latter half is straightforward (four signals, one inverted, going into a quad-AND gate). To implement the former half, we could have used four inverters going into a quad-AND gate, but instead we took advantage of DeMorgan's Laws to create an equivalent circuit using OR gates. Since four OR gates come to a chip, this approach lends itself to a more efficient design.

Our RAM chip will be enabled whenever an address on the address bus is within its range. We need to add some circuitry to the chip's address lines to make it map to two separate blocks. Since we have 8 KB and we want two blocks of 4 KB, we can use one of the address lines as a switch. A12 can be that address line. Let's make A12 high when the address is in the \$Exxx region. A12 is low for every other region, but because our chip is only enabled for \$Exxx and \$0xxx, that effectively means A12 is only low for \$0xxx.

When enabled, our memory chip is always either reading or writing data, as dictated by the processor's Read/Write line. The Read/Write line is high when the processor is reading and low when the processor is writing. We need to connect this Read/Write line directly to the Output Enable (/OE) line of memory. We also connect the inverted Read/Write line to the chip's Write Enable (/WE), but since we don't want to write until the data is available, we AND the inverted R/W with the Phi 2 clock, which is used for timing memory writes. Both lines are active-low (are activated by a low signal, as opposed to a high). "Output Enable" corresponds to "Read." Because Read/Write is low when the processor wants to write and Write Enable is active-low, we can hook the Read/Write line directly to the Write Enable pin. Whenever we're not writing, we're reading; consequently, Output Enable should be high whenever Write Enable is low, and vice versa. This means Output Enable should be hooked up to the inverted Read/Write signal. The complete schematic is shown in Figure 7.17.

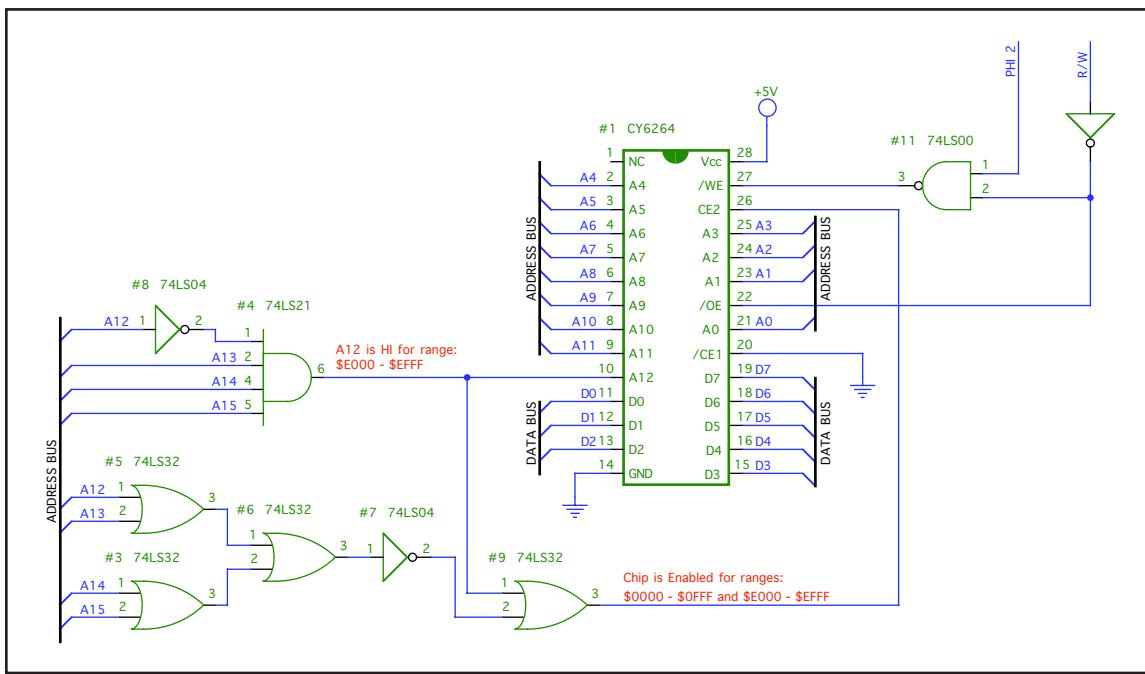


Figure 7.17 8 KB RAM Schematic

Dynamic and Static RAM

The original Apple I was built using dynamic RAM, which is still prevalent in computers today. In dynamic RAM, each memory cell (one bit) is made up of a transistor and a capacitor. If the capacitor loses its charge, the content of memory is forgotten. A capacitor will discharge after just a couple milliseconds, so dynamic memory relies upon external refresh circuitry to keep the capacitor charged. Static memory, by contrast, uses a flip-flop for each bit. A flip-flop, of course, requires no refreshing, so static memory is a lot easier to work with. It's also faster. Unfortunately, a flip-flop is a lot more complex than a mere transistor and capacitor; therefore, static memory is a lot more expensive, which is why dynamic memory remains so common today. A static RAM chip of a few kilobytes costs only a couple dollars, so for our project it's perfectly economical.

Implementing 32 KB RAM

The 32 KB RAM design used in the Replica I is far simpler than the 8 KB example we just examined. The chip used is a 62256 static RAM IC (Figure 7.18). The design for the 62256 mirrors our earlier example. The chip should be enabled whenever the lower half of memory is addressed. This occurs whenever A15 is low, giving us the result of $CE = A15$. Our only chip enable line is $/CE1$, so $/CE1 = A15$, so this leaves us with the Write Enable and Output (Read) Enable lines to configure. Writing takes precedence if both Write Enable and Output Enable are true. Therefore, we can enable output whenever the chip is enabled. Write should be enabled when Read/Write is low and Phi 2 (the clock used for memory writes) is high. The completed design is shown in Figure 7.19.

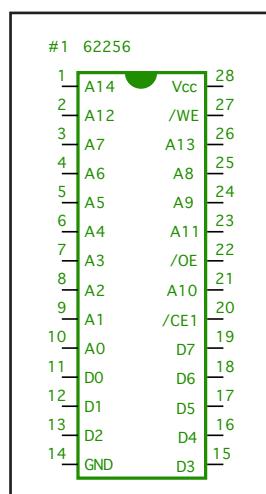


Figure 7.18 62256 Pinout

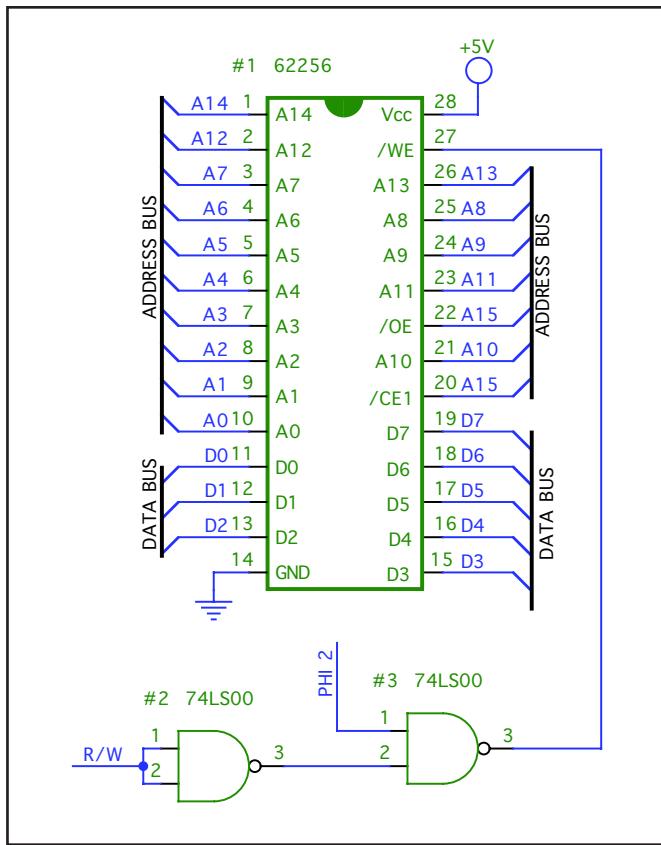


Figure 7.19 32 KB RAM Schematic

Implementing the EPROM

If you want to use an erasable programmable read-only memory (EPROM) solely for the Monitor program, as the Apple I does, the circuit is straightforward. This design uses a 2716 EPROM, which is much bigger than we need, so address lines A8 through A10 are permanently tied low. This leaves us with 2^8 , or 256, bytes of addressable space—exactly the number we need for the Monitor ROM range of \$FF00 to \$FFFF. Address lines A0 through A7 can be connected to the address bus, and the data lines can be connected to the data bus.

Output Enable, which is active-low, should be low whenever read is high; subsequently, /OE is connected to the R/W line through an inverter. Lastly, the chip should be enabled whenever the address is in the \$FFxx range. Refer to the schematic in Figure 7.20.

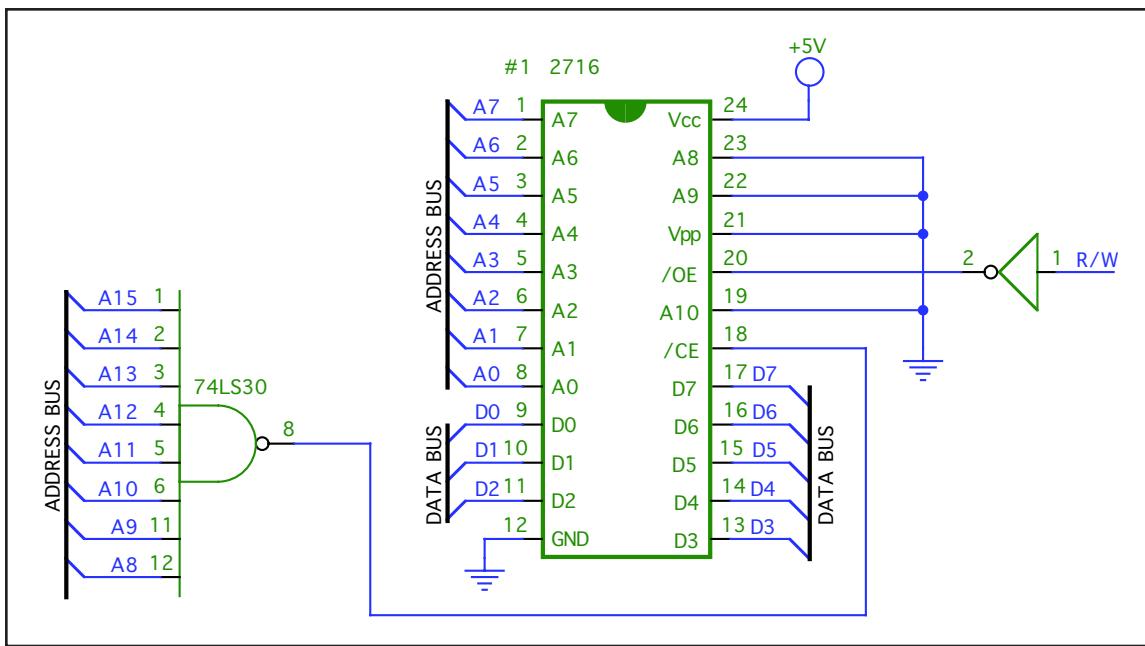


Figure 7.20 2716 EPROM Schematic

EPROM and EEPROM

An EPROM is a programmable read-only memory (PROM) that can be erased under ultraviolet light. An electrically-erasable programmable read-only memory is an EPROM that can be erased with an electrical charge. Both EPROMs and EEPROMs can be read an unlimited number of times, but erasing and writing is typically limited to a few hundred thousand times. EPROM is pronounced *ee-prom*, and EEPROM is pronounced *ee-ee-prom*.

Implementing the Expanded ROM

The Replica I uses an alternative chip for the ROM, a 28c64. It also uses a more complex method for chip enable. The EEPROM can be enabled for the entire range from \$E000 to \$FFFF. This can be conveniently done with a 74LS138 decoder. This same decoder can also be used to decode the addresses for our 6821 PIA, discussed in the next section; therefore, it is advantageous to become familiar with its use at this point.

Wiring the 74LS138

The 74LS138 is a one-of-eight decoder, giving us eight possible outputs. The EEPROM requires addresses \$E000 through \$FFFF. The 6821 PIA discussed later uses the \$Dxxx range. Ranges \$Axxx, \$Bxxx, and \$Cxxx aren't used, but since we have extra outputs, we can make them available for future projects. Table 7.3 shows the ranges expressed as hexadecimal and binary values.

Hexadecimal	Binary
\$Axxx	1010 xxxx xxxx xxxx b
\$Bxxx	1011 xxxx xxxx xxxx b
\$Cxxx	1100 xxxx xxxx xxxx b
\$Dxxx	1101 xxxx xxxx xxxx b
\$Exxx	1110 xxxx xxxx xxxx b
\$Fxxx	1111 xxxx xxxx xxxx b

Table 7.3 Addresses for the 74LS138 Decoder

The highest bit (A15) is 1 for all the ranges we're working with, so A15 can simply be used as an enable line (G1). Bits A12, A13, and A14 all vary, so they will be the input lines A, B, and C. For the range \$Fxxx, ABC = 111b. 111b equals 7, making the output Y7 true. For \$Exxx, ABC = 110b, which is 6 in binary, making Y6 true. The EEPROM should be enabled whenever Y6 or Y7 is true. Since /CE, /Y6, and /Y7 are all active-low, we should have an OR gate with an inverter at each input and output. Using DeMorgan's Laws, this is equivalent to an AND gate. Since this design already has a 7400 IC on board, a NAND followed by another NAND (to invert) is equivalent.

Wiring the 28C64

Since we're using this EEPROM as a simple ROM, Write Enable (/WE) is permanently tied high. The chip is only enabled when data is to be read from it, so it follows that /OE = /CE. The address lines go directly to the address bus and the data lines go directly to the data bus. A schematic of this is shown in Figure 7.21.

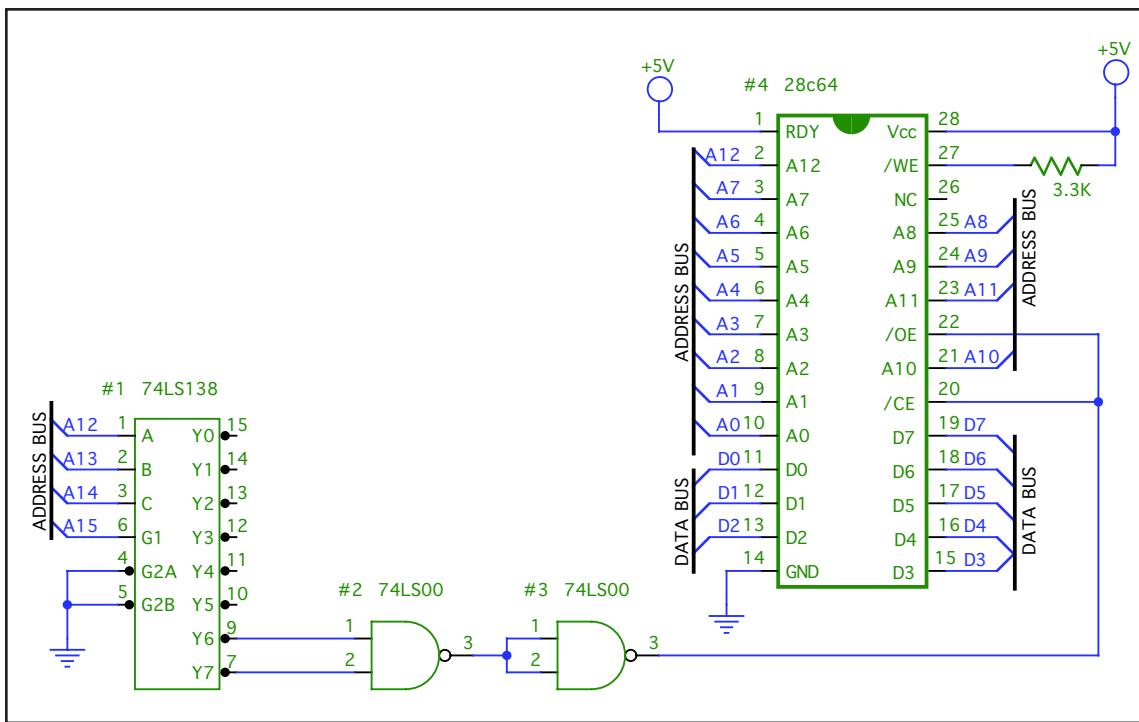


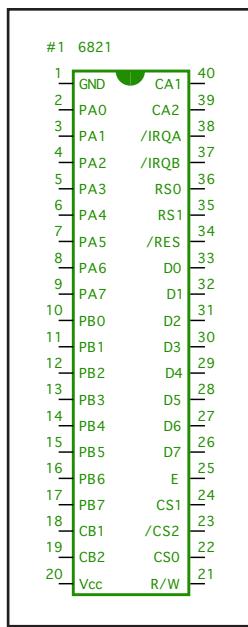
Figure 7.21 28c64 EEPROM Schematic

I/O with the 6821

The Motorola 6821 (Figure 7.22) is a *peripheral interface adapter* (PIA), which means that it's used for interfacing peripherals, such as keyboard and video, to the computer. The 6821 has two configurable eight-bit ports, PA and PB (for "port A" and "port B"), which can be used either for output or input. The Apple I uses PA for input from the keyboard and PB for output to the video circuit. The data is transferred to and from the computer via the data bus.

This section will provide a rather in-depth look at the operation of the 6821. If you plan to use the Apple I's standard interfaces for keyboard in and video out, this section might not interest you. If, on the other hand, you intend to substantially modify the Apple I circuit for some special purpose, or if you are interested in building your own microcomputer using the 6821, this information will be very helpful.

Each eight-bit port can be configured for either input or output. In fact, each bit direction in each port can be individually specified. When the Apple I is reset, the Monitor configures the 6821 to work with our circuit. On reset, both ports default to being inputs and the keyboard port is left untouched. The video port is set to output except for the highest bit, which is used as the "Data Available" line.

**Figure 7.22** PIA 6821 Pinout

The Monitor uses the information in Table 7.4 for setting up the address lines. Changing the value of the bits CRA 2 and CRB 2 can alter which register lines that RS1 and RS0 address. The bits in Control Register A (CRA), including bit 2 (CRA 2), can be set when RS1 = 1 and RS0 = 0. First, set CRA 2 to 0. Next, make RS1 = 0 and RS0 = 0. This allows us to pass eight bits to the 6821 that will set the data direction for register A. Then, set RS1 high and RS0 low to access CRA and then set CRA 2 to 0. Now, when RS1 = 0 and RS0 = 0, Peripheral Register A will be accessed, whether for reading or writing, depending upon which data direction bits were set. The same technique applies to Peripheral Register B.

Control Register Bit				
RS1	RS0	CRA 2	CRB 2	Location Selection
0	0	1	X	Peripheral Register A
0	0	0	X	Data Direction Register A
0	1	X	X	Control Register A
1	0	X	1	Peripheral Register B
1	0	X	0	Data Direction Register B
1	1	X	X	Control Register B

Table 7.4 Internal Addressing

Take a look at Figure 7.23. The Register Select lines RS0 and RS1 are connected to the Address lines A0 and A1, respectively. The chip select line CS0 goes to A4. For the other

chip select, CS2, we use the same 74LS138 as in the memory section to decode the high bits in the address, giving us everything in the \$Dxxx range. This combination of decoding makes it possible for us to use the addresses seen in Table 7.5.

