

"Ed Nisley is a true national treasure."
— Steve Ciarcia, Publisher, *Circuit Cellar INK*

DISK INCLUDED!

The Embedded PC's ISA Bus:

Firmware, Gadgets, and Practical Tricks

Ed Nisley, P.E.



point on the LCD panel
ays uses the buffer pointer

```
PUBLIC      C LCDSetDot  
PROC       LCDSetDot  
ARG        Row:WORD, Col:WORD  
USES       ES, DI
```

```
MOV        AX, [Col]  
MOV        BX, [Row]  
CALL       LCDMakeAddr
```

```
ES         AX, [pNewBuf]  
DI, AX
```

```
MOV        BX, 0101h  
HX, CL  
BL
```

```
TEST       [Value], 0001h  
JNZ        @@2  
AND        [ES:DI], BL  
JMP        SHORT @@3
```

The Embedded PC's ISA Bus

Firmware
Gadgets
and
Practical Tricks

The Embedded PC's ISA Bus

Firmware Gadgets and Practical Tricks

Edward Nisley, PE



Peer-to-Peer CommunicationsSM
San Jose, California
<http://www.peer-to-peer.com>

The Embedded PC's ISA Bus: Firmware, Gadgets, and Practical Tricks

Copyright ©1997 Edward Nisley. All rights reserved.

No part of this book may be reproduced, in any form or by any means, without prior written permission from the publisher.

Published by Peer-to-Peer Communicationssm, Inc.

P.O. Box 640218, San Jose, California 95164-0218, U.S.A.

Phone: 800-420-2677 Fax: 408-435-0895

E-mail: info@peer-to-peer.com Web: <http://www.peer-to-peer.com>

Thanks to: Pat KE4FCJ, John KF4EUZ, Rick KA3PSK, Bobby Emory

Cover design: Gordon Haber

Production advice: Mayapriya Long, Bookwrights, Charlottesville, Virginia

Printing: Data Reproductions

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10

ISBN 1-57398-017-X

Library of Congress Cataloging in Publication Data

Nisley, Edward, 1953-

The embedded PC's ISA bus : firmware, gadgets, and practical tricks / Edward Nisley.

p. cm.

Includes bibliographical references and index.

ISBN 1-57398-017-X (paper)

1. Embedded computer systems--Design and construction.

2. Microcomputers--Buses 3. Computer firmware. 4. Expansion boards (Microcomputers)--Design and construction. I. Title

TK7895.E42N57 1997

004.64--dc21

97-12572

This book and diskette are provided "as is." The implied warranties of merchantability and fitness for a particular purpose are expressly disclaimed. The program code contained herein has not been thoroughly tested under all conditions, therefore its reliability, serviceability, and function are not guaranteed.

The information contained herein was valid at the time it was written and is conveyed as accurately as possible by the author. However, some information may be incorrect or may have changed prior to publication. The author and publisher make no claims that the material contained herein is entirely correct, make no express or implied warranty of any kind, and assume no liability for incidental or consequential damages in connection with or arising out of the use of the information or programs in this book or on the supplied diskette.

All trademarks used herein are for identification only and remain the property of their owners.

Peer-to-Peer Communications offers discounts on this book when ordered in bulk quantities. For more information, or to receive a catalog, please contact the publisher at the address above.

Peer-to-Peer Communications' books are distributed to bookstores worldwide through

International Thomson Publishing, Inc. In the U.S.A. and Canada, contact ITP at:

International Thomson Publishing

Customer Service

7625 Empire Drive

Florence, Kentucky 41042-2978

For the addresses of International Thomson Publishing offices in other countries, please contact Peer-to-Peer Communications.