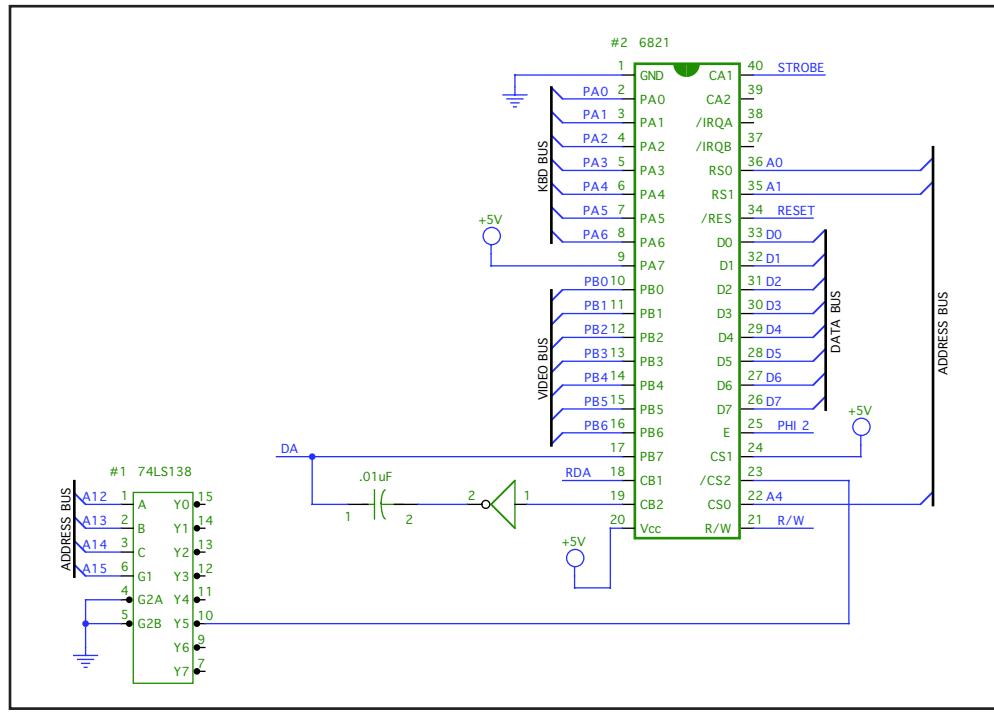


Figure 7.23 Control Registers A and B

Mnemonic	Apple I Name	6821 Name	Address	Description
DSP	Display Register	Peripheral Register B	\$D012	A character being sent to the display is held in this register until the video section is ready to take it.
DSPCR	Display Control Register	Control Register B	\$D013	Defines the behavior of the Display Register.
KBD	Keyboard Register	Peripheral Register A	\$D010	A character from the keyboard is held in this register until it is requested by the processor.
KBDCR	Keyboard Controller Register	Control Register A	\$D011	Defines the behavior of the Keyboard Register.

Table 7.5 External Addressing

When the circuit is reset, the DSP and KBD ports are in configuration mode (that is, bits CRA 2 and CRB 2 are low). This means that when we send data to DSP or KBD, it will go to the data direction register instead of the peripheral register. By sending \$7F (that's 0111 1111b) to DSP, the lower seven bits are set to outputs and the highest bit is set to an input.

Next, the control registers (Figures 7.24 and 7.25) need to be configured. Both are configured in the same way, as shown in Figure 7.26.

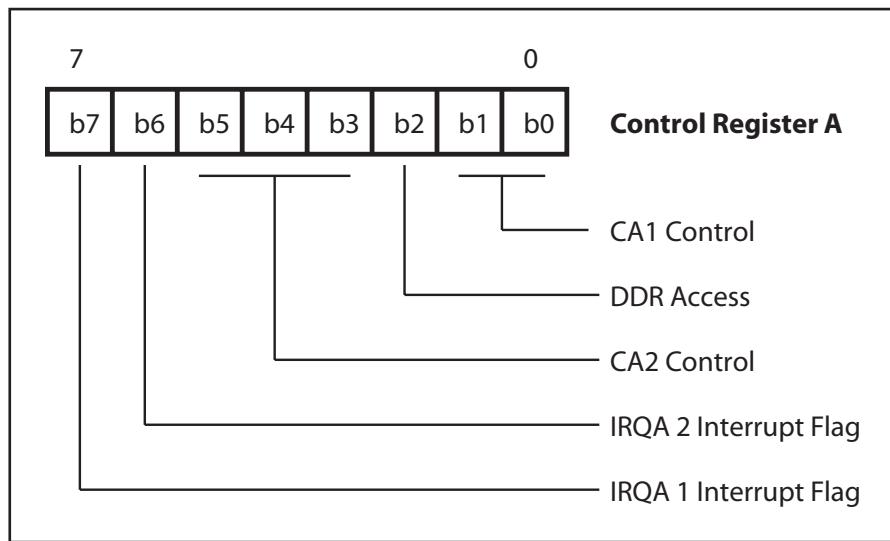


Figure 7.24 Keyboard Control Register

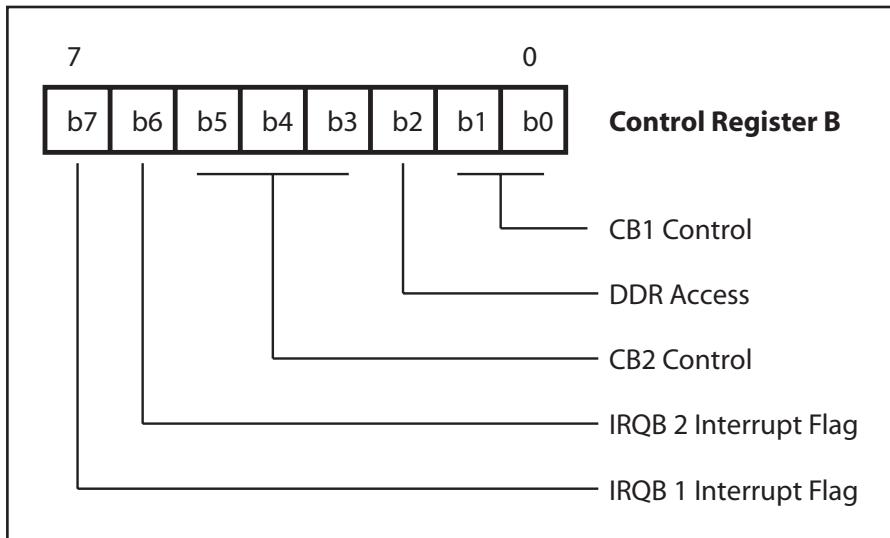


Figure 7.25 Display Control Register

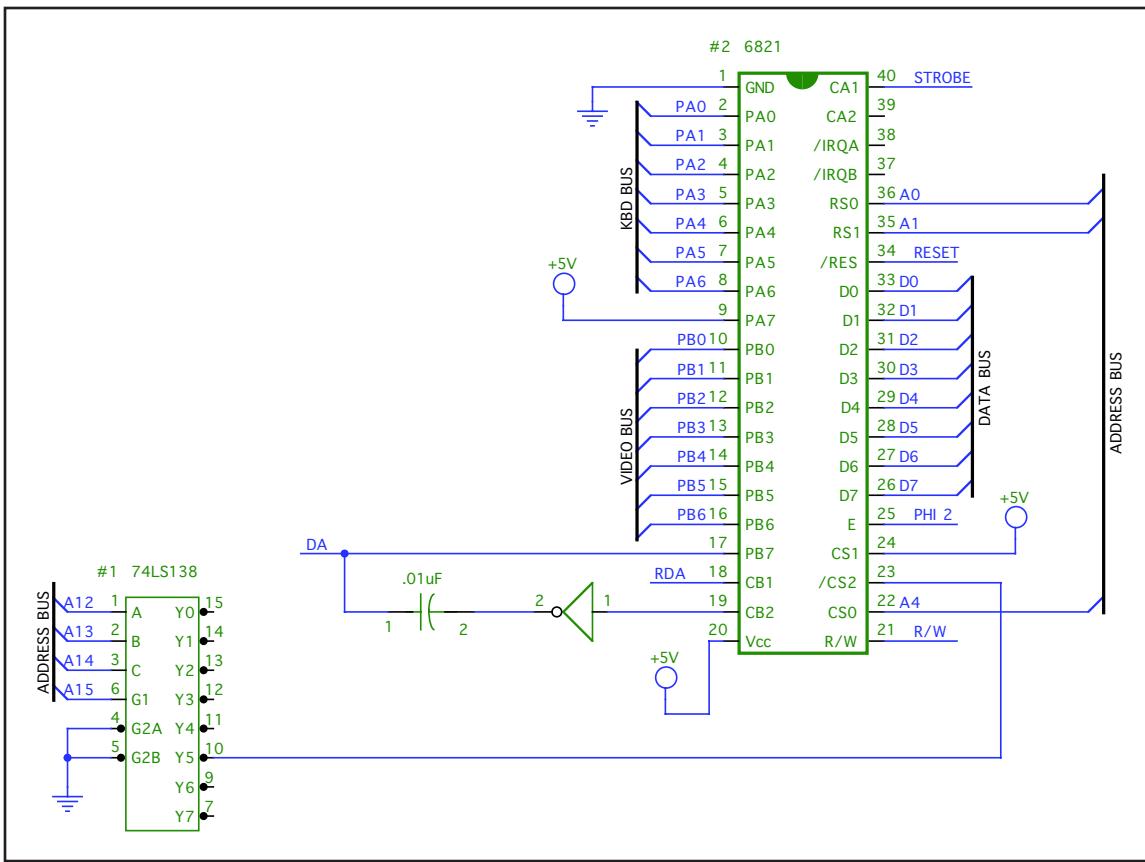


Figure 7.26 Control Registers A and B

DDR Access

Note that b2, which is CRA 2 and CRB 2, is high. This changes the register selection from the data direction register to the peripheral register. Now, data sent to DSP and KBD will go to the peripheral registers, not to the data direction registers.

CA1 (CB1) Control

b0 enables interrupts on CA1 and CB1. CA1 is connected to the keyboard's strobe line. Whenever the keyboard sends a strobe signal, the 6821 loads the character on the KBD bus into its keyboard register. CB1 is used for the video circuit and is connected to RDA (Ready Data Accept). RDA comes from the video section and tells the 6821 that the video section is ready to receive a character.

b1 is high, which sets CA1 and CB1 to look for high-to-low transitions, as opposed

to low-to-high transitions. This means that a change from 0 to 1 will cause an interrupt, whereas a change from 1 to 0 will not.

CA2 (CB2) Control

The keyboard's CA2 has no connection; consequently, these settings are only important for the video's CB2. b5 is high, which means CB2 is an output. b4 does nothing in this configuration. b3 is a flag which goes high when CB1, which is connected to the RDA line, goes high. The flag is cleared on the clock pulse (Phi 2 on pin E) after data is loaded into the display register.

IRA Interrupt Flags

IRQA and IRQB are not used.

Keyboard

When a pulse arrives on the strobe line, the character from the keyboard is loaded into the keyboard register and a flag is set in the keyboard control register. In the Apple I's memory map, the keyboard register is at \$D010 and the keyboard control register is at \$D011. The processor checks for the flag at \$D011. When it sees it, the processor goes to \$D010 and loads the character.

Video

The processor sends a character to the 6821, which then sends it out to the video section. The display register is at \$D012. Before sending data, the processor checks bit 7 at memory location \$D012. If this bit is high, it means the "Data Available" line is high, and the 6821 currently has data it is waiting to send to the video section. The processor waits until the bit goes low, indicating the 6821 has sent its data and is now ready to talk to the processor. The processor then sends its character to the display register, which is located at \$D012 in the memory map. The 6821 loads this character into the display register and tells the video section that it has data available by setting DA (Data Available) high. When the video section is ready, it responds by setting RDA (Ready Data Accept) high. The 6821 sends the character to the video section and sets DA to low.

Keyboard Input

The Apple I uses a very simple ASCII keyboard (Figure 7.27). When a key is pressed, that character's seven-bit ASCII value is placed on the keyboard's data lines. A pulse on the strobe line indicates the data is ready. Many ASCII keyboards will also have a reset button. This button is connected to the processor's reset line. The rest of the signals, including data lines and strobe, go to the 6821. A schematic of this layout is shown in Figure 7.28.



Figure 7.27 ASCII Keyboard

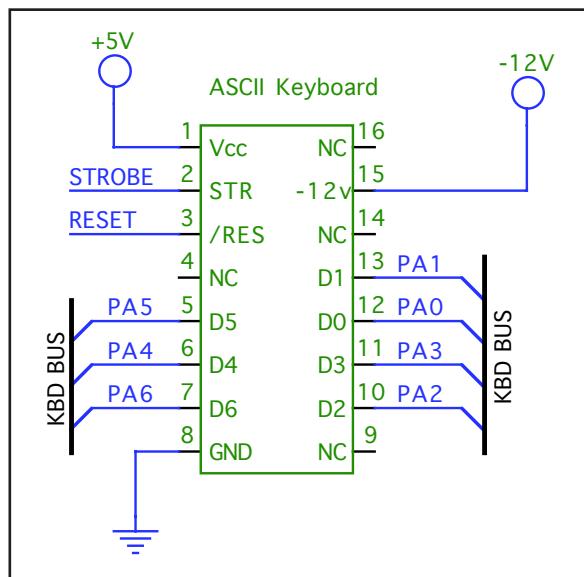


Figure 7.28 ASCII Keyboard Schematic

An ASCII keyboard is the simplest way to get data into an Apple I replica. Unfortunately, these keyboards are becoming very hard to find. The most reliable source is an Apple II+, but that computer is becoming a rarity in its own right. If you do borrow a II+ keyboard, treat the computer gently. The Apple II+ is a fun computer you'll want to keep around.

For those who can't find or don't want an ASCII keyboard, there is an alternative. Briel Computers has designed a PS/2 interface around an ATmega microcontroller (Figure 7.29). The PS/2 keyboard sends its scancodes to the microcontroller. The microcontroller converts them to ASCII and sends them to the 6821, using the strobe line to indicate when data is available.

You'll need an 8MHz crystal and two 18pF capacitors to provide a clock for the microcontroller, and a PS/2 connector to plug your keyboard into. If you want the option of switching between a PS/2 keyboard and an ASCII keyboard, provide a jumper between the ATmega's reset pin and ground (as shown in Figure 7.29). Since Reset is active-low, jumpering it to ground will disable the ATmega by placing it permanently into reset mode, letting you use your ASCII keyboard without conflict.

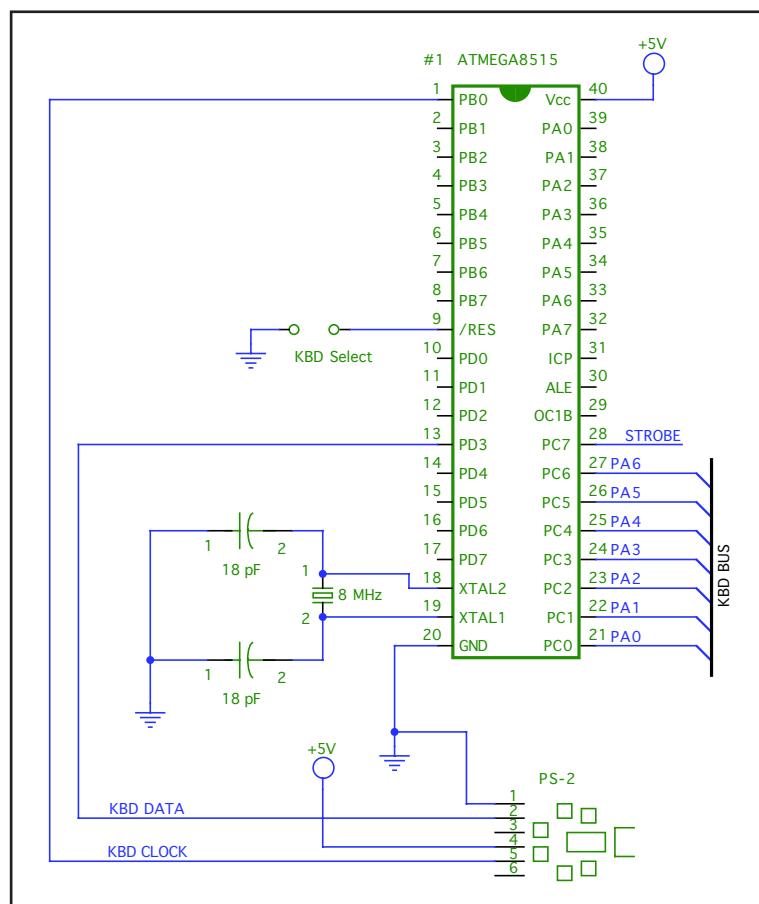


Figure 7.29 PS/2 Keyboard Schematic

Microcontrollers

A *microcontroller* is a specialized microcomputer contained on a single chip. Microcontrollers are often used to control intelligent peripherals such as hard drives and are now common even in simple devices such as keyboards and mice. You'll find many Web sites and books devoted to the microcontroller hobby.

Changes in the Replica I TE

Briel Computers' newest version of the Replica, the Replica I TE, leaves the processor section unchanged, but significantly updates the video circuitry and keyboard interface. Instead of using an Atmel ATmega8 AVR and ATmega8515 for the video circuitry and keyboard interface, respectively, a Parallax Propeller microcontroller now handles both. The Propeller also provides a serial interface. The video section of the Propeller sends data to both the video port and serial interface, while the keyboard section listens for data from both the keyboard and serial interface.

Video Output

The Apple I has very limited video capabilities. Most computers have video mapped to memory. On a black-and-white display, each pixel would be represented in memory by a single bit—1 for black and 0 for white. The Apple I's video section is text only. Instead of storing each individual pixel for a character, it stores the character's ASCII code. The video section has a ROM containing the pixel layouts for every character it supports. The layout table is referenced for each pixel of each character that is being displayed. This saves a tremendous amount of space, but at significant expense in versatility. Only the characters stored in that ROM can be displayed on screen. Thus, no bitmapped graphics can be displayed.

Even more limiting is the fact that once a character is sent to the display, it cannot be modified. It's there until you enter enough lines that it scrolls off the top of the screen, or until you manually clear the entire display. In this way, the function of the video is very similar to a typewriter or Teletype.

Why did Apple make these choices? At that time, memory—which would have allowed full graphics—was expensive. Instead, to save money, shift registers were used. When a character was sent to the video section, it was loaded into the shift register. As

new characters appeared, it was slowly shifted through the register and then out. The Apple I's video section, which performed this shifting and then converted the signal into one readable video monitor, is far more complex than the processor section, which we've spent this chapter discussing. Understanding it also requires a decent knowledge of analog circuits, which is outside the scope of this book.

There are a couple of alternatives to a full-blown video section, such as hooking the output up to a Teletype, using a liquid crystal display (LCD) that accepts ASCII input, or setting up a serial connection to a PC. The option we're going to focus on uses a micro-computer design by Briel Computers to deliver output to a video monitor. This chip and a few TTL chips are equivalent to the original Apple I's video section.

You can see the circuit in Figure 7.30. At the center is the ATmega microcontroller, which is the heart of our video section. To the left of it is the clock circuitry. Depressing the Clear switch resets the microcontroller and clears the display. The 74LS74 is used as a clock divider, to cut the clock frequency down to about 7.1MHz. The 74LS166 is a shift register, which sends our data out to the display. Output goes to an ordinary composite monitor through an RCA jack (Figure 7.31). The diodes prevent any accidental signals the video monitor might generate from reaching and damaging the circuitry.

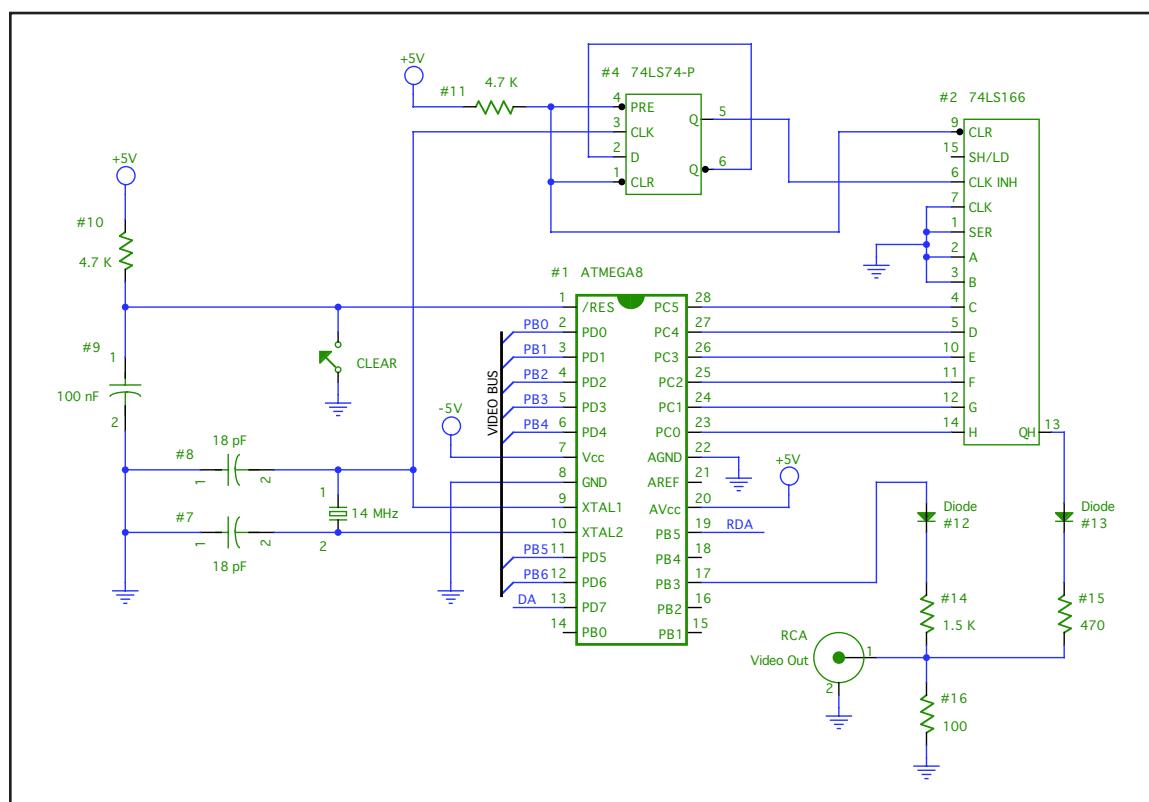


Figure 7.30 Briel Computers Replica I Video Section Schematic

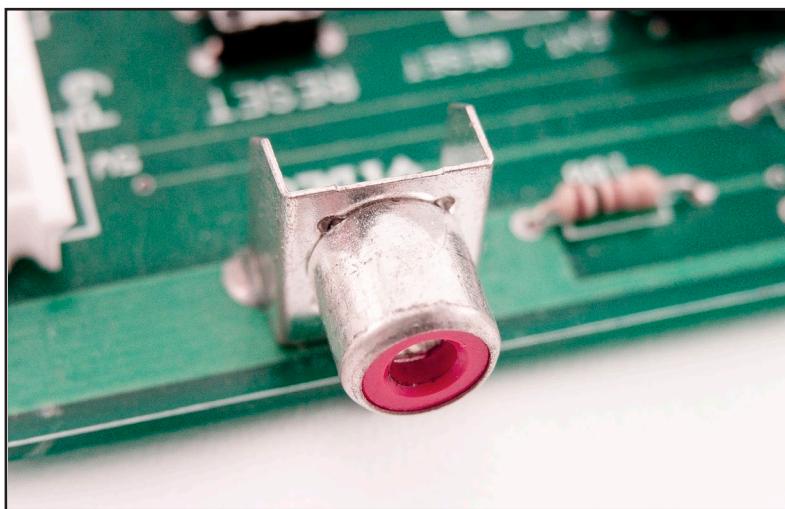


Figure 7.31 RCA Phono Jack

The electron beam in the video monitor's cathode ray tube (CRT) scans the display line by line. It starts in the upper-left corner and moves right. The display is black and white. When turned on, the electron beam creates a white pixel on the display, which indicates its current location. When turned off, that pixel remains black. A 0-volt signal turns the beam on to full power and creates a white pixel. A 3-volt signal turns it completely off. Anything in between produces a shade of gray.

The electron beam is constantly moving and our signal is constantly changing its voltage. If we wanted to send a line of alternating black and white pixels, we would alternate between 0 and 3 volts for every pixel. The beam then moves to the next line. When the entire display has been drawn, the electron beam moves back to the upper-left corner and begins drawing it over again. The entire process is so quick that the beam is redrawing the first pixels on the display before they even start to fade. Look at the CRT through the display of a digital camera and you'll be able to see a line moving down the screen as the display refreshes.

Summary

Though very simple relative to microcomputers today, there is nonetheless quite a bit to learn when studying the Apple I. In this chapter we've examined the processor, memory, and input/output circuitry. With this information, and perhaps some help from other members of the Apple I Owners Club on Applefritter, you can begin writing software and modifying the hardware design. Whatever your successes or failures, be sure to share what you learn with the rest of the Apple I community at www.applefritter.com/apple1.

Appendix A

ASCII Code Chart

Dec	Hex	Char
0	0	NUL (null)
1	1	SOH (start of heading)
2	2	STX (start of text)
3	3	ETX (end of text)
4	4	EOT (end of transmission)
5	5	ENQ (enquiry)
6	6	ACK (acknowledge)
7	7	BEL (bell)
8	8	BS (backspace)
9	9	TAB (horizontal tab)
10	A	LF (line feed)
11	B	VT (vertical tab)
12	C	FF (form feed)
13	D	CR (carriage return)
14	E	SO (shift out)
15	F	SI (shift in)
16	10	DLE (data link escape)
17	11	DC1 (device control 1)
18	12	DC2 (device control 2)
19	13	DC3 (device control 3)
20	14	DC4 (device control 4)
21	15	NAK (neg. acknowledge)
22	16	SYN (synchronous idle)
23	17	ETB (end of trans.)
24	18	CAN (cancel)
25	19	EM (end of medium)
26	1A	SUB (substitute)
27	1B	ESC (escape)
28	1C	FS (file separator)
29	1D	GS (group separator)
30	1E	RS (record separator)
31	1F	US (unit separator)

Dec	Hex	Char
32	20	Space
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
39	27	'
40	28	(
41	29)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
92	5D]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	
125	7D	}
126	7E	~
127	7F	DEL

Appendix B

Operation Codes and Status Register

The following pages contains a table referencing operation codes and status registers, with their corresponding hexadecimal codes. For more information on opcodes and programming in Assembly, refer to Chapter 6.

Mnemonic	a	(a,x)	a,x	a,y	(a)	A	#	i	r	zp	(zp,x)	zp,y	(zp)	y	Processor Status Register			
	6D	7D	79		69	65	61	75		71	N	6	5	4	3	2	1	0
	2D	3D	39		29	25	21	35		31	V	N	Z	Z	C	Z	C	
ADC																		
AND	OE	1E		0A		90	B0	F0										
ASL																		
BCC																		
BCS																		
BEQ																		
BIT	2C				89		24											
BMI																		
BNF																		
BPL																		
BRA																		
BRK																		
BVC																		
BVS																		
CLC																		
CID																		
CLI																		
CLV																		
CMP	CD	DD	D9		C9	C5	C1	D5		D1	N	N	Z	C	Z	C	Z	C
CPX	EC	EE		E0	E4						N	N	Z	Z	Z	Z	Z	C
CPY	CC	CC		C0	C4						N	N	Z	Z	Z	Z	Z	Z
DEC	CE	DE		C6	C6						N	N	Z	Z	Z	Z	Z	Z
DEX																		
DEY																		
EOR	4D	5D	59		49	88	45	41										
INC	EE	EE																
INX																		
INY																		
JMP	4C	7C																

Mnemonic	a	(a,x)	a,x	a,y	(a)	A	#	i	r	zp	(zp,x)	zp,y	(zp,y)	Processor Status Register			
										N	V	1	1	D	I	Z	C
JSR	20		BD	B9		A9				A5	A1	B5	B1	N	N	Z	
LDA	AD		BE	BE		A2				A6		B6		N	N	Z	Z
LDX	AE		BC			A0				A4		B4		N	N	Z	Z
LDY	AC		5E		4A				46		56		O		Z	C	
LSR	4E																
NOP																	
ORA	0D		1D	19		09				05	01	15		N		Z	
PHA																	
PHP																	
PLA																	
PLP																	
ROL	2E		3E		2A					26		36		N	N	Z	C
ROR	6E		7E		6A					66		76		N	N	Z	C
RTI																	
RTS																	
SBC	ED		FD	F9		E9				E5	E1	F5	F1	N	V	1	
SEC																	
SED																	
SEI																	
STA	8D				9D	99								91			
STX	8E																
STY	8C																
TAX																	
TAY																	
TSX																	
TXA																	
TXS																	
TYA																	

Appendix C

Operation Code Matrix

The following pages contains a matrix correlating hexadecimal codes with their corresponding operation codes and status registers. This matrix is, essentially, the reverse of the table in Appendix B. For more information on opcodes and programming in Assembly, refer to Chapter 6.

	0	1	2	3	4	5	6	7
0	BRK i	ORA (zp,x)				ORA zp	ASL zp	
1	BPL r	ORA (zp),y				ORA zp,x	ASL zp,x	
2	JSR a	AND (zp,x)		BIT zp		AND zp	ROL zp	
3	BMI r	AND (zp),y				AND zp,x	ROL zp,x	
4	RTI i	EOR (zp,x)				EOR zp	LSR zp	
5	BVC r	EOR (zp),y				EOR zp,x	LSR zp,x	
6	RTS s	ADC (zp,x)				ADC zp	ROR zp	
7	BVS r	ADC (zp),y				ADC zp,x	ROR zp,x	
8		STA (zp,x)		STY zp		STA zp	STX zp	
9	BCC r	STA (zp),y		STY zp,x		STA zp,x	STX zp,y	
A	LDY #	LDA (zp,x)	LDX #	LDY zp		LDA zp	LDX zp	
B	BCS r	LDA (zp),y		LDY zp,x		LDA zp,x	LDX zp,x	
C	CPY #	CMP (zp,x)		CPY zp		CMP zp	DEC zp	
D	BNE r	CMP (zp),y				CMP zp,x	DEC zp,x	
E	CPX #	SBC (zp,x)		CPX zp		SBC zp	INC zp	
F	BEQ R	SBC (zp),y				SBC zp,x	INC zp,x	
	0	1	2	3	4	5	6	7

8	9	A	B	C	D	E	F
PHP i	ORA #	ASL A			ORA a	ASL a	0
CLC i	ORA a,y				ORA a,x	ASL a,x	1
PLP i	AND #	ROL A	BIT a		AND a	ROL a	2
SEC i	AND a,y				AND a,x	ROL a,x	3
PHA i	EOR #	LSR A	JMP a		EOR a	LSR a	4
CLI i	EOR a,y				EOR a,x	LSR a,x	5
PLA i	ADC #	ROR A	JMP (a)		ADC a	ROR a	6
SEI i	ADC a,y				ADC a,x	ROR a,x	7
DEY i	BIT #	TXA i	STY a		STA a	STX a	8
TYA i	STA a,y	TXS i			STA a,x		9
TAY i	LDA #	TAX i	LDY a		LDA a	LDX a	A
CLV i	LDA a,y	TSX i	LDY a,x		LDA a,x	LDX a,y	B
INY i	CMP #	DEX i	CPY a		CMP a	DEC a	C
CLD i	CMP a,y				CMP a,x	DEC a,x	D
INX i	SBC #	NOP i	CPX a		SBC a	INC a	E
SED i	SBC a,y				SBC a,x	INC a,x	F
8	9	A	B	C	D	E	F

Appendix D

Instructions by Category

Load and Store

LDA — Load Accumulator with Memory

M → A
Flags: N, Z

Addressing Mode	Opcode
a	AD
a,x	BD
a,y	B9
#	A9
zp	A5
(zp,x)	A1
zp,x	B5
(zp).y	B1

LDX — Load Index X with Memory

M → X
Flags: N, Z

Addressing Mode	Opcode
a	AE
a,y	BE
#	A2
zp	A6
zp,y	B6

LDY — Load Index Y with Memory

$M \rightarrow Y$
Flags: N, Z

Addressing Mode	Opcode
a	AC
a,x	BC
#	A0
zp	A4
zp,x	B4

STA — Store Accumulator in Memory

$A \rightarrow M$
Flags: *none*

Addressing Mode	Opcode
a	8D
a,x	9D
a,y	99
zp	85
(zp,x)	81
zp,x	95
(zp).y	91

STX — Store Index X in Memory

$X \rightarrow M$
Flags: *none*

Addressing Mode	Opcode
a	8E
zp	86
zp,y	96

STY — Store Index Y in Memory

$Y \rightarrow M$

Flags: *none*

Addressing Mode	Opcode
a	8C
zp	84
zp,x	94

Arithmetic

ADC — Add Memory to Accumulator with Carry

$A + M + C \rightarrow A$

Flags: N, V, Z, C,

Addressing Mode	Opcode
a	6D
a,x	7D
a,y	79
#	69
zp	65
(zp,x)	61
zp,x	75
(zp),y	71

SBC — Subtract Memory from Accumulator with Borrow

$A - M - \sim C \rightarrow A$

Flags: N, V, Z, C

Addressing Mode	Opcode
a	ED
a,x	FD
a,y	F9
#	E9
zp	E5
(zp,x)	E1
zp,x	F5
(zp).y	F1

Increment and Decrement

INC — Increment Memory by One

$M + 1 \rightarrow M$

Flags: N, Z

Addressing Mode	Opcode
a	EE
a,x	FE
zp	E6
zp,x	F6

INX — Increment Index X by One

$X + 1 \rightarrow X$

Flags: N, Z

Addressing Mode	Opcode
i	E8

INY — Increment Index Y by One

$Y + 1 \rightarrow Y$

Flags: N, Z

Addressing Mode	Opcode
i	C8

DEC — Decrement Memory by One

$M - 1 \rightarrow M$

Flags: N, Z

Addressing Mode	Opcode
a	CE
a,x	DE
zp	C6
zp,x	D6

DEX — Decrement Index X by One

$X - 1 \rightarrow X$

Flags: N, Z

Addressing Mode	Opcode
i	CA

DEY — Decrement Index Y by One

$Y - 1 \rightarrow Y$

Flags: N, Z

Addressing Mode	Opcode
i	88

Shift and Rotate

ASL — Accumulator Shift Left One Bit

$C \leftarrow 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0 \leftarrow 0$

Flags: N, Z, C

Addressing Mode	Opcode
a	0E
a,x	1E
A	0A
zp	06
zp,x	16

LSR — Logical Shift Right One Bit

$0 \rightarrow 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0 \rightarrow C$

Flags: N, Z, C

Addressing Mode	Opcode
a	4E
a,x	5E
A	4A
zp	46
zp,x	56

ROL — Rotate Left One Bit

$C \leftarrow 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0 \leftarrow C$

Flags: N, Z, C

Addressing Mode	Opcode
a	2E
a,x	3E
A	2A
zp	26
zp,x	36

ROR — Rotate Right One Bit

$C \rightarrow 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0 \rightarrow C$

Flags: N, Z, C

Addressing Mode	Opcode
a	6E
a,x	7E
A	6A
zp	66
zp,x	76

Logic

AND — AND Memory with Accumulator

$A \wedge M \rightarrow A$

Flags: N, Z

Addressing Mode	Opcode
a	2D
a,x	3D
a,y	39
#	29
zp	25
(zp,x)	21
zp,x	35
(zp),y	31

ORA — OR Memory with Accumulator

$A \vee M \rightarrow A$

Flags: N, Z

Addressing Mode	Opcode
a	0D
a,x	1D
a,y	19
#	09
zp	05
(zp,x)	01
zp,x	15
(zp),y	11

EOR — Exclusive-OR Memory with Accumulator

$A \vee M \rightarrow A$

Flags: N, Z

Addressing Mode	Opcode
a	4D
a,x	5D
a,y	59
#	49
zp	45
(zp,x)	41
zp,x	55
(zp).y	51

Compare and Test Bit

For all Compare instructions:

Condition	N	Z	C
Register < Memory	1	0	0
Register = Memory	0	1	1
Register > Memory	0	0	1

CMP — Compare Memory and Accumulator

$A - M$

Flags: N, Z, C

Addressing Mode	Opcode
a	CD
a,x	DD
a,y	D9
#	C9
zp	C5
(zp,x)	C1
zp,x	D5
(zp).y	D1

CPX — Compare Memory and Index X

X - M

Flags: N, Z, C

Addressing Mode	Opcode
a	EC
#	E0
zp	E4

CPY — Compare Memory and Index Y

Y - M

Flags: N, Z, C

Addressing Mode	Opcode
a	CC
#	C0
zp	C4

BIT — Test Bits in Memory with Accumulator

A ^ M

Flags: N = M7, V = M6, Z

Addressing Mode	Opcode
a	2C
#	89
zp	24

Branch

BCC — Branch on Carry Clear

Branch if C = 0

Flags: *none*

Addressing Mode	Opcode
r	90

BCS — Branch on Carry Set

Branch if C = 1

Flags: *none*

Addressing Mode	Opcode
r	B0

BEQ — Branch on Result Zero

Branch if Z = 1

Flags: *none*

Addressing Mode	Opcode
r	F0

BMI — Branch on Result Minus

Branch if N = 1

Flags: *none*

Addressing Mode	Opcode
r	30

BNE — Branch on Result not Zero

Branch if Z = 0

Flags: *none*

Addressing Mode	Opcode
r	D0

BPL — Branch on Result Plus

Branch if N = 0

Flags: *none*

Addressing Mode	Opcode
r	10

BVC — Branch on Overflow Clear

Branch if V = 0

Flags: *none*

Addressing Mode	Opcode
r	50

BVS — Branch on Overflow Set

Branch if V = 1

Flags: *none*

Addressing Mode	Opcode
r	70

Transfer

TAX — Transfer Accumulator to Index X

$A \rightarrow X$
Flags: N, Z

Addressing Mode	Opcode
i	AA

TXA — Transfer Index X to Accumulator

$X \rightarrow A$
Flags: N, Z

Addressing Mode	Opcode
i	8A

TAY — Transfer Accumulator to Index Y

$A \rightarrow Y$
Flags: N, Z

Addressing Mode	Opcode
i	AB

TYA — Transfer Index Y to Accumulator

$Y \rightarrow A$
Flags: N, Z

Addressing Mode	Opcode
i	98

TSX — Transfer Stack Pointer to Index X

$S \rightarrow X$
Flags: N, Z

Addressing Mode	Opcode
i	BA

TXS — Transfer Index X to Stack Register

$X \rightarrow S$
Flags: N, Z

Addressing Mode	Opcode
i	9A

Stack

PHA — Push Accumulator on Stack

$A \rightarrow S$
Flags: *none*

Addressing Mode	Opcode
i	48

PLA — Pull Accumulator from Stack

$S \rightarrow A$
Flags: N, Z

Addressing Mode	Opcode
i	68

PHP — Push Processor Status on Stack

P → S

Flags: *none*

Addressing Mode	Opcode
i	08

PLP — Pull Processor Status from Stack

S → P

Flags: *all*

Addressing Mode	Opcode
i	28

Subroutines and Jump

JMP — Jump to New Location

Jump to new location

Flags: *none*

Addressing Mode	Opcode
a	4C
(a,x)	7C
(a)	6C

JSR — Jump to New Location Saving Return Address

Jump to Subroutine

Flags: *none*

Addressing Mode	Opcode
a	20

RTS — Return from Subroutine

Return from Subroutine

Flags: *none*

Addressing Mode	Opcode
i	60

RTI — Return from Interrupt

Return from Interrupt

Flags: *none*

Addressing Mode	Opcode
i	40

Set and Clear

SEC — Set Carry Flag

$1 \rightarrow C$

Flags: *none*

Addressing Mode	Opcode
i	38

SED — Set Decimal Mode

$1 \rightarrow D$

Flags: *none*

Addressing Mode	Opcode
i	F8

SEI — Set Interrupt Disable Status

$1 \rightarrow I$

Flags: *none*

Addressing Mode	Opcode
i	78

CLC — Clear Carry Flag

$0 \rightarrow C$

Flags: $C = 0$

Addressing Mode	Opcode
i	18

CLD — Clear Decimal Mode

$0 \rightarrow D$

Flags: $D = 0$

Addressing Mode	Opcode
i	D8

CLI — Clear Interrupt Disable Bit

$0 \rightarrow I$

Flags: $I = 0$

Addressing Mode	Opcode
i	58

CLV — Clear Overflow Flag

$0 \rightarrow V$

Flags: $V = 0$

Addressing Mode	Opcode
i	B8

Miscellaneous

NOP — No Operation

No Operation

Flags: *none*

Addressing Mode	Opcode
i	EA

BRK — Break

Force an interrupt

Flags: B = 1, I = 1

Addressing Mode	Opcode
i	00

Appendix E

Electrical Engineering Basics

Introduction

Understanding how hardware hacks work usually requires an introductory-level understanding of electronics. This appendix describes electronics fundamentals and the basic theory of the most common electronic components. We also look at how to read schematic diagrams, how to identify components, proper soldering techniques, and other engineering topics.