Dedicated to

Steve Ciarcia

*for showing all of us
how it's done.*

Contents

Foreword	xi
0: Getting Started	1
What's Firmware?	1
Hosts and Targets	2
(Hardware) Construction Ahead	3
Logic Levels and Pin Names	4
Some Assembly (Language) Required	5
What Else Do You Need?	5
Numeric Representation	6
Reference Material	7
Errata and Updates	7
Typography	8
1: Blindsided by Technology	9
Commodity Computers	9
Essential Hardware	10
Bootting Firmware	11
The Boot's Strap	12
The First Program	16
Release Notes	18
2: Chickens and Eggs	21
The Bare Essentials	21
Load and Go	23
Micro-C Startup	26
Console I/O	27
Say "Hello!"	32
Release Notes	34
3: Barnacles on the ISA Bus	35
One Byte at a Time	36
ISA Board Basics	39
Buffering the Bus	41
Access Control	43
Essential I/O	45
Stepwise Construction	45
Delayed I/O	48
Release Notes	51
Bus Labels	52

4: I/O Time	53
Where Does Time Come From?	53
I/O Addressing	57
Timing from Both Sides	57
Count the Ways...	62
Hardware Checkout	64
Release Notes	69
5: After This Brief Interruption	71
The Inside Story	71
The Rest of the Story	74
Collision Alarm!	79
Three Timers Ticking	81
Digital Readout	85
A Cautionary Tale	89
Release Notes	92
6: Memories Are Made of This.....	93
Where Does Memory Come From?	93
Confronting the ISA Slows	95
Writing to EEPROM	100
The BIOS Connection	106
BIOS Extension Hints & Tips	109
Release Notes	111
7: Absolute Power Corrupts	113
Prepare to RAM	113
Backup Warning	117
Data Defense	120
Processor Protection	122
Firmware Supervision	123
Getting Down to Code	125
Unmasking the NMI	129
Release Notes	131
8: Ticks, Pops, and Restarts	133
The Key to the Code	133
Extension Essentials	133
The Case of the Missing PSP	134
Capturing Interrupts	138
Failing Power	139
Resets and the Worst Hack	142
Release Notes	145

9: IDs and LCDs	147
Simple Circuitry	147
Bidirectional Bit Banging	150
Hardware Timing	153
Checking Your ID	157
LCDs Redux	158
Shared Bits	161
Release Notes	162
10: Booting C from ROM	163
Basic BIOS Booting	163
Modeling Memory	164
Sequenced Startup	167
Getting the Boot	174
Release Notes	178
11: Beyond Small	179
Along the Mainline	180
Jumping the Mainline Track	182
Embedding a Sieve	184
Debugging by Wire	188
The Second-Worst Hack	190
Release Notes	192
12: Lots'a Dots	193
Dots in Ranks and Rows	194
Seeing the Big Picture	202
Variations on a Theme	205
Release Notes	210
13: Bitmapped LCD Panel Hardware	211
Counting Clocks	211
A Chain of Dots	217
Off by One...	220
Porting a RAM	224
Practical Power	226
Greater & Lesser Arrays	227
Release Notes	228
14: Testing the Graphic LCD Panel	229
Testing...	229
...Testing! One...	231
... Two...	237
... Four...	239

... and More!	240
Release Notes	242
15: Bringing the Graphic LCD Panel to Life.	243
Putting Life to Work	243
Doing Dots	245
Variations on a Theme	250
Stashing Slides	252
Release Notes	256
16: All Text Is Graphics	257
Filching Fossil Fonts	257
Divvying up the Dots	262
Gazing in the Glass	265
Assume the Position...	268
The Rest of the Stories	272
The Case of the Capital "T"	272
Release Notes	275
17: For Further Study.	277
Crossing the 1 MB Line	277
Direct Memory Access	278
Busmastering Boards	279
Release Notes	280
Sources	281
Semiconductor Manufacturers	281
Publications	282
Hardware	284
Software	285
Surplus Electronics	286
Bibliography	287
Acronyms and Abbreviations	293
Schematics	295
Port Addresses	329
Control and Status Bits	329
Memory Addresses	330
Bill of Materials	331
Index.	335

Foreword

By Jeff Duntemann KG7JF

The first computer program I wrote for a machine of my own didn't add, or display text, or crunch data. It turned on an LED. This was what we might call an omen. Or maybe imprinting.