This appendix is not going to turn you into an electronics guru, but it will teach you enough about the basics so that you can start to find your way around. For more detail on the subject, see the suggested reading list at the end of this appendix.

Fundamentals

It is important to understand the core fundamentals of electronics before venturing into the details of specific components. This section provides a background on numbering systems, notation, and basic theory used in all facets of engineering.

Bits, Bytes, and Nibbles

At the lowest level, electronic circuits and computers store information in binary format, which is a base-2 numbering system containing only 0 and 1, each known as a *bit* (a portmanteau of *binary digit*). By comparison, the common decimal numbering system we use in everyday life is a base-10, which consists of the digits 0 through 9.

Electrically, a 1 bit is generally represented by a positive voltage (SV, for example), and a 0 bit is generally represented by a zero voltage (or ground potential). However, many protocols and definitions map the binary values in different ways.

A group of 4 bits is a *nibble* (also known as a *nybble*), 8 bits is a *byte*, and 16 bits is

typically defined as a *word* (though a word is sometimes defined differently, depending on the system architecture you are referring to). Figure E.1 shows the interaction of bits, nibbles, bytes, and words. This diagram makes it easy to grasp the concept of how they all fit together.

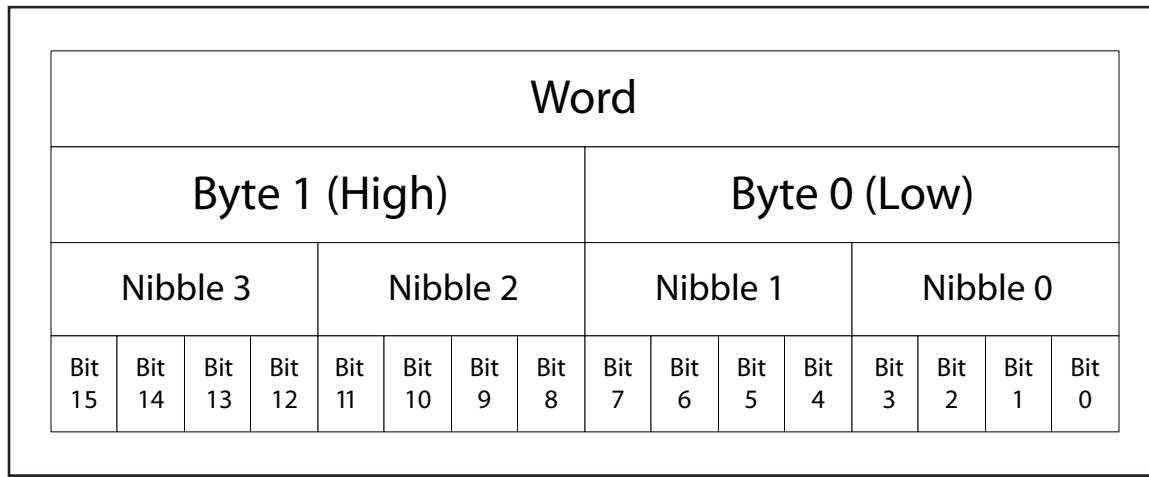


Figure E.1 Breakdown of a 16-Bit Word into Bytes, Nibbles, and Bits

The larger the group of bits, the more information that can be represented. A single bit can represent only two combinations (0 or 1). A nibble can represent 2^4 (or 16) possible combinations (0 to 15 in decimal); a byte can represent 2^8 (or 256) possible combinations (0 to 255 in decimal); and a word can represent 2^{16} (or 65,536) possible combinations (0 to 65,535 in decimal).

Hexadecimal format, also called *hex*, is commonly used in the digital computing world to represent groups of binary digits. It is a base-16 system in which 16 sequential values are used as base units before adding a new position for the next number (digits 0 through 9 and letters A through F). One hex character can represent the arrangement of 4 bits (a nibble). Two hex characters can represent 8 bits (a byte). Table E.1 shows equivalent number values in the decimal, hexadecimal, and binary number systems. Hex characters are sometimes prefixed with 0x or \$ to avoid confusion with other numbering systems.

The American Standard Code for Information Interchange, or ASCII (pronounced *ask-key*), is the common code for storing characters in a computer system. The ASCII character set (Table E.2) uses 1 byte to correspond to each of 128 different letters, numbers, punctuation marks, and special characters. Many of the special characters are holdovers from the original specification created in 1968 and are no longer commonly used for their original intended purpose. Only the decimal values 0 through 127 are assigned, which is half of the space available in a byte. An extended ASCII character set uses the full range of 256 characters, in which the decimal values of 128 through 255 are assigned to represent other special characters that are used in foreign languages, graphics, and mathematics.

Decimal	Binary	Hex
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14
21	10101	15
22	10110	16
23	10111	17
24	11000	18
25	11001	19
26	11010	1A
27	11011	1B
28	11100	1C
29	11101	1D
30	11110	1E
31	11111	1F
32	100000	20
...
63	111111	3F
...
127	1111111	7F
...
255	11111111	FF

Table E.1 Number System Equivalents: Decimal, Binary, and Hexadecimal

Dec	Hex	Char
0	0	NUL (null)
1	1	SOH (start of heading)
2	2	STX (start of text)
3	3	ETX (end of text)
4	4	EOT (end of transmission)
5	5	ENQ (enquiry)
6	6	ACK (acknowledge)
7	7	BEL (bell)
8	8	BS (backspace)
9	9	TAB (horizontal tab)
10	A	LF (line feed)
11	B	VT (vertical tab)
12	C	FF (form feed)
13	D	CR (carriage return)
14	E	SO (shift out)
15	F	SI (shift in)
16	10	DLE (data link escape)
17	11	DC1 (device control 1)
18	12	DC2 (device control 2)
19	13	DC3 (device control 3)
20	14	DC4 (device control 4)
21	15	NAK (neg. acknowledge)
22	16	SYN (synchronous idle)
23	17	ETB (end of trans.)
24	18	CAN (cancel)
25	19	EM (end of medium)
26	1A	SUB (substitute)
27	1B	ESC (escape)
28	1C	FS (file separator)
29	1D	GS (group separator)
30	1E	RS (record separator)
31	1F	US (unit separator)

Dec	Hex	Char
32	20	Space
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
39	27	'
40	28	(
41	29)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Table E.2 The Standard ASCII Character Set

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
92	5D]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	
125	7D	}
126	7E	~
127	7F	DEL

Table E.2 Continued The Standard ASCII Character Set

Reading Schematics

Before we get into the theory of individual electronic components, it is important to learn how circuit designs are drawn and described. A *schematic* is essentially an electrical road map of a circuit. Reading basic schematics is a good skill to have, even if it is just to identify a particular component that needs to be removed. Reading schematics is much easier than it may appear, and with practice it will become second nature.

In a schematic, each component of the circuit is assigned its own symbol, unique to the type of device it is. The United States and Europe sometimes use different symbols, and there are even multiple symbols to represent one type of part. A resistor has its own special symbol, as does a capacitor, diode, or integrated circuit. Think of schematic symbols as an alphabet for electronics. Table E.3 shows a selection of basic components and their corresponding designators and schematic symbols. This is by no means a complete list, and as mentioned, a particular component type may have additional symbols that aren't shown here.

A *part designator* is also assigned to each component and is used to distinguish between two parts of the same type and value. The designator is usually an alphanumeric character followed by a unique numerical value (R1, C4, or SW2, for example). The part designator and schematic symbol are used as a pair to define each discrete component of the circuit design.

Figure E.2 shows an example circuit using some of the basic schematic symbols. It describes a light-emitting diode (LED) powered by a battery and controlled by a switch. When the switch is off, no current is able to flow from the battery through the rest of the circuit, so the LED will not illuminate. When the switch is on, current will flow and the LED will illuminate.

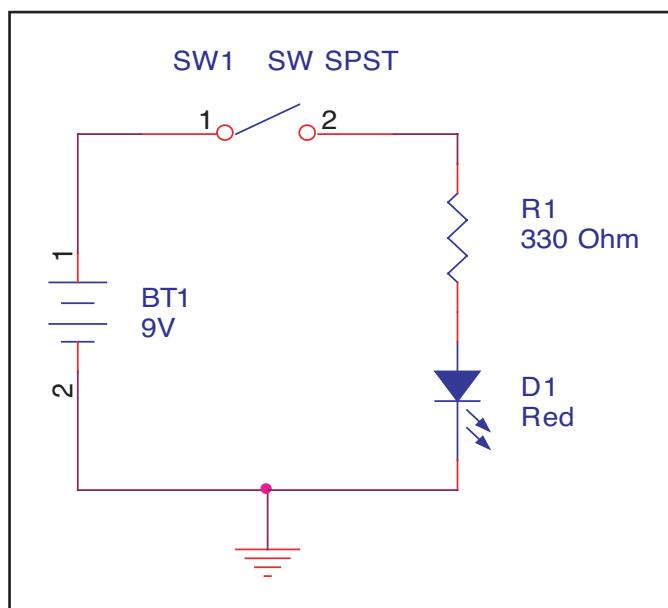


Figure E.2 An Example Circuit: A Basic LED with a Current-Limiting Resistor and Switch

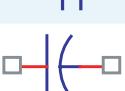
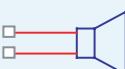
Component	Designator	Symbol
Resistor	R	
Potentiometer (variable resistor)	R	
Capacitor (nonpolarized)	C	
Capacitor (polarized)	C	
Diode	D	
LED	D	
Photodiode	D	
Transistor (NPN)	Q	
Transistor (PNP)	Q	
Crystal	Y	
Switch	SW	
Pushbutton switch	SW	
Speaker	LS	
Fuse	F	
Battery	BT	
Ground	None	
Ground	None	
Ground	None	

Table E.3 Designator and Schematic Symbols for Basic Electronic Components

Voltage, Current, and Resistance

Voltage and current are the two staple quantities of electronics. *Voltage*, also known as a *potential difference*, is the amount of work (energy) required to move a positive charge from a lower potential (a more negative point in a circuit) to higher potential (a more positive point in a circuit). Voltage can be thought of as an electrical pressure or force and has a unit of volts (V). It is denoted with a symbol V, or sometimes E or U.

Current is the rate of flow (the quantity of electrons) passing through a given point. Current has a unit of amperes, or *amps* (A), and is denoted with a symbol of I. Kirchhoff's Current Law states that the sum of currents into a point equals the sum of the currents out of a point (corresponding to a conservation of charge).

Power is a “snapshot” of the amount of work being done at that particular point in time and has a unit of watts (W). One watt of power is equal to the work done in one second by one volt moving one coulomb of charge. Furthermore, one coulomb per second is equal to one ampere. A coulomb is equal to 6.25×10^{18} electrons (a very, very large amount). Basically, the power consumed by a circuit can be calculated with the following simple formula:

$$P = V \times I$$

where

- P = Power (W)
- V = Voltage (V)
- I = Current (A)

Differentiating Between Voltage and Current

We use special terminology to describe voltage and current. You should refer to voltage as going between or across two points in a circuit—for example, “The voltage across the resistor is 1.7V.” You should refer to current going through a device or connection in a circuit—for example, “The current through the diode is 800mA.” When we’re measuring or referring to a voltage at a single given point in a circuit, it is defined with respect to ground (typically 0V).

Direct Current and Alternating Current

Direct current (DC) is simple to describe because it flows in one direction through a conductor and is either a steady signal or pulses. The most familiar form of a DC supply is a battery. Generally, aside from power supply or motor circuitry, DC voltages are commonly used in electronic circuits.

Alternating current (AC) flows in both directions through a conductor (see Figure

E.3) and is arguably more difficult to analyze and work with than DC. The most familiar form of an AC supply is an electrical outlet in your home. In the United States and Canada, these outlets provide 120V AC at 60Hz (cycles per second). In other parts of the world, varying AC voltages and line frequencies are used.

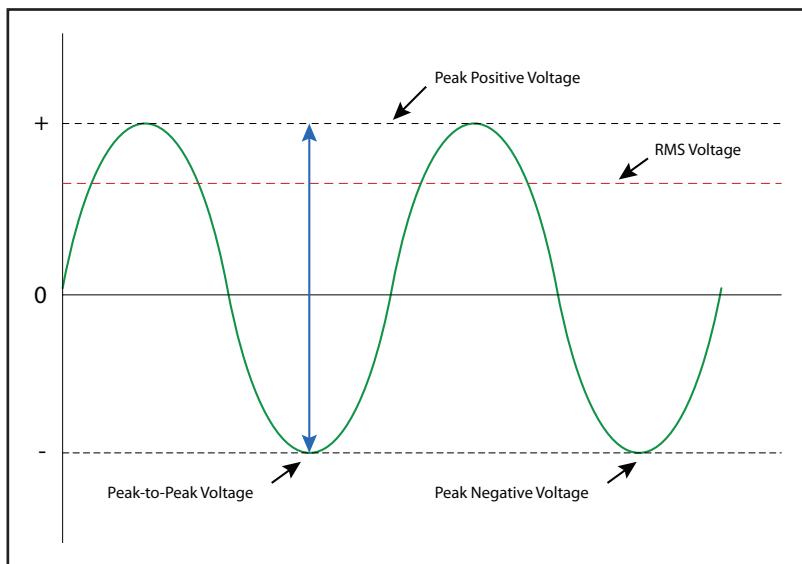


Figure E.3 An Example of an Alternating Current Waveform

Several terms are used to describe the AC signal:

- **Peak voltage (V_{PEAK})**. The maximum positive and negative points of the AC signal from a center point of reference.
- **Peak-to-peak voltage (V_{pp})**. The total voltage swing from the most positive to the most negative point of the AC signal.
- **Root-mean-square (RMS) voltage (V_{RMS})**. The most common term used to describe an AC voltage. Since an AC signal is constantly changing (as opposed to DC, in which the signal is constant), the RMS measurement is the most accurate way to determine how much work will be done by an AC voltage.

For a typical sinusoidal AC signal (like the one shown in Figure E.3), the following four formulas can be used:

$$\begin{aligned} \text{Average AC Voltage } (V_{AVG}) &= 0.637 \times V_{PEAK} = 0.9 \times V_{RMS} \\ V_{PEAK} &= 1.414 \times V_{RMS} = 1.57 \times V_{AVG} \\ V_{RMS} &= 0.707 \times V_{PEAK} = 1.11 \times V_{AVG} \\ V_{pp} &= 2 \times V_{PEAK} \end{aligned}$$

Resistance

Resistance can be described with a simple analogy of water flowing through a pipe: If the pipe is narrow (high resistance), the flow of water (current) will be restricted. If

the pipe is large (low resistance), water (current) can flow through it more easily. If the pressure (voltage) is increased, more current will be forced through the conductor. Any current prevented from flowing (if the resistance is high, for example) will be dissipated as heat (based on the first law of thermodynamics, which states that energy cannot be created or destroyed, simply changed in form). Additionally, there will be a difference in voltage on either side of the conductor.

Resistance is an important electrical property and exists in any electrical device. Resistors are devices used to create a fixed value of resistance. (For more information on resistors, see the “Basic Device Theory” section in this appendix.)

Ohm's Law

Ohm's Law, proven in the early 19th century by George Simon Ohm, is a basic formula of electronics that states the relationship between voltage, current, and resistance in an ideal conductor. The current in a circuit is directly proportional to the applied voltage and inversely proportional to the circuit resistance. Ohm's Law can be expressed as the following equations:

$$V = I \times R$$

or...

$$I = V \div R$$

or...

$$R = V \div I$$

where...

- V = Voltage (V)
- I = Current (A)
- R = Resistance (in ohms, designated with the omega symbol, Ω)

Basic Device Theory

This section explores the five most common electronic components: resistors, capacitors, diodes, transistors, and integrated circuits. Understanding the functionality of these parts is essential to any core electronics knowledge and will prove useful in designing or reverse-engineering products.

Resistors

Resistors are used to reduce the amount of current flowing through a point in a system. Resistors are defined by three values:

- Resistance (Ω)
- Heat dissipation (in watts, W)
- Manufacturing tolerance (%)

A sampling of various resistor types is shown in Figure E.4. Resistors are not polarized, meaning that they can be inserted in either orientation with no change in electrical function.

The value of a resistor is indicated by an industry-standard code of four or five colored bands printed directly onto the resistor (Figure E.5). The bands define the resistance, multiplier, and manufacturing tolerance of the resistor. The manufacturing tolerance is the allowable skew of a resistor value from its ideal rated value.

A resistor's internal composition can consist of many different materials, but the most common three types are carbon, metal film, and wire-wound. The material is usually wrapped around a core, with the wrapping type and length corresponding to the resistor value. Carbon-filled resistors, used in most general-purpose applications such as current limiting and nonprecise circuits, allow a 5% tolerance on the resistor value. Metal film resistors are for more precise applications such as amplifiers, power supplies, and sensitive analog circuitry; they usually allow a 1% or 2% tolerance. Wire-wound resistors can also be very accurate.