The year was 1976, and the machine was a 6"×8" piece of perfboard with a couple of chips and some toggle switches on it and a rat's nest of wire-wrapping underneath. It was called a COSMAC Elf, and it had been published as a do-it-yourself project in *Popular Electronics* magazine in April of that year. It had 256 bytes of memory. Bytes—not kilobytes or megabytes or (lord knows) gigabytes. It wasn't much of a number cruncher. But boy, could it control LEDs.

Over the next several years I lashed together numerous COSMAC machines and branched out beyond LEDs into controlling motors, relays, and then (haltingly) graphics patterns on a TV screen. In 1981 I submitted an article to *Popular Electronics* describing the COSMAC IMP (Inexpensive Matrix Printer) which was a little cash-register thermal printhead/motor module with a simple parallel interface to the Elf. Alas, said *Popular Electronics*, the age of Elves is past us. But if I had anything on this brand new IBM PC thing, well, they'd fall at my feet.

And thus the path forked: Down *this* road went the PC, to revolutionize the whole data processing the industry, and down *that* road went the embedded machines, with exotic processors and built-in PROM, spinning steppers and winking LEDs inside everything electronic that *wasn't* a PC. Like most of us, I had to choose, and looking back over my shoulder said bye-bye to a process I actually enjoyed very much. After all, I'd been doing electronics when computers cost a gazillion dollars and were never seen by mere mortals. *Beware of programmers with screwdrivers*, so all the management handbooks went. With that in mind I entered the PC age, and left my screwdriver at the door.

I've been basically living the PC since 1982, without my screwdriver, and without the time to remember too hard what I'd been doing in those golden Seventies. And since 1982 something very significant has happened: The PC has eaten its way both up and down the food chain, devouring typewriters at the bottom and minicomputers at the top. It's now rapidly encroaching on Big Iron turf up on the high side, and—strangest of all strange things!—swallowing the embedded systems market whole.

The almost unimaginable scale on which PCs and their components are manufactured has sent prices through the floor, to the point where midrange and high-end embedded work can be done economically on PC compatible devices. The PC architecture is cheap, fast, and almost infinitely versatile. It's well-documented, and the available software works well and can be had for a song.

So the two paths have converged again. If you understand the PC, you can pick up your screwdriver again and not have to learn a new architecture or instruction set. All you really need to learn is the way to hook those LEDs (and LCDs, and what have you) to the PC's well-understood bus. Ed can teach you that. You may be a little surprised (especially if you've spent the last fifteen years coding in Basic) at what the PC actually does at the chips-and-memory level—but that's all here as well.

There may come a day when Windows CE (or whatever they call it this week) greets you from an LCD on the side of your toaster. But until then, it's you, the bits, and the bus. If there's ever been a better book than this for developing those new skills (or scraping the rust off some very old skills) I've yet to see it.

Good luck. Don't forget your screwdriver.

Jeff Duntemann KG7JF
Scottsdale, Arizona
March, 1997

0 Getting Started

This book is a guide for people who want to apply the IBM PC's Industry Standard Architecture (ISA) bus to their projects. Quite simply, that includes anyone interested in building PC hardware, writing PC firmware, or interfacing a PC with the real world. If you've gotten this far, that means *you!*

- **Engineers** designing embedded PC hardware
- **Programmers** writing firmware to drive that hardware
- **Technicians** debugging recalcitrant ISA bus gadgets
- **Students** relating abstract coursework to practical applications
- **Enthusiasts** controlling projects with computers

Regardless of whether you consider yourself a *hardware* or *software* person, you'll benefit by learning some tricks and techniques from the *other* side:

- Recognize fundamental ISA bus speed limits
- Follow hardware interrupt signals through the PC to your software routine
- Understand how and why watchdog timers protect your code
- Fix hardware problems with firmware (yes, indeed)
- Discover how to boot programs directly from disk or EPROM
- Write code that runs automatically before the PC boots
- Debug and solve truly perplexing problems!

I emphasize the value of hands-on experience gained by *building* a hardware gadget, *writing* some code to control it, *measuring* the system's performance, and *understanding* how the hardware and firmware work together. Although you can certainly enjoy this book in armchair-traveller mode, you'll derive far more benefit by rolling up your sleeves and working through the pages.

You can also use this book as an introduction to hardcore embedded PC development using off-the-shelf PC/104 hardware. Essentially everything you learn about the ISA bus in this book applies directly to the PC/104 bus. In fact, you can run many of your own programs directly on a standard PC while debugging them, then transfer them directly to a PC/104 system without any changes.

What's Firmware?

Firmware is software that controls the bare silicon, down below the operating system, down on the bit-twiddling level, where microseconds matter and high level languages dare not tread. It implements the most fundamental software functions of all, the strange routines that make the hardware work. You must leave behind

those familiar interfaces and programming habits that depend on operating system support and human reaction time. Firmware, at least the firmware you'll meet here, operates under entirely different constraints than normal PC application software.

For example, you'll write programs that run directly from a diskette without DOS or generate characters on an LCD panel without calling the BIOS. You'll learn how to boot a program, without a disk, from nonvolatile memory out on the ISA bus. And you'll do this on a PC that lacks a keyboard and video monitor, no less.

Once you understand how firmware works on this level, you can apply the same techniques to the device drivers and kernel code found in operating systems, as well as your own embedded programs. Conversely, if you *don't* understand how this firmware works, you'll spend a lot of time and effort figuring out why your code fails in real life.

The sample code and debugging programs throughout the book use reasonably straightforward C and assembly language constructs. I deliberately avoided esoteric programming techniques in order to concentrate on the interface between the hardware and software. You, too, should pay attention to the small details, before slathering on a high level language topping.

Hosts and Targets

Your existing desktop PC, which I'll call the *host system*, holds all your development tools: editors, compilers, assemblers, and so forth. You will edit and compile the firmware on the host system, then transfer binary or HEX files to the target system. Chapter 1 covers the details of sending a program to the target system using a floppy disk, so you can get started with no specialized hardware at all.

In order to modify the sample code in this book, your host system must have a C compiler, an assembler, and a way to build firmware for a target PC that doesn't have DOS installed. I'll use Micro-C from Dunfield Development Systems, as well as Borland C plus Paradigm *Locate*, to convert the EXE files into binary. Take your pick: Micro-C is inexpensive, while Borland and Paradigm have more features.

The compilers and assemblers run quite happily on nearly any current PC. I used OS/2 to develop the programs and Windows 95 to check them out for this book, as both of those operating systems support ordinary DOS sessions. A multitasking OS simplifies running DOS programs, comm programs, and editors at the same time, but you can accomplish all these tasks with whatever PC you have at hand.

However, an embedded PC *isn't* an ordinary desktop system. You should have a separate PC, which I'll call the *target system*, devoted exclusively to the hardware

and firmware for these projects. Regardless of how careful you are, you run the risk of damaging your target system with a wiring error or a slipped scope probe.

Some of the firmware in this book requires at least a '386-class CPU on the target system. You'll also need a power supply, a diskette drive, an I/O controller board with serial and parallel ports, and a case that holds everything together. Because '386 desktop PCs are now giveaway items, converting one into a dedicated target system isn't expensive at all. Remember, you *won't* need a hard disk, monitor, or even a keyboard to get started. Just ask around and see what's available!

A certain **CAUTION** applies to the firmware:

Run the firmware *only* on your dedicated target system!

The firmware assumes that it controls the target system hardware described in this book. It may produce **unpredictable** and possibly **harmful** results on a desktop PC.

Do not run the firmware programs in this book on your host system!