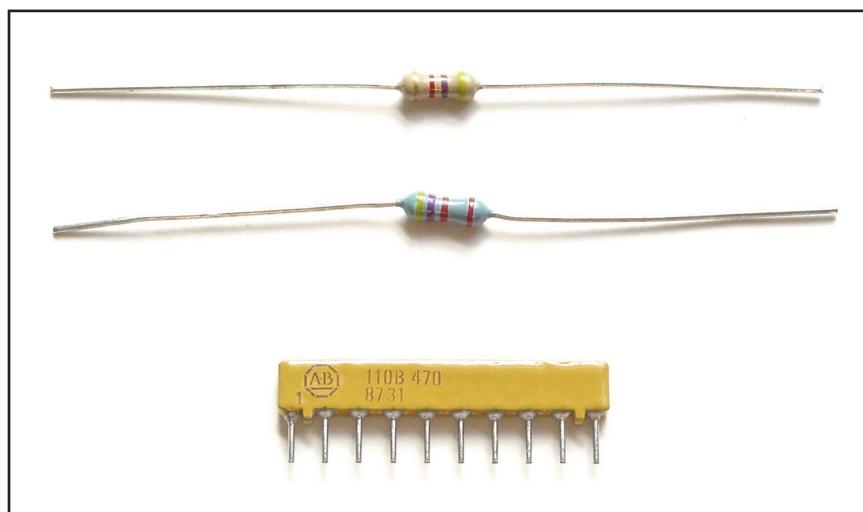
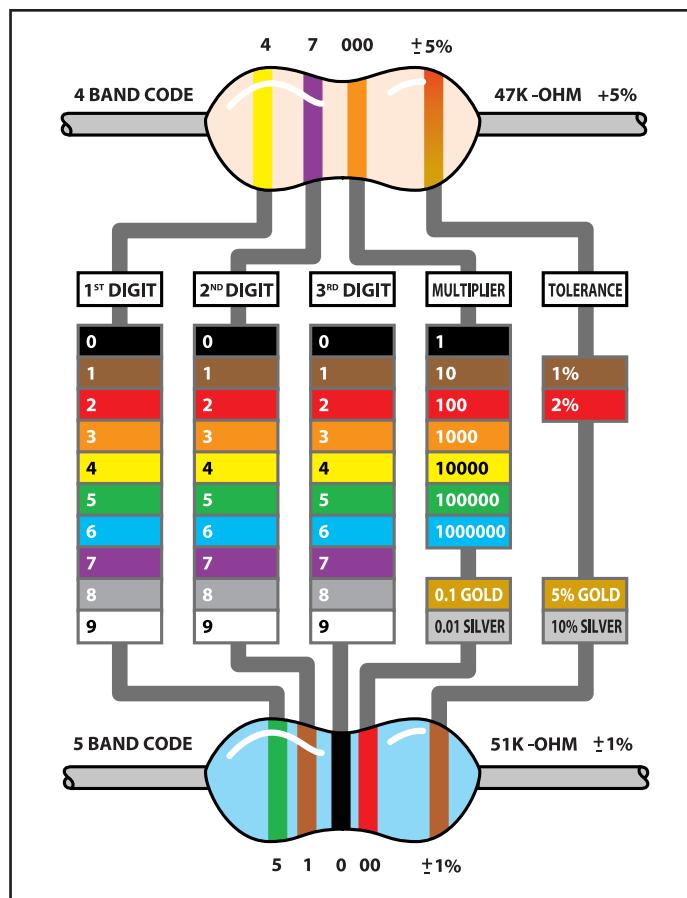
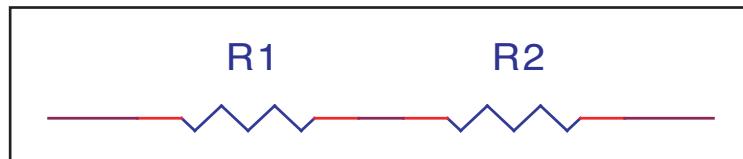


Figure E.4 Various Resistor Types

When resistors are used in series in a circuit (Figure E.6), their resistance values are additive, meaning that you simply add the values of the resistors in series to obtain the total resistance. For example, if R1 is 220 ohms and R2 is 470 ohms, the overall resistance will be 690 ohms.

**Figure E.5** Resistor Color Code Chart**Figure E.6** Resistors in Series

Parallel circuits provide alternative pathways for current flow, although the voltage across the components in parallel is the same. When resistors are used in parallel (Figure E.7), a simple equation is used to calculate the overall resistance:

$$1 \div R_{\text{TOTAL}} = (1 \div R1) + (1 \div R2) + \dots$$

This same formula can be extended for any number of resistors used in parallel. For example, if R1 is 220 ohms and R2 is 470 ohms, the overall resistance will be 149.8 ohms.

For only two resistors in parallel, an alternate formula can be used:

$$R_{\text{TOTAL}} = (R1 \times R2) \div (R1 + R2)$$

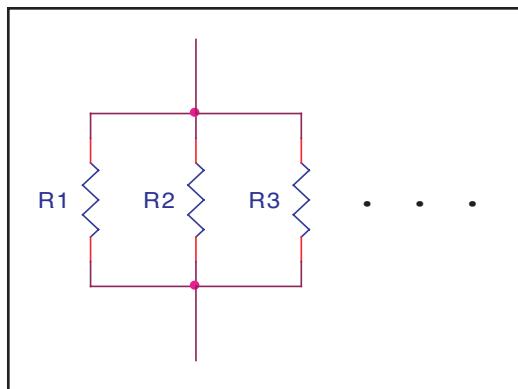


Figure E.7 Resistors in Parallel

Carbon and metal film resistors typically come in wattage values of 1/16W, 1/8W, 1/4W, 1/2W and 1W. This corresponds to how much power they can safely dissipate. The most commonly used resistors are 1/4W and 1/2W. For high-current applications, wire-wound resistors are typically used because they can support wattages greater than 1W. The wattage of the resistor usually corresponds to its physical size and surface area. For most consumer electronics, resistors greater than 1W are seldom used. To calculate the required wattage value for your application, use the following equation:

$$P = V \times I$$

or...

$$P = I^2 \times R$$

where...

- P = Power (W)
- V = Voltage across the resistor (V)
- I = Current flowing through the resistor (A)
- R = Resistance value (Ω)

Capacitors

A capacitor's primary function is to store electrical energy in the form of electrostatic charge. Consider a simple example of a water tower, which stores water (charge): When the water system (circuit) produces more water than a town or building needs, the excess is stored in the water tower (capacitor).

At times of high demand, when additional water is needed, the excess water (charge) flows out of the water tower to keep the pressure up.

A capacitor is usually implemented for one of three uses:

- **To store a charge.** Typically used for high-speed or high-power applications, such as a laser or a camera flash. The capacitor will be fully charged by the circuit in a

fixed length of time, and then all of its stored energy will be released and used almost instantaneously, just like in the water tower example.

- **To block DC voltage.** If a DC voltage source is connected in series to a capacitor, the capacitor will instantaneously charge and no DC voltage will pass into the rest of the circuit. However, an AC signal flows through a capacitor unimpeded because the capacitor will charge and discharge as the AC fluctuates, making it appear that the alternating current is flowing.

- **To eliminate ripples.** Useful for filtering, signal processing, and other analog designs. If a line carrying DC voltage has ripples or spikes in it, also known as “noise,” a capacitor can smooth or “clean” the voltage to a more steady value by absorbing the peaks and filling in the valleys of the signal.

Capacitors are constructed of two metal plates separated by a dielectric. The dielectric is any material that does not conduct electricity, and varies for different types of capacitors. It prevents the plates from touching each other. Electrons are stored on one plate of the capacitor and they discharge through the other. Consider lightning in the sky as a real-world example of a capacitor: One plate is formed by the clouds, the other plate is formed by the earth’s ground, and the dielectric is the air in between. The lightning is the charge releasing between the two plates.

Depending on their construction, capacitors are either polarized, meaning that they exhibit varying characteristics based on the direction they are used in a circuit, or nonpolarized, meaning that they can be inserted in either orientation with no change in electrical function. A sampling of various capacitor types is shown in Figures E.8 and E.9.

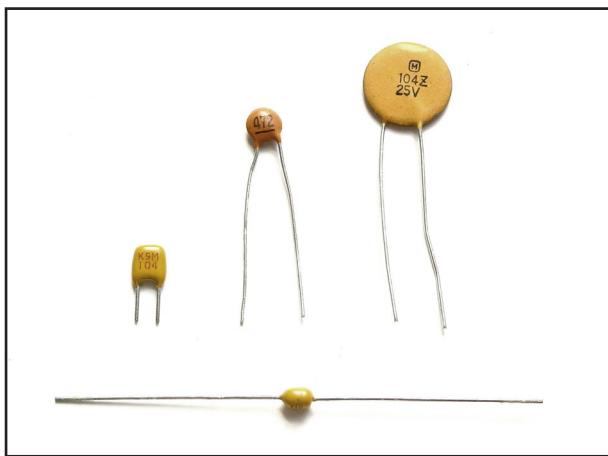


Figure E.8 Various Nonpolarized Capacitor Types (Ceramic Disc and Multilayer)

Capacitors are measured in *farads* (F). A 1-farad capacitor can store one coulomb of charge at 1 volt (equal to one amp-second of electrons at 1 volt). A single farad is a very large amount. Most capacitors store a minuscule amount of charge and are usually denoted in μF (microfarads, $10^{-6} \times \text{F}$) or pF (picofarads, $10^{-12} \times \text{F}$). The physical size of the capacitor is usually related to the dielectric material and the amount of charge that the capacitor can hold.

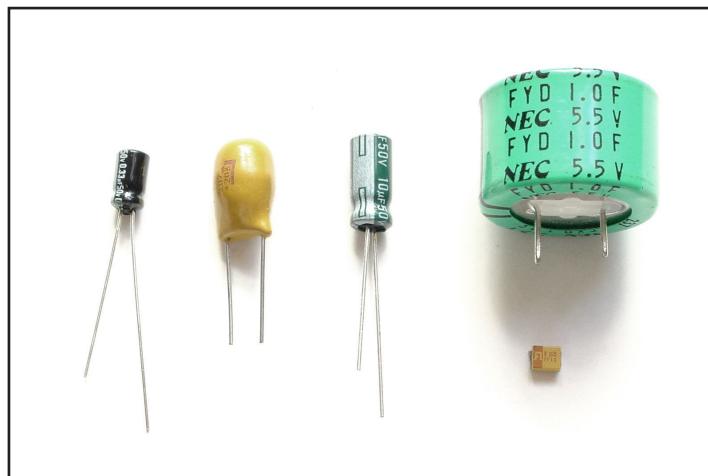


Figure E.9 Various Polarized Capacitor Types (Electrolytic and Tantalum)

Unlike resistors, capacitors do not use a color code for value identification. Today, most monolithic and ceramic capacitors are marked with a three-number code called an *IEC marking* (Figure E.10). The first two digits of the code indicate a numerical value; the last digit indicates a multiplier. Electrolytic capacitors are always marked in μF . These devices are polarized and must be oriented correctly during installation. Polarized devices have a visible marking denoting the negative side of the device (in the case of surface-mount capacitors, the marking is on the positive side). There may be additional markings on the capacitor (sometimes just a single character); these usually denote the capacitor's voltage rating or manufacturer.

VALUE	CODE	MULTILAYER (270 pF)	CERAMIC DISCS (.001 μF) (0.1 μF)	ELECTROLYTIC 1 μF
10 pF	= 100			
100 pF	= 101			
1000 pF	= 102	271		
.001 μF	= 102		102	
.01 μF	= 103			
.1 μF	= 104		104	35V 1 μF - +

Figure E.10 Examples of Some Capacitor IEC Markings

The calculations to determine effective capacitance of capacitors in series and parallel are essentially the reverse of those used for resistors. When capacitors are used in series (Figure E.11), a simple equation is used to calculate the effective capacitance:

$$\frac{1}{C_{\text{TOTAL}}} = \left(\frac{1}{C_1} + \frac{1}{C_2} + \dots \right)$$

This same formula can be extended for any number of capacitors used in series. For example, if C_1 is $100\mu\text{F}$ and C_2 is $47\mu\text{F}$, the overall capacitance will be $31.9\mu\text{F}$.

For only two capacitors in series, an alternate formula can be used:

$$C_{\text{TOTAL}} = (C_1 \times C_2) \div (C_1 + C_2)$$

When using capacitors in series, you store effectively less charge than you would by using either one alone in the circuit. The advantage to capacitors in series is that it

increases the maximum working voltage of the devices.

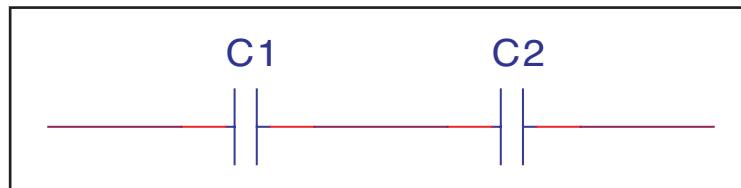


Figure E.11 Capacitors in Series

When capacitors are used in parallel in a circuit (Figure E.12), their effective capacitance is additive, meaning that you simply add the values of the capacitors in parallel to obtain the total capacitance. For example, if C1 is $100\mu\text{F}$ and C2 is $47\mu\text{F}$, the overall capacitance will be $147\mu\text{F}$.

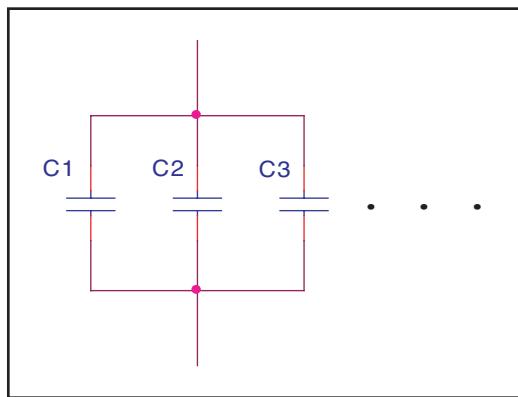


Figure E.12 Capacitors in Parallel

Capacitors are often used in combination with resistors in order to control their charge and discharge times. Resistance directly affects the time required to charge or discharge a capacitor (the higher the resistance, the longer the time).

Figure E.13 shows a simple RC circuit. The capacitor will charge as shown by the curve in Figure E.14. The amount of time for the capacitor to become fully charged in an RC circuit depends on the values of the capacitor and resistor in the circuit.

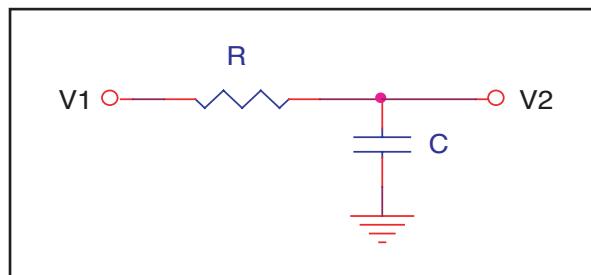


Figure E.13 A Simple RC Circuit to Charge a Capacitor

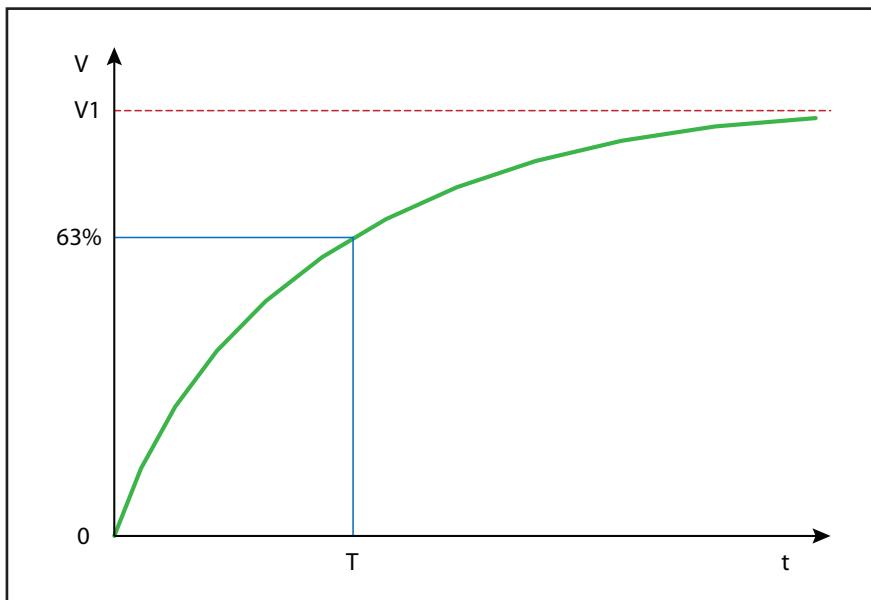


Figure E.14 Capacitor-Charging Curve

The variable T (called the *time constant*) is used to define the time it takes for the capacitor to charge to 63.2% of its maximum capacity. The time constant can be calculated by the following formula:

$$T = R \times C$$

where...

- T = Time constant (seconds)
- C = Capacitance (F)
- R = Resistance (Ω)

A capacitor reaches 63.2% of its charge in one-fifth of the time it takes to become fully charged. Capacitors in commercial applications are usually not charged to their full capacity because it takes too long.

Diodes

In the most basic sense, diodes pass current in one direction while blocking it from the other. This allows for their use in rectifying AC into DC, filtering, limiting the range of a signal (known as a *diode clamp*), and as “steering diodes,” in which diodes are used to allow voltage to be applied to only one part of a circuit.

Most diodes are made with semiconductor materials such as silicon, germanium, or selenium. Diodes are polarized, meaning that they exhibit varying characteristics depending on the direction they are used in a circuit. When current is flowing through

the diode in the direction shown in Figure E.15 (from anode, left, to cathode, right), the diode appears as a short circuit. When current tries to pass in the opposite direction, the diode exhibits a high resistance, preventing the current from flowing.

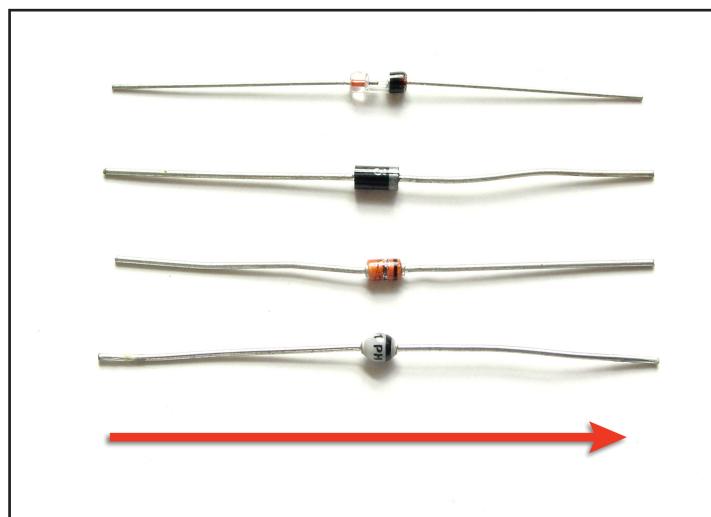


Figure E.15 Various Diode Types Showing Direction of Current Flow

Diodes come in many types and sizes, each with varying electrical properties. You need to consider a number of characteristics when designing with diodes or replacing one in a circuit:

- **Breakdown/reverse voltage (V_R)**, also known as the *peak inverse voltage* (P_{IV}), is the maximum voltage you can apply across a diode in the reverse direction and still have it block conduction. If this voltage is exceeded, the diode goes into “avalanche breakdown” and conducts current, essentially rendering the diode useless (unless it’s a Zener diode, which is designed to operate in this breakdown region).
- **Forward voltage (V_F)** is the voltage drop across the diode. This usually corresponds to the forward current (the greater the current flowing through the diode, the larger the voltage drop). Typical forward voltage of a general-purpose diode is between 0.5V and 0.8V at 10mA.
- **Forward current (I_F)** is the maximum current that can flow through the diode. If current flowing through the diode is more than it can handle, the diode will overheat and fail, causing a short circuit.
- **Reverse recovery time (T_{RR})** is the time it takes a diode to go from forward conduction to reverse blocking. (Think of this as a revolving door that goes in both directions, and the people coming in and going out are the current.) If the turnaround time is too slow, current will flow in the reverse direction when the polarity changes and cause the diode junction to heat up and possibly fail. This is primarily of concern for AC-rectifying circuits commonly used in power supplies.

Figure E.16 shows the diode V-I curve, a standard curve that illustrates the relationship between voltage and current with respect to a diode.