(Hardware) Construction Ahead

During the course of this book, we investigate how firmware interacts with ISA bus hardware on the target system. However, that process requires some specialized hardware that simply doesn't exist in a standard PC: 16-bit I/O ports, an adjustable wait-state generator, precision timers, text and graphic LCD panels, ID numbers, and so forth. You will see how to build a Firmware Development Board holding that hardware, debug it with firmware test routines, and measure the results.

The hardware projects in each chapter are largely independent of each other, although all of them depend on the ISA bus data buffers and address decoding logic shown in Chapter 3. The simple, two-digit LED display and DIP switches in that chapter provide a convenient way to verify that your bus interface hardware is working correctly. Subsequent chapters introduce various I/O and memory circuits, along with firmware that tests your wiring and exercises the hardware. You can, for example, build just the timers in Chapter 4 and the EEPROM in Chapter 6, if your project demands precision pulses from a PC that boots without a diskette drive.

For those of you in armchair-traveller mode, you can skip most of the hardware construction by building the simple LED-and-switch gadget shown in Chapter 1. It attaches to the parallel port of nearly any IBM PC and gives you practical experience with hardware I/O. You can go a long way with a handful of bits that

reveal your code's realtime behavior, as I'll show throughout this book. Consult Jan Axelson's *Parallel Port Complete* for much more information about parallel ports.

The hardware designs use readily available logic gates and parts, rather than specialized programmable logic and LSI controllers. Because signals on the ISA bus run at relatively low speeds, at least by contemporary standards, you won't need exotic construction techniques or tools: Wire Wrap or solder will work fine.

However, these designs, particularly the Graphic LCD Interface starting in Chapter 12, are not particularly good hardware projects for beginners. Circuit construction depends on your experience, your techniques, and the materials at hand, so I did not include step-by-step instructions. You must be familiar with digital logic, have reasonably good soldering skills, and understand the handling and assembly precautions required to keep integrated circuits working properly.

A further **CAUTION** is in order here:
Regardless of your hardware construction experience or techniques,
you could injure yourself or damage your target system
if you're not careful while building the projects in this book.
Pay attention to what you're doing, **be careful** with hot soldering irons,
and **seek advice** from folks with more knowledge, *before* attempting a
project beyond your experience!

If you already have a few projects under your belt, though, the circuitry shown here will present both a challenge and an opportunity to learn new skills. Take it slowly and carefully... you'll do all right.

The Schematics appendix has the complete set of schematic diagrams and a Bill of Material. You'll also find tables summarizing the I/O and memory addresses, bit definitions, and so forth that appear throughout the source code. Although I don't currently have a printed circuit board or parts kit available, check the Web page mentioned at the end of this chapter for further developments.

Logic Levels and Pin Names

There are, perhaps, as many ways to represent the logic sense of hardware signals as there are schematic capture programs. I follow the hyphen convention, where an active-low or falling-edge clock signal has a leading hyphen: **-SMemW** indicates that the **SMemW** becomes active when it falls near zero volts and is inactive when

near **V_{cc}**. Most TTL control signals, with the notable exception of the ISA bus interrupt lines that we'll meet in Chapter 5, are active-low.

You'll find that the same signal appears quite differently in other references:

/SMemW SMemW/ *SMemW SMemW* SMemW

and other variations too numerous to mention. Pay attention to the meaning of the signal and its logic polarity should follow along easily.

Worse than that, some ISA bus pins bear completely different names in different references. For example, pin B08 can be called **-EndXfr**, **-SRdy**, or **-NoWS**. The hardware doesn't care what name you use: the pin does the same thing in every system. See the ISA bus pin diagram in Chapter 3 for the names I used here.

Some Assembly (Language) Required

Because this firmware operates at the point where software meets hardware, you'll find plenty of assembly language routines. The precise control afforded by assembler code enables you to perform tasks beyond the capacity of higher-level languages, both in speed and simplicity. In many cases, a few assembler instructions can illustrate a key point by showing precisely how the CPU and ISA bus operate.

I won't dwell on the intricacies of splicing assembly language routines into C programs. The sample code on the diskette accompanying this book shows how I accomplished the task, but you may find that your compiler and assembler use an entirely different technique. I trust that, once you see what must be done, you can figure out a way to accomplish it with the tools you have available.

If you've never used assembly language before, you're in for a treat. It's a vital skill for folks who must wring the last bit of performance from a system... after you've done everything else, assembler turns on the afterburner.

What Else Do You Need?

Yes, hardware and firmware debugging can be difficult without the right tools.

Many of the programs send debugging information from the target system's serial port to a comm program running on your host system. Some embedded programs and routines, however, must run at times and in places where the serial ports aren't available. In those cases, you'll see how to blink LEDs, send trace information to the parallel port, and use other debugging techniques that don't depend on exotic and expensive test equipment.

Sometimes, however, you simply *must* have the right hammer for the job.

While you won't absolutely need an oscilloscope for the projects in this book, a good 'scope certainly helps show what's going on as you tweak the firmware. You may have seen "high resolution" software timers described in books and magazine articles, but the events we're dealing with happen far faster than a CPU can measure with software. I'll show you how a scope can benefit your firmware development efforts by measuring realtime performance.

Similarly, a logic analyzer presents timing information for many digital channels at once and can reveal behavior that's otherwise invisible. You'll see some logic-analyzer screen photos that show timing information gathered from the target system's ISA bus signals and firmware trace outputs on the parallel port. When you build similar trace and triggering outputs into your code, you can make a logic analyzer even more valuable by capturing precisely the information you need.

Given a choice between a scope and a logic analyzer, I'd pick a scope every time. Current digital scopes make that decision less painful than it used to be, by incorporating many features from logic analyzer. If you plan on doing low-level firmware, you can greatly simplify your life with good test equipment.

Even a simple logic probe can provide evidence that your firmware reached (or *didn't* reach) a particular instruction. Just add a few lines of code that pulse a parallel port pin, hitch your probe to the pin, and see what happens. That may be all you need to trace down a gnarly problem that resists conventional debugging.

Beyond the hardware and firmware lie the most important debugging tools you can have: your active curiosity and desire to figure out how things work. I'll describe what you should look for and suggest how best to see it, which will improve your ability to uncover out-of-the-ordinary problems. If you've got the time and inclination, investigating how this firmware works by building your own target hardware will improve your understanding, give you considerable practical experience, and help you avoid designing problems into your systems.

Remember... the best debugging is no debugging at all!

Numeric Representation

Because this book deals with firmware written in both assembly language and C, you'll find numeric constants using hexadecimal (radix 16) and decimal (radix 10). The same numeric value will look different, depending on where and how it's used. I've attempted to write the numbers in the firmware's source-code format, rather than force-fit them into typographic consistency.

The Micro-C assembler represents hexadecimal numbers with a leading dollar sign: `INT $19` and `MOV AL, $FF`. The Borland assembler uses a trailing `h` and insists that the first digit be numeric: `INT 19h` and `MOV AL, 0FFh`. Firmware written in C, regardless of the dialect, uses a leading `0x` for hex numbers: `intr(0x19)`.

Because memory addresses always use hex notation, I generally won't bother with a radix identifier unless decimal numbers lurk nearby. Although those of you with experience in C on non-PC platforms may be familiar with C's leading-zero octal notation, it doesn't appear anywhere in the book or firmware listings. For example, the value 0123 represents 0123 hex, not 123 octal, or, for that matter, 123 decimal.

Reference Material

The Bibliography appendix contains pointers to all the reference material mentioned in the text. Because a book's ISBN changes with each edition, you must use the title or author information to locate the most recent version. Any good bookstore should be able to track down and order a particular book for you, although you may find that they have trouble with smaller publishers and works that are out of print.