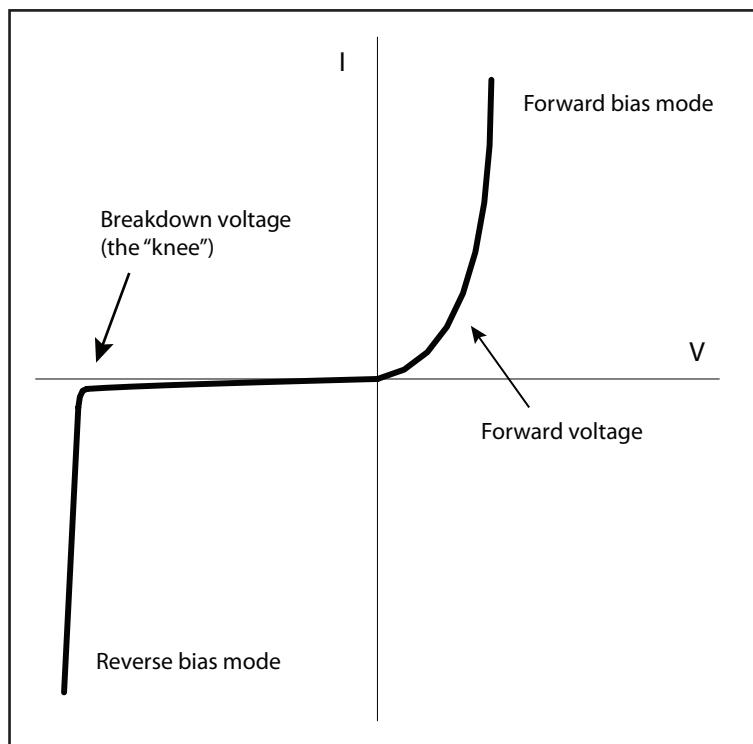


Figure E.16 The Diode V-I Curve

In normal forward bias operation (shown on the right side of the graph), the diode begins to conduct and act as a short circuit after the forward voltage drop is met (usually between 0.5V and 0.8V). When the diode is reverse biased (shown on the left side of the graph), current is essentially prevented from flowing in that direction, with the exception of a very small leakage current (measured in the nA range). The point at which the diode begins its avalanche breakdown is called the “knee,” as shown by the visible increase in reverse current on the curve, looking somewhat similar to a profile of a knee. Breakdown is not a desirable mode to which to subject the diode, unless the diode is of a Zener type (in which case proper current limiting should be employed).

Transistors

The transistor is arguably the greatest invention of the 20th century and the most important of electronic components. It is a three-terminal device that essentially serves as an amplifier or switch to control electronic current. When a small current is applied to its base, a much larger current is allowed to flow from its collector. This gives a transistor its switching behavior, since a small current can turn a larger current on and off.

The first transistor was demonstrated on December 23, 1947, by Bell Telephone scientists William Shockley, John Bardeen, and Walter Brattain. The transistor was the first device designed to act as both a transmitter, converting sound waves into electronic waves, and a resistor, controlling electronic current. The name transistor comes from the

words *transmitter* and *resistor*. Although its use has gone far beyond the function that combination implies, the name remains.

The transistor became commercially available in 1954 from Texas Instruments, and quickly replaced bulky and unreliable vacuum tubes, which were much larger and required more power to operate. Over 50 years later, transistors are now an essential part of engineering, used in practically every circuit and by the millions in single integrated circuits taking up an area smaller than a fingernail. Companies such as AMD, NEC, Samsung, and Intel are pushing the envelope of transistor technology, continuing to discover new ways to develop smaller, faster, and cheaper transistors.

This appendix only scratches the surface of transistor theory and focuses only on the most general terms. A sampling of various discrete transistors is shown in Figure E.17.

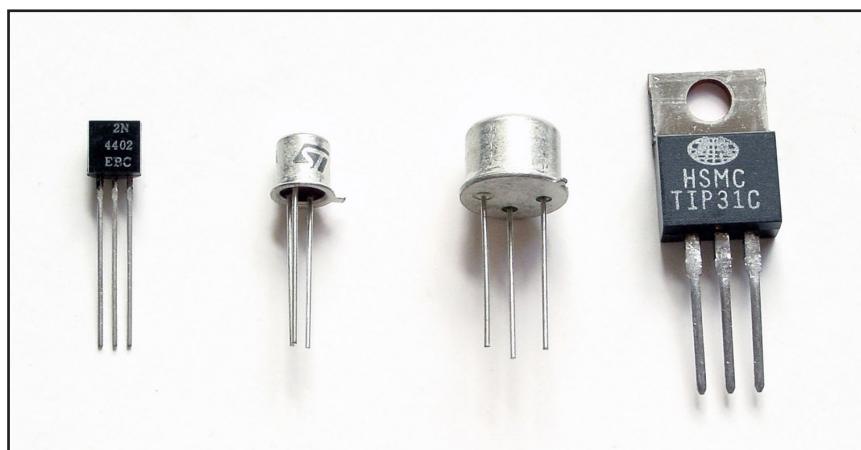


Figure E.17 Various Discrete Transistor Types

The transistor is composed of a three-layer sandwich of semiconductor material. Depending on how the material's crystal structure is treated during its creation (a process known as *doping*), it becomes more positively charged (P-type) or negatively charged (N-type). The transistor's structure contains a P-type layer between N-type layers (known as an NPN configuration) or an N-type layer sandwiched between P-type layers (known as a PNP configuration).

The voltages at transistor terminals—the collector, emitter, and base—are measured with respect to ground and are identified by their pin names, V_C , V_E , and V_B , respectively. The voltage drop measured between two terminals on the transistor is indicated by a double-subscript (for example, V_{BE} corresponds to the voltage drop from the base to the emitter). Figure E.18 shows the typical single NPN and PNP schematic symbols and notations.

A trick to help you remember which diagram corresponds to which transistor type is to think of NPN as “not pointing in” in reference to the base-emitter diode. Thus, the other transistor is the PNP type.

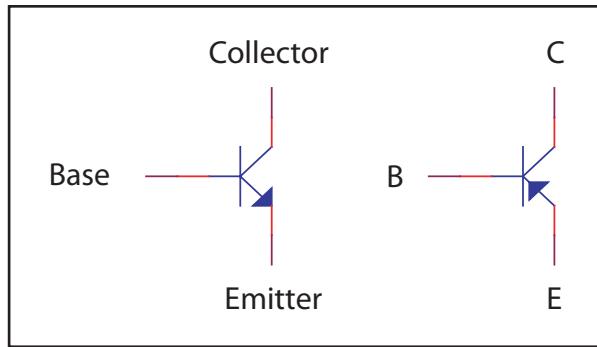


Figure E.18 NPN (Left) and PNP (Right) Transistor Diagrams

An NPN transistor has four properties that must be met (the properties for the PNP type are the same, except the polarities are reversed):

1. The collector must be more positive than the emitter.
2. The base-emitter and base-collector circuits look like two diodes back-to-back (Figure E.19). Normally the base-emitter diode is conducting (with a forward voltage drop, V_{BE} , of approximately 0.7V) and the base-collector diode is reversebiased.
3. Each transistor has maximum values of I_C , I_B , and V_{CE} that cannot be exceeded without risk of damaging the device. Power dissipation and other limits specified in the manufacturer's data sheet should also be obeyed.
4. The current flowing from collector to emitter (I_C) is roughly proportional to the current input to the base (I_B), shown in Figure E.20, and can be calculated with the following formula:

$$I_C = h_{FE} \times I_B$$

or

$$I_C = B \times I_B$$

where h_{FE} (also known as beta, B) is the current gain of the transistor. Typically, B is around 100, though it is not necessarily constant.

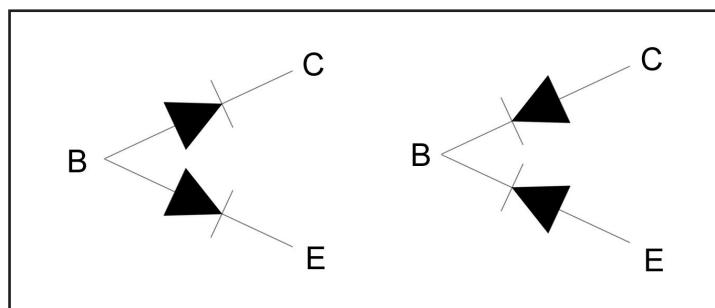


Figure E.19 Diode Representation of a Transistor, NPN (Left) and PNP (Right)

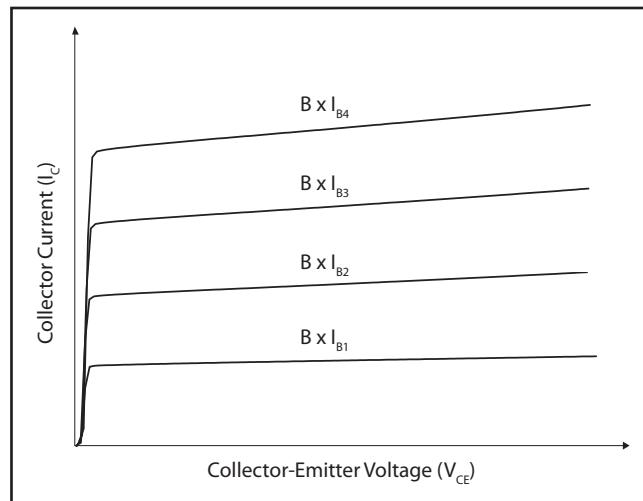


Figure E.20 NPN Transistor Characteristic Curve

Integrated Circuits

Integrated circuits (ICs) combine discrete semiconductor and passive components onto a single microchip of semiconductor material. These may include transistors, diodes, resistors, capacitors, and other circuit components. Unlike discrete components, which usually perform a single function, ICs are capable of performing multiple functions. There are thousands of IC manufacturers, but some familiar ones are Intel, Motorola, and Texas Instruments.

The first generation of commercially available ICs was released by Fairchild and Texas Instruments in 1961 and contained only a few transistors. In comparison, the Pentium 4 processor from Intel contains over 175 million transistors in a die area approximately the size of your thumbnail.

ICs are easy to identify in a circuit by their unique packaging. Typically, the silicon die (containing the microscopic circuitry) is mounted in a plastic or ceramic housing with tiny wires connected to it (Figure E.21). The external housing (called a *package*) comes in many mechanical outlines and various pin configurations and spacings.

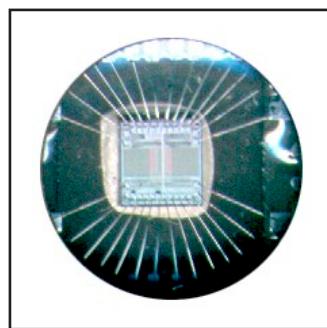


Figure E.21 Silicon Die Inside an Integrated Circuit

With the constant advances in technology, ICs are shrinking to inconceivable sizes. Figure E.22 shows a variety of IC packages, including, from left to right, Dual Inline Package (DIP), Narrow DIP, Plastic Leadless Chip Carrier (PLCC), Thin Small Outline Package (TSOP) Type II, TSOP Type I, Small Outline Integrated Circuit (SOIC), Shrink Small Outline Package (SSOP), and Small Outline Transistor (SOT-23).

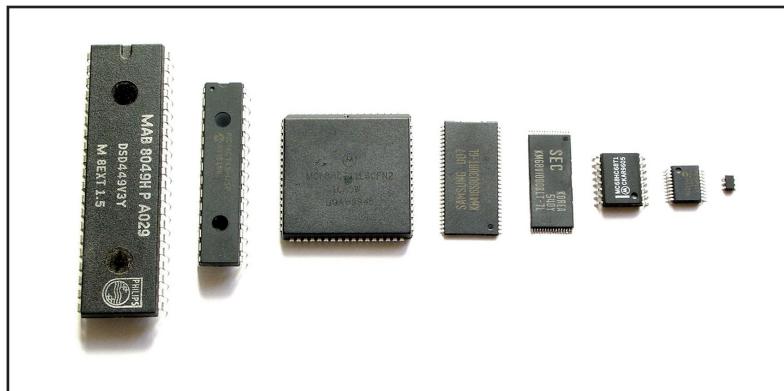


Figure E.22 Various IC Package Types

Ball Grid Array (BGA) is a relatively new package type that locates all the device leads underneath the chip, which reduces the area necessary for the device (Figure E.23). However, it is extremely difficult to access the balls of the BGA without completely removing the device, which limits its use by hobbyists. BGA devices are becoming more popular due to their small footprint and low failure rates. The testing process (done during product manufacturing) is more expensive as X-rays need to be used to verify that the solder has properly bonded to each of the ball leads.



Figure E.23 BGA Packaging

With Chip-on-Board (COB) packaging—colloquially referred to as *glop-tops*—the silicon die of the IC is mounted directly to the PCB and protected by epoxy encapsulation (Figure E.24).

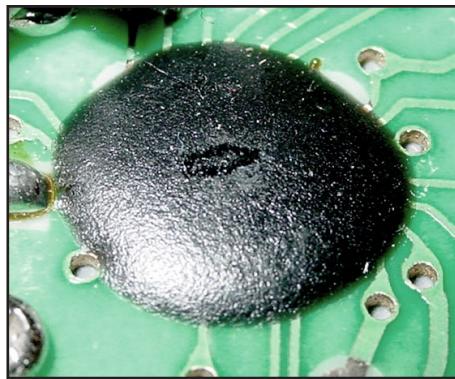


Figure E.24 COB Packaging

Proper IC positioning is indicated by a dot or square marking (known as a *key*) located on one end of the device (Figure E.25). Some devices mark pin 1 with an angled corner (such as with square package types like PLCC). On a circuit board, pin 1 is typically denoted by a square pad, whereas the rest of the IC's pads will be circular. Sometimes, a corresponding mark will be silkscreened or otherwise noted on the circuit board. Pin numbers start at the keyed end of the case and progress counter-clockwise around the device, unless noted differently in the specific product data sheet.

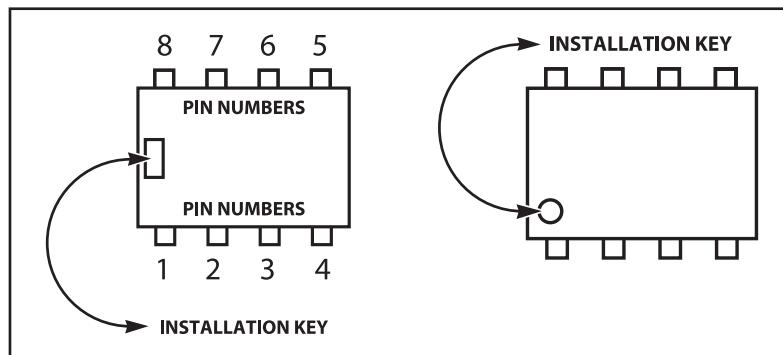


Figure E.25 IC Package Showing Pin Numbers and Key Marking

Microprocessors and Embedded Systems

A microprocessor—also known as a microcontroller or CPU (central processing unit), though there are slight technical differences—is essentially a general-purpose computer and is the heart of any embedded system. It is a complete computational engine fabricated on a single integrated circuit. In embedded systems, there is a union of hardware (the underlying circuitry) and software/firmware (code that is executed on the processor). You cannot have one without the other. Just about every electronic device you own can be considered an embedded system.

In 1971, Intel released the first microprocessor, the 4004. There are now thousands of microprocessors available, each with its own benefits and features, including:

- Cost
- Size
- Clock speed
- Data width (for example, 8-, 16-, or 32-bit)
- On-chip peripherals (such as on-chip memory, I/O pins, LCD control, RS-232, USB, wireless networking, analog-to-digital converters, or voltage references)

Common microprocessors include the Intel x86 family (used in most personal computers), Motorola 6800- and 68000-series (such as the 68020 or 68030 used in some Macintosh computers or the DragonBall MC68328 used in some Palm PDA devices), Zilog Z8, Texas Instruments OMAP, and Microchip PIC.

While we don't cover the specifics of various microprocessors here, their ubiquity inside hardware products should be noted. When you're hardware hacking or reverse-engineering a product, chances are that you will encounter a microprocessor of some type. But fear not: Microprocessor data sheets, usually available from the manufacturer, contain instruction sets, register maps, and device-specific details that will give you the inside scoop on how to operate the device. And, once you understand the basic theory of how microprocessors work and the low-level assembly language that they execute, it is fairly trivial to apply that knowledge to a new device or processor family.

Soldering Techniques

Soldering is an art form that requires proper technique in order to be done correctly. With practice, you will become comfortable and experienced with the process. The two key parts of soldering are good heat distribution and cleanliness of the soldering surface and component. In the most basic sense, soldering requires a soldering iron and solder. There are many shapes and sizes of tools to choose from (more details of which are available in Chapter 2). This section uses hands-on examples to demonstrate proper soldering and desoldering techniques.

Soldering Iron Safety

Improper handling of the soldering iron can lead to burns or other physical injuries. Wear safety goggles and other protective clothing when working with soldering tools. With temperatures hovering around 700 degrees F, the tip of the soldering iron, molten solder, and flux can quickly burn clothing and skin. Keep all soldering equipment away from flammable materials and objects. Be sure to turn off the iron when it is not in use and store it properly in its stand.

Hands-On Example: Soldering a Resistor to a Circuit Board

This simple example shows the step-by-step process to solder a through-hole component to a printed circuit board (PCB). We use a piece of prototype PCB and a single resistor (Figure E.26). Before you install and solder a part, inspect the leads or pins for oxidation. If the metal surface is dull, sand with fine sandpaper until it is shiny. In addition, clean any oxidation or excess solder from the soldering iron tip to ensure maximum heat transfer.

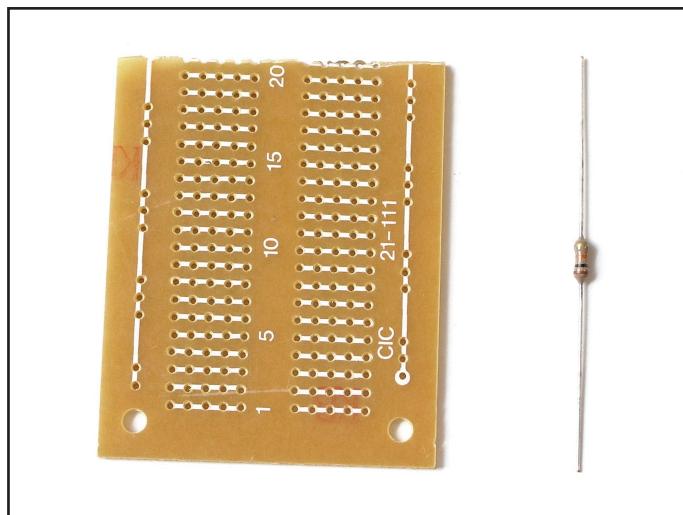


Figure E.26 Prototype PCB and Resistor

Bend and insert the component leads into the desired holes on the PCB. Flip the board to the other side. Slightly bend the lead you will be soldering to prevent the component from falling out when the board is turned upside-down (Figure E.27).

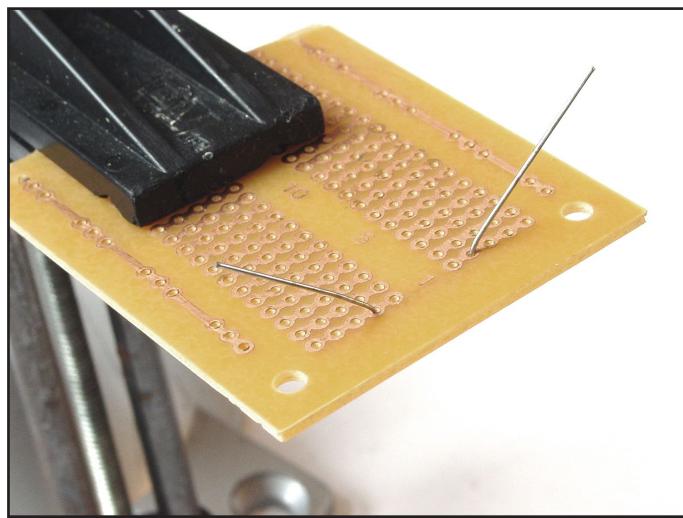


Figure E.27 Resistor Inserted into PCB

To begin the actual soldering process, allow the tip of your iron to contact both the component lead and the pad on the circuit board for about a second before feeding solder to the connection. This will allow the surface to become hot enough for solder to flow smoothly (Figure E.28).