In the last few years, the World Wide Web has become an invaluable source of technical information. The Sources appendix lists URLs and addresses for some of the companies I've dealt with through the years, but a few minutes with any of the Web search engines will turn up even more companies. Unless you live in or near a hotbed of technical activity, you'll find that most of your crucial parts and supplies will arrive in boxes from distant companies that you never see face-to-face.

Remember the source code on the diskette tucked into this book! The listings throughout the book present what I consider the key sections of the programs, but the files hold many other tricks and techniques that I haven't mentioned.

Much of the text in this book came from a series of *Firmware Furnace* columns I wrote in *Circuit Cellar INK* magazine. Thanks go to *INK*'s staff and readers, who gave me plenty of feedback and suggestions that I've included in this book. The remaining errors are, of course, entirely of my own invention.

Errata and Updates

Despite my best efforts, I'm *certain* some of this book's code and hardware won't work on your target system. Newer and faster target CPUs, different system chipsets, and your hardware construction techniques all affect firmware that runs down at the bare silicon level. Who knows? You may even find a bug or two in the listings or schematics...

Your first line of defense lies in the debugging techniques you'll learn here. Try using the debuggers, inserting trace statements into the code, lighting LEDs, or triggering your oscilloscope or logic probe. You can tweak the code to dump registers, pinpoint problems, and work around peculiarities. You'll run into those problems in your own projects, so consider the hardware and firmware you find here as valuable warmup exercises for the real world.

When all else fails, check this book's FAQ on the publisher's Web site at www.peer-to-peer.com. I'll add any corrections, clarifications, and tips that arrive after the book went to press.

You can also email me at isabus.book@pobox.com. If you're stuck, I may be able get you around a problem. Your comments and suggestions will certainly make the next edition even better.

Typography

Plain, flat ASCII text simply can't cope with the requirements of firmware and hardware documentation. In this book, different typefaces distinguish different types of words. You will see, at a glance, whether a word represents a **file name**, a **firmware variable**, or a **logic signal**:

Function	Face
Body text: <i>italic</i> and bold for emphasis	Adobe Caslon
Captions	Helvetica
Source code listings	Lucida Console
Hardware logic signals	Courier Bold
Disk file names	American Typewriter Condensed
DOS and system commands	Trade Gothic Condensed Bold
Special symbols: × μ Ω © ↑	Symbol Set

Enjoy!

Ed Nisley KE4ZNU

1 Blindsided by Technology

Suppose your project needs a high-performance CPU, lots of DRAM, comprehensive interrupt support, a real-time clock, nonvolatile configuration RAM, serial and parallel ports, and a megabyte-per-second I/O bus.

Quickly: how long will the hardware design take and what does the first unit cost?

Give up? How about five minutes and a few hundred bucks?

Once again, the rules are changing...

Commodity Computers

Now that a fully loaded, high-performance PC system board retails for under \$200, it's easier than ever to justify devoting adequate computing power to tasks once squeezed into microcontrollers. With reasonably standardized PC hardware available from many vendors, you can devote your attention to the proprietary part of your project, the knowledge that makes your product unique.

Your product may, of course, require specialized hardware that nobody else knows how to build. By constructing that hardware on a board that plugs into a standard PC, you need not build an entire computer before your project gets off the ground.

Current PCs include both PCI and ISA standard peripheral buses. The Peripheral Component Interconnect bus is optimized for very high speed data transfers between the CPU and several intelligent peripherals. Unfortunately, its high data rate precludes "simple" hardware design and PCI device engineering is not for the faint of heart.

The Industry Standard Architecture bus, in contrast, is essentially the same bus as IBM designed into the PC AT back in the mid 1980s. At the time, it was entirely fast enough for the job. By contemporary standards, it is both slow and limited.

The ISA bus has two compelling advantages: a comparatively simple hardware interface and across-the-board availability. *Every* desktop PC in the world has a few ISA bus connectors on its backplane. If your gadget can operate