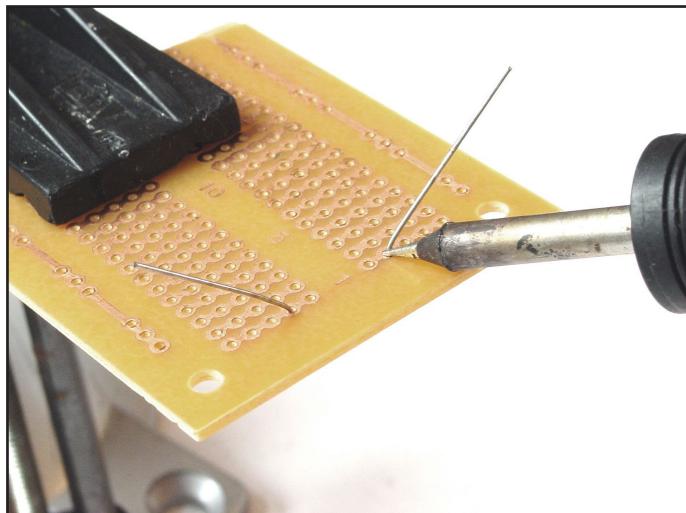


Figure E.28 Heating the Desired Solder Connection

Next, apply solder sparingly and hold the iron in place until solder has evenly coated the surface (Figure E.29). Ensure that the solder flows all around the two pieces (component lead and PCB pad) that you are fastening together. Do not put solder directly onto the hot iron tip before it has made contact with the lead or pad, as doing so can cause a cold solder joint. Soldering is a function of heat, and if the pieces are not heated uniformly, solder may not spread as desired. A cold solder joint will loosen over time and can build up corrosion, causing it to fail.

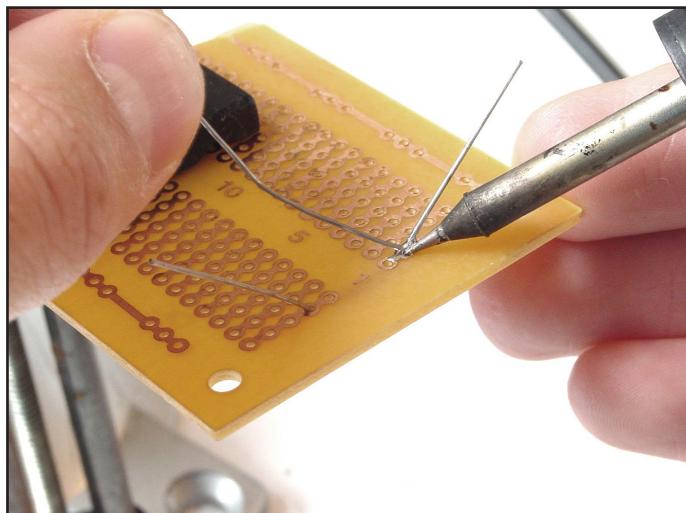


Figure E.29 Applying Heat and Solder to the Connection

When it appears that the solder has flowed properly, remove the iron from the area and wait a few seconds for the solder to cool and harden. Do not attempt to move the

component during this time. The solder joint should appear smooth and shiny, resembling the image in Figure E.30. If your solder joint has a dull finish, reheat the connection and add more solder if necessary.

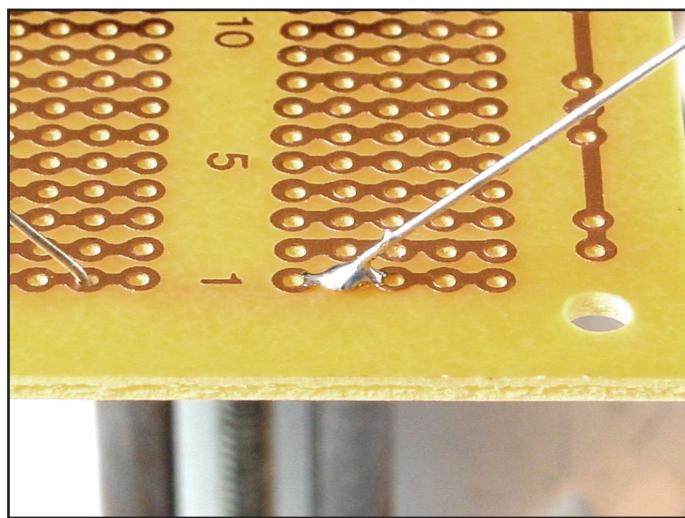


Figure E.30 Successful Solder Joint

Once the solder joint is in place, snip the lead to your desired length (Figure E.31). Usually, you will simply cut the remaining portion of the lead that is not part of the actual solder joint (Figure E.32). This prevents any risk of short circuits between leftover component leads on the board.

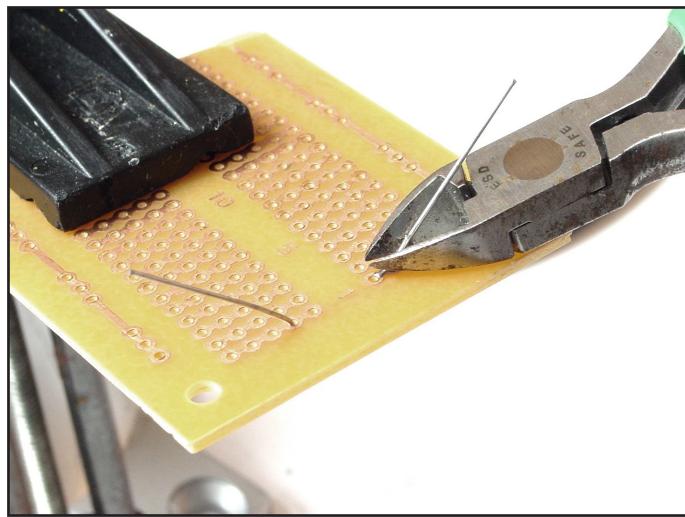


Figure E.31 Snipping Off the Remaining Component Lead

Every so often during any soldering session, use a wet sponge to lightly wipe the excess solder and burned flux from the tip of your soldering iron. This allows the tip to stay clean and heat properly. Proper maintenance of your soldering equipment will also increase its life span.

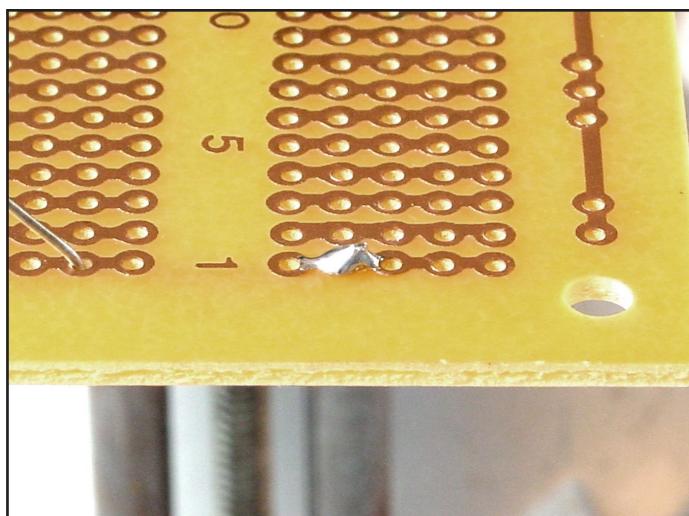


Figure E.32 Completed Soldering Example

Desoldering Tips

Desoldering, or removing a soldered component from a circuit board, is typically trickier than soldering, because you can easily damage the device, the circuit board, or surrounding components.

For standard through-hole components, first grasp the component with a pair of needle-nose pliers. Heat the pad beneath the lead you intend to extract and pull gently. The lead should come out, then repeat for the other lead. If solder fills in behind the lead as you extract it, use a spring-loaded solder sucker to remove the excess solder.

For through-hole ICs or multipin parts, use a solder sucker or desoldering braid to remove excess from the hole before attempting to extract the part. You can use a small flat-tip screwdriver or IC extraction tool to help loosen the device from the holes. Be careful to not overheat components, as they can become damaged and may fail during operation.

For surface mount devices (SMDs) with more than a few pins, the easiest method to remove the part is by using the ChipQuik SMD Removal Kit, as shown in the following step-by-step example. Removal of SMD and BGA devices is normally accomplished with special hot-air rework stations. These stations provide a directed hot-air stream used with specific nozzles, depending on the type of device to be removed. The hot air can flow freely around and under the device, allowing the device to be removed with minimal risk of overheating. Rework stations are typically priced beyond the reach of hobbyist hardware hackers, and the ChipQuik kit works quite well as a low-cost alternative.

Hands-On Example: SMD Removal Using ChipQuik

The ChipQuik kit (www.chipquik.com) allows you to quickly and easily remove surface mount components such as PLCC, SOIC, TSOP, QFP, and discrete packages. The primary component of the kit is a low-melting-point solder (requiring less than 300 degrees F) that reduces the overall melting temperature of the solder already on the SMD pads. Essentially, this enables you to just lift the part right off the PCB.

Figure E.33 shows the contents of the basic ChipQuik SMD Removal Kit, from top to bottom: alcohol pads for cleaning the circuit board after device removal, the special low-melting temperature alloy, standard no-clean flux, and application syringe.



Figure E.33 ChipQuik SMD Removal Kit Contents

Figure E.34 shows the circuit board before the SMD part removal. Our target device to remove is the largest device on the board, the Winbond WTS701EM/T 56-pin TSOP IC.



Figure E.34 Circuit Board Before Part Removal

The first step is to assemble the syringe, which contains the no-clean flux. Simply insert the plunger into the syringe and push down to dispense the compound (Figure E.35). The flux should be applied evenly across all the pins on the package you will be removing. Flux is a chemical compound used to assist in the soldering or removal of electronic components or other metals. It has three primary functions:

1. Cleans surfaces to assist the flow of filler metals (solder) over base metals (device pins).
2. Assists with heat transfer from heat source (soldering iron) to metal surface (device pins).
3. Helps in the removal of surface metal oxides (created by oxygen in the air when the metal reaches high temperatures).

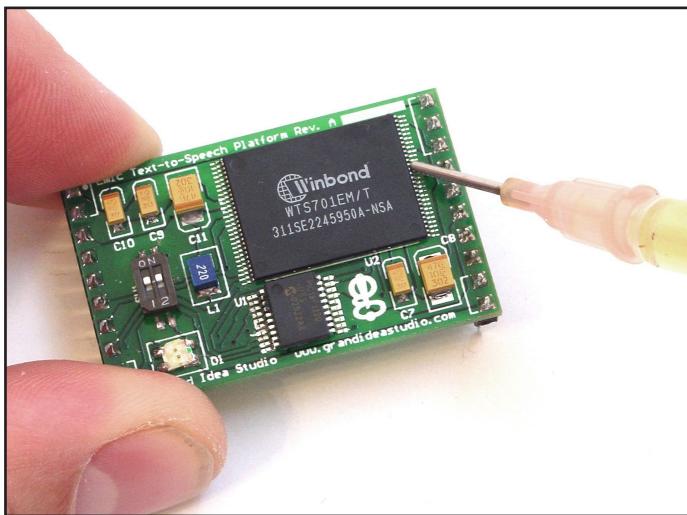


Figure E.35 Applying Flux to the Leads

Once the flux is evenly spread over the pins of the target device, the next step is to apply the special ChipQuik alloy to the device (Figure E.36). This step is just like soldering: Apply heat to the pins of the device and the alloy at the same time. You should not

have to heat the alloy with the soldering iron for very long before it begins to melt. The molten alloy should flow around and under the device pins (Figure E.37).

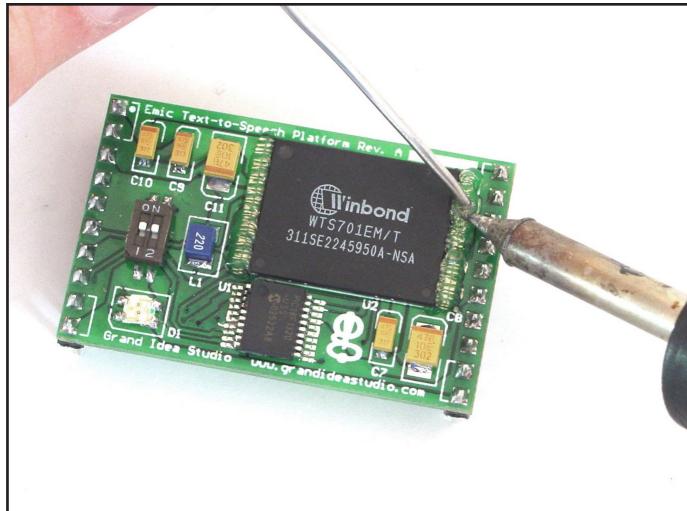


Figure E.36 Applying Heat and Alloy to the Leads



Figure E.37 Chip with Alloy Applied

Starting at one end of the device, simply heat and apply the alloy. Repeat for the other side(s) of the device. The flux will help with ensuring a nice flow of the alloy onto the device pins. Ensure that the alloy has come in contact with every single pin by gently moving the soldering iron around the edges of the device. Avoid touching nearby components on the PCB with the soldering iron.

Now that the alloy has been properly applied to all pins of the device, it is time to remove the device from the board. After making sure that the alloy is still molten by reheating all of it with the soldering iron, gently slide the component off the board with a tool such as a small jeweler's flat-blade screwdriver (Figure E.38). If the device is stuck, reheat the alloy and wiggle the part back and forth to help the alloy flow underneath the pads of the device and loosen the connections.

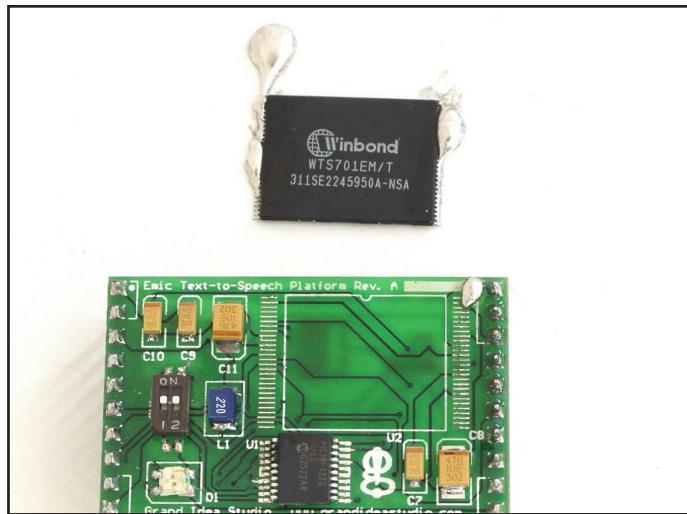


Figure E.38 Removing the Device from the Board

The final step in the desoldering process is to clean the circuit board. This step is important because it will remove any impurities left behind from the ChipQuik process.

First, use the soldering iron to remove any stray alloy left on the device pads or anywhere else on the circuit board. Then, apply a thin, even layer of flux to all of the pads that the device was just soldered to. Use the included alcohol swab or a flux remover spray to remove the flux and clean the area (Figure E.39).

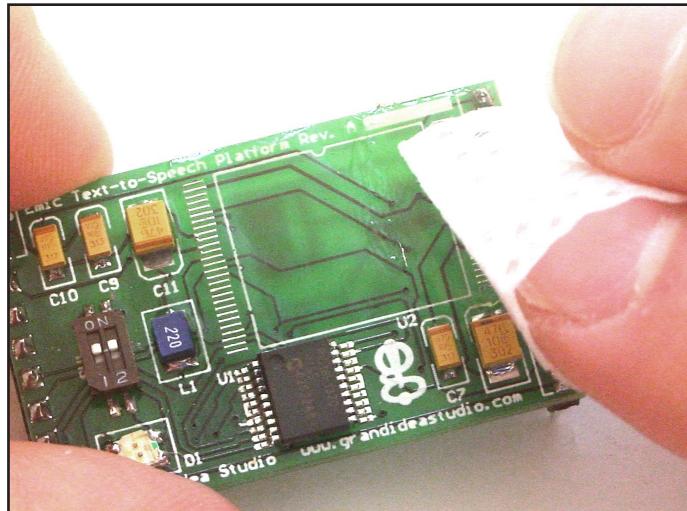


Figure E.39 Using Flux and Alcohol Swab to Clean Area

The desoldering process is now complete. The surface mount device has been removed and the circuit board cleaned (Figure E.40). If you intend to reuse the device you just removed, use the soldering iron to remove any stray alloy or solder left over on the pins and ensure there are no solder bridges between pins.

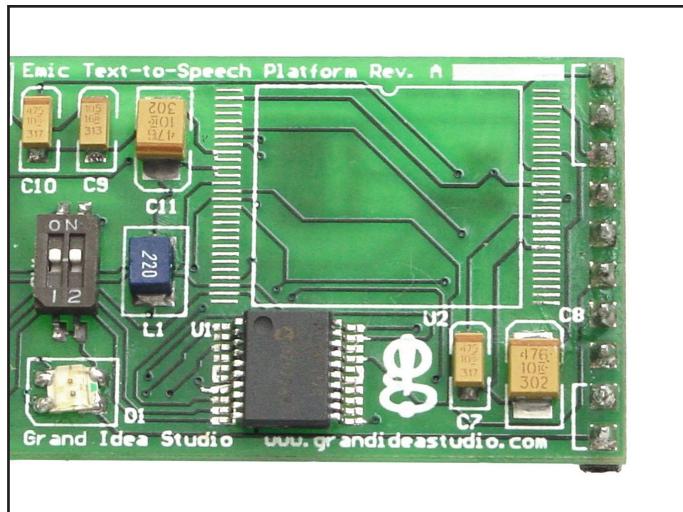


Figure E.40 Circuit Board with Part Successfully Removed

Common Engineering Mistakes

During engineering design and debugging, you should remember the important maxim KISS—Keep It Simple, Stupid—at all times. It can be frustrating to troubleshoot a problem for hours or days on end and then discover the cause was a simple oversight. The most common engineering mistakes for hobbyists are listed here. Although there are hundreds of other simple mistakes that can cause an engineer to quickly lose his or her hair, this list should get you started:

- **Faulty solder connections.** After soldering, inspect the connections for cold solder joints or bridges. Cold solder joints happen when you don't fully heat the connection or when metallic corrosion and oxide contaminate a component lead or pad. This is the most common mistake for amateur and hobbyist electronics builders. Solder bridges form when a trail of excess solder shorts pads or tracks together (see the "Soldering Techniques" section in this appendix).
- **Installing the wrong part.** Verify the part type and value before you insert and solder the component to the circuit board. Although many devices appear to look similar (e.g., a 1K and a 10K resistor look almost the same except for the color of one band), they have different operating characteristics and may act very differently in an electronic circuit. Surface-mount components are typically harder to distinguish from one another. Double-check to ensure that each part is installed properly. Keep in mind that the only way to properly test a component's value is to remove it from the board and then test it.
- **Installing parts backwards.** ICs have a notch or dot at one end indicating the correct direction of insertion. Electrolytic capacitors have a marking to denote the negative lead (on polarized surface mount capacitors, the positive lead has the marking). Through-hole capacitors also have a shorter-length negative lead than the positive lead. Transistors have a flat side or emitter tab to help you identify the correct mounting position and are often marked to identify each pin. Diodes have a banded end indicating the cathode side of the device.

- **Verify power.** Ensure that the system is properly receiving the desired voltages from the power supply. If the device uses batteries, check to make sure that they have a full charge and are installed properly. If your device isn't receiving power, chances are it won't work.

Web Links and Other Resources

General Electrical Engineering Books

- Radio Shack offers a wide variety of electronic hobby and how-to books, including an Engineer's Notebook series that provide an introduction to formulas, tables, basic circuits, schematic symbols, integrated circuits, and optoelectronics (light-emitting diodes and light sensors). Other books cover topics on measurement tools, amateur radio, and computer projects.
 - *Nuts & Volts* (www.nutsvolts.com) and *Circuit Cellar* (www.circuitcellar.com) magazines are geared toward both electronics hobbyists and professionals. Both are produced monthly and contain articles, tutorials, and advertisements for all facets of electronics and engineering.
 - Horowitz and Hill, *The Art of Electronics*, Cambridge University Press, 1989. Essential reading for basic electronics theory. It is often used as a course textbook in university programs.
 - C. R. Robertson, *Fundamental Electrical & Electronic Principles*, Newnes, 2001. Covers the essential principles that form the foundations for electrical and electronic engineering courses.
 - M. M. Mano, *Digital Logic and Computer Design*, Prentice-Hall, 1979. Digital logic design techniques, binary systems, Boolean algebra and logic gates, simplification of Boolean functions, and digital computer system design methods.
 - K. R. Fowler, *Electronic Instrument Design*, Oxford University Press, 1996. Provides a complete view of the product development life cycle. Offers practical design solutions, engineering trade-offs, and numerous case studies.

Electrical Engineering Web Sites

- **ePanorama.net:** www.epanorama.net A clearinghouse of electronics information found on the Web. The content and links are frequently updated. Copious amounts of information for electronics professionals, students, and hobbyists.
- **The EE Compendium, The Home of Electronic Engineering and Embedded Systems Programming:** <http://ee.cleversoul.com> Contains useful information for professional electronics engineers, students, and hobbyists. Features many papers, tutorials, projects, book recommendations, and more.
- **Discover Circuits:** www.discovercircuits.com A resource for engineers, hobbyists, inventors, and consultants, Discover Circuits is a collection of over 7,000 electronic

circuits and schematics cross-references into more than 500 categories for finding quick solutions to electronic design problems.

- **WebEE, The Electrical Engineering Homepage:** www.web-ee.com Large reference site of schematics, tutorials, component information, forums, and links.
- **Electro Tech Online:** www.electro-tech-online.com A conununity of free electronic forums. Topics include general electronics, project design, microprocessors, robotics, and theory.
- **University of Washington EE Circuits Archive:** www.ee.washington.edu/circuit_archive A large of collection of circuits, data sheets, and electronic-related software.

Data Sheets and Component Information

When reverse-engineering a product for hardware hacking purposes or reusing parts, identifying components and device functionality is typically an important step. Understanding what the components do may provide detail of a particular area that could be hacked. Nearly all vendors post their component data sheets openly on the Web, so simple searches will yield a decent amount of information. The following resources will also help you if the vendors don't:

- **Data Sheet Locator:** www.datasheetlocator.com A free electronic engineering tool that enables you to locate product data sheets from hundreds of electronic component manufacturers worldwide.
- **IC Master:** www.icmaster.com The industry's leading source of integrated circuit information, offering product specifications, complete contact information, and Web site links.
- **Integrated Circuit Identification (IC-ID):** www.elektronikforurn.de/ic-id Lists of manufacturer logos, names, and datecode information to help identifying unknown integrated circuits.
- **PartMiner:** www.freetradezone.com Excellent resource for finding technical information and product availability and for purchasing electronic components.

Major Electronic Component and Parts Distributors

- Digi-Key, 1-800-344-4539, www.digikey.com
- Mouser Electronics, 1-800-346-6873, www.mouser.com
- Newark Electronics, 1-800-263-9275, www.newark.com
- Jameco, 1-800-831-4242, www.jameco.com

Obsolete and Hard-to-Find Component Distributors

When trying to locate obscure, hard-to-find materials and components, don't give up easily. Sometimes it will take hours of phone calls and Web searching to find exactly what you need. Many companies that offer component location services have a minimum order (upwards of \$100 or \$250), which can easily turn a hobbyist project into one collecting dust on a shelf. Some parts-hunting tips:

- Go to the manufacturer Web site and look for any distributors or sales representatives. For larger organizations, you probably won't be able to buy directly from the manufacturer. Call your local distributor or representative to see if they have access to stock. They will often sample at small quantities or have a few-piece minimum order.
- Be creative with Google searches. Try the base part name, manufacturer, and combinations thereof.
- Look for cross-reference databases or second-source manufacturers. Many chips have compatible parts that can be used directly in place.

The following companies specialize in locating obsolete and hard-to-find components. Their service is typically not inexpensive, but as a last resort to find the exact device you need, these folks will most likely find one for you somewhere in the world:

- USBid, www.usbid.com
- Graveyard Electronics, 1-800-833-6276, www.graveyardelectronics.com
- Impact Components, 1-800-424-6854, www.impactcomponents.com
- Online Technology Exchange, 1-800-606-8459, www.onlinetechx.com

Index

A

Absolute addressing 181
Absolute Indexed addressing 186
Absolute Indexed addressing with X 181
Absolute Indexed addressing with Y 182
Absolute Indexed Indirect addressing 182
Accumulator 173, 174, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 189, 190, 191, 192, 194, 195, 197, 198, 200, 214, 215, 247, 248, 249, 250, 252, 254, 255, 256, 259, 260
Accumulator Shift Left 200
ADC 198
Add Memory to Accumulator with Carry 198
address bus 205, 206, 208, 211, 212, 219, 220, 221, 223, 225
address lines 208, 212, 219, 221, 223, 225, 227
Altair 20, 21, 37, 41
alternating current 46, 241, 248, 272, 273, 278, 281, 282
AND 56, 73, 74, 75, 76, 77, 78, 79, 80, 81, 84, 86, 87, 88, 135, 137, 143, 154, 155, 156, 157, 158, 159, 160, 195, 196, 197, 220, 221, 225, 240, 244, 245, 254
AND Memory with Accumulator 195
Applefritter 40, 123, 124, 167, 236
Apple II 25, 28, 34, 35, 37, 39, 40, 41, 59,

97, 127, 233
Apple I Owners Club 25, 26, 27, 28, 30, 40, 123, 167, 171, 201, 217, 236
Arithmetic and Logic Unit 215
array 138, 139
ASCII 20, 30, 59, 97, 98, 99, 111, 115, 117, 130, 174, 179, 185, 186, 187, 189, 191, 195, 198, 205, 232, 233, 234, 235, 237, 266, 268, 269
ASL 200, 201
assembler 39, 42, 171, 172, 174, 177, 185
Assembly (programming language) 22, 30, 119, 145, 169, 171, 176
ATmega 118, 233, 234, 235
AUTO 145
avalanche breakdown 282, 283

B

Ball Grid Array 287, 293
BASIC 20, 21, 22, 29, 30, 33, 35, 39, 42, 127, 128, 130, 131, 132, 134, 135, 136, 137, 140, 143, 144, 145, 150, 151, 156, 167, 169, 173, 174, 176, 177, 217, 218
Baum, Allen 27, 41
BEQ 185
binary 93, 94, 103, 104, 144, 189, 190, 193, 194, 198, 199, 220, 225, 265, 266, 299
Binary Coded Decimal 198
Boolean algebra 84, 86, 299
Borrill, Ray 27, 32

BPL 176, 177
Branch on Minus 189
Branch on Result Not Zero 189
Branch on Result Zero 185
breadboard 48, 49, 64, 67, 69, 74, 91
breakdown/reverse voltage 282
Briel Computers 57, 110, 121, 160, 217, 233, 234, 235
bus 22, 27, 34, 97, 205, 206, 208, 211, 212, 219, 220, 221, 223, 225, 226, 230
Byte Magazine 20, 25, 30, 32, 33, 190, 194
Byte Shop 20, 30, 33

C

CALL 145, 146
capacitor 37, 72, 116, 222, 270, 271, 277, 278, 279, 280, 281
carry flag 198, 199, 200, 201
cassette 20, 21, 22, 30, 32, 33, 34, 41, 42, 128, 217
cassette interface 20, 30, 33, 41, 42, 128
chip 28, 29, 56, 57, 58, 74, 75, 101, 119, 121, 204, 205, 206, 207, 208, 210, 212, 219, 220, 221, 222, 223, 224, 225, 227, 228, 234, 235, 287, 289
Chip Enable 219
Chip-on-Board 287
ChipQuik SMD Removal Kit 293, 294, 295, 297
chip straightener 58
CHOICE 156
CLC 199
Clear Carry Flag 199
clock 92, 96, 97, 100, 101, 117, 118, 206, 208, 209, 211, 221, 222, 231, 233, 235
CLR 145
CMP 185, 189
cold solder joint 109, 291, 298

Compare 185
composite video 60, 235
Computer Playground 38, 40
CPX 185
CPY 185, 189
crystals 111, 117, 118, 271
current 41, 67, 68, 69, 181, 185, 212, 214, 215, 236, 270, 272, 273, 274, 275, 276, 277, 278, 281, 282, 283, 285

D

Data Available line 226, 231
data bus 205, 206, 208, 211, 212, 219, 220, 223, 225, 226
Data Domain 32, 33, 34
decoder 104, 224, 225
DEL 145
DeMorgan's Laws 84, 86, 87, 88, 105, 220, 225
desolder 53, 289, 293, 297
digital logic 63, 72, 84, 93, 106
DIM 139
diode 69, 112, 113, 235, 270, 272, 281, 282, 283, 284, 285
direct current 46, 67, 272, 273, 278, 281
Drennan, Richard 25, 29
DSP 228, 229, 230

E

Echo 174, 191, 194
EEPROM 111, 123, 224, 225, 226
ELSE 134
Enable, Write 225
encoder 103
EOR 195, 197
EPROM 29, 223, 224
Exclusive-OR Memory with Accumulator 195

F

farad 278
 Felsenstein, Lee 32, 41
 Fish, Steve 27, 38, 39, 40
 Flip-Flop 56, 91, 92, 97
 flux 289, 292, 294, 295, 296, 297
 Focal (programming language) 25
 FOR 133, 139
 Fortran 42
 forward current 282
 forward voltage 282

G

GetChar 177
 GOSUB 136, 174
 GOTO 130, 134, 136, 173, 176
 graphics 25, 29, 35, 234, 266

H

Hamfest 55
 Hatfield, Fred 25
 Hello World 128, 130, 173, 174, 176, 178, 179
 hexadecimal 30, 93, 94, 143, 144, 170, 171, 179, 185, 225, 266
 HexEdit 173, 175, 186, 187, 188
 HIMEM 145, 160
 Homebrew Computer Club 25, 29, 35, 37, 41, 42

I

IC extraction tool 57, 293
 IEC marking 279
 IF/THEN 134
 Immediate addressing 180
 Implied addressing 180
 INPUT 131, 153
 integrated circuit 56, 57, 58, 74, 77, 78, 79,

81, 83, 92, 102, 111, 118, 121, 123, 222, 225, 285, 286, 287, 288, 293, 294, 300

Interface Age Magazine 20, 25, 27, 30

interrupt request 211

inverter 76, 78, 80, 86, 87, 101, 223, 225

itty bitty machine company 32, 33

J

JMP 171, 173, 176
 Jobs, Steve 33, 34, 41
 JSR 173, 174, 176
 jumper 36, 64, 117, 120, 233

K

Kbd 177
 KbdRdy 177
 keyboard 20, 30, 33, 34, 36, 39, 40, 59, 111, 115, 117, 120, 121, 128, 176, 177, 204, 226, 228, 230, 231, 232, 233, 234
 KIM-1 22, 29, 31

L

LDA 171, 173, 174, 176, 184, 189, 205
 LDX 184
 LDY 184
 LEN 140, 141, 142
 light-emitting diode 31, 69, 70, 75, 76, 78, 79, 90, 270, 271
 LIST 145
 Logical Shift Right 200
 logic expressions 84, 86
 logic probe 48, 66, 91
 LOMEM 145
 LO*OP Center 35, 36, 37, 38
 Loop, Liza 27, 35
 LSR 200, 201

M

McCAD 57, 96, 107, 123, 124, 125

McMenomy, Sarah 147
microcontroller 233, 234, 235, 288
MOD 132
Monitor (program) 21, 29, 59, 60, 170, 171, 172, 173, 174, 175, 178, 179, 186, 187, 188, 190, 193, 196, 200, 203, 216, 223, 226, 227
multimeter 46, 47, 48, 64, 67, 71, 112

N

NAND 56, 76, 77, 78, 80, 86, 87, 88, 91, 135, 225
negative flag 189
Nelson, Larry 25, 26, 27, 28, 29, 30, 32
NEXT 133, 139
nibble 97, 265, 266
non-maskable interrupt 211
NOR 56, 79, 80, 81, 87, 90, 135
NOT 135
NULL character 186
NVRAM 123

O

Ohm's Law 274
OR 135
ORA 195, 196, 197
OR Memory with Accumulator 195
Otaguro, Dr. Will 39
Output Enable 220, 221, 222
overflow flag 212

P

Parallax Propeller 234
parallel port 22, 25
part designator 270
peak-to-peak voltage 273
peak voltage 273
PEEK 143

peripheral interface adapter 226
PHA 192
PHP 192, 193
PLA 192
PLP 192, 193
POKE 143
popping (stack function) 191, 192, 215, 216
power supply 20, 30, 33, 34, 36, 40, 49, 54, 55, 64, 68, 69, 72, 272, 299
PRINT 128, 129, 134, 151, 153, 155, 156
PrintBin 190, 193
printed circuit board 57, 123, 125, 287, 290, 291, 294, 296
printer 22, 25, 31, 39, 43, 122
processor status register 215
Program Counter 181, 192, 213, 214
PS/2 59, 111, 115, 117, 118, 119, 121, 128, 233
Pull Accumulator from Stack 192
Pull Processor Status from Stack 192
Push Accumulator on Stack 192
pushing (stack function) 191, 215, 216
Push Processor Status on Stack 192

R

Read/Write line 221
Ready line 211
register 96, 100, 173, 180, 187, 189, 191, 192, 193, 194, 197, 208, 213, 214, 215, 227, 228, 229, 230, 231, 234, 235, 289
Relative addressing 181
Replica I 57, 59, 107, 108, 110, 111, 112, 114, 120, 121, 123, 124, 125, 128, 145, 170, 217, 218, 222, 224, 234, 235
Replica I TE 110, 234
resistance 46, 69, 71, 273, 274, 275, 276, 280, 282
resistor 69, 71, 108, 112, 270, 271, 272,

275, 277, 280, 283, 284, 290, 298
RETURN 136, 153, 176
 reverse recovery time 282
 RF modulator 20, 30
RND 133
 root-mean-square voltage 273
ROR 195
 Rotate Left 190
 Rotate Right 190
RTS 176, 192
RUN 145

S

S-100 22, 27, 34
SBC 200
 Schawlow, Dr. Arthur 25, 26, 29
SCR 145
SEC 199, 200
 serial I/O board 107, 121, 122
 Set Carry Flag 199
 set-reset latch 89
 shift register 100, 234, 235
 6502 processor 27, 28, 29, 31, 34, 111, 124, 173, 179, 192, 198, 201, 209, 210, 211, 212, 213, 214, 216
 socket 58, 64, 66, 111, 114, 115, 119, 121, 122
 solder 52, 53, 57, 108, 109, 112, 114, 118, 287, 289, 290, 291, 292, 293, 294, 295, 297, 298
 soldering iron 43, 51, 108, 289, 290, 292, 295, 296, 297
 Sonoma County Computer Club 35, 36
SRAM 111, 123
STA 184
 stack 191, 192, 193, 194, 214, 215, 216
 stack pointer 192, 214, 215, 216
 Stanford University 25

STEP 133
strings 30, 130, 131, 139, 140, 141, 142, 143, 146, 150, 151, 178, 185, 186, 187, 188, 190, 194, 195, 196
STX 184
STY 184
 substrings 140, 142
 Supplemental Software package 57, 124, 160, 171, 173
 surface-mount device 293, 294
 Sync line 212

T

TAB 129, 154
Teletype 20, 32, 34, 39, 234, 235
 time constant 281
 Torzewski, Joe 25, 27, 28, 31, 217
 Transfer Index X to Stack Register 192
 Transfer Stack Pointer to Index X 192
 transistor 222, 283, 284, 285
 tri-state buffer 101
 truth table 73, 104, 105
TSX 192
TV Typewriter 176, 177, 179
 Two's Complement 144, 189
TXS 192

V

video connector 118
 video monitor 20, 204, 235, 236
 Vintage Computer Festival 37
 voltage 46, 55, 67, 68, 69, 70, 72, 74, 75, 76, 212, 236, 265, 272, 273, 274, 276, 278, 279, 280, 281, 282, 283, 284, 285, 289

W

wire-wrap 50
 word 33, 96, 266
 Wozniak, Steve 19, 25, 35, 36, 37, 39, 41,

42, 217

Write Enable 219, 221, 222

X

XOR 56, 82, 83, 84, 87, 88, 89

X (register) 173, 178, 187, 191

Y

Y (register) 173, 178, 187, 189, 191, 200

Z

Zero flag 189

Zero Page addressing 181

Zero Page Indexed addressing 182

Zero Page Indexed addressing with Y
182

Zero Page Indexed Indirect addressing
183

Zero Page Indirect Indexed addressing
187

Zero Page Indirect Indexed addressing
with Y 183

Foreword Contributor

Steve Wozniak, a Silicon Valley icon and philanthropist for the past three decades, Founder, Chairman and CEO of Wheels of Zeus (wOz), helped shape the computing industry with his design of Apple's first line of products (the Apple I and II) and influenced the popular design of the Macintosh. For his achievements at Apple Computer, Wozniak was awarded the National Medal of Technology by the President of the United States in 1985, the highest honor bestowed upon America's leading innovators.

In 2000 Wozniak was inducted into the Inventors Hall of Fame and was awarded the prestigious Heinz Award for Technology, the Economy and Employment for "single-handedly designing the first personal computer, and for then redirecting his lifelong passion for mathematics and electronics toward lighting the fires of excitement for education in grade school students and their teachers."

Making significant investments of both his time and resources in education, Wozniak "adopted" the Los Gatos School District, providing students and teachers with hands-on teaching and donations of state-of-the-art technology equipment. He was the founding sponsor of the Tech Museum, Silicon Valley Ballet, and Children's Discovery Museum of San Jose.

Author

Tom Owad is owner of Schnitz Technology, an IT consultancy providing Mac OS X and Linux support in south-central Pennsylvania. He serves on the board of directors of the Apple I Owners Club, where he is also webmaster and archivist. Owad is the owner and webmaster of Applefritter, an Apple-centric community of artists and engineers. Applefritter provides its members with discussion boards for the exchange of ideas and hosts countless member-contributed hardware hacks and other projects. He holds a BA in computer science and international affairs from Lafayette College, PA.

Technical Editor

John Greco is a professor of electrical and computer engineering at Lafayette College, where he has taught digital circuit and system design for over 30 years. He holds a Ph.D. in electrical engineering from the City University of New York. In addition, Greco has taught at the University of Petroleum and Minerals in Saudi Arabia. He has worked for GTE-Sylvania and has performed consulting work for the former Bell Laboratories and Moore Products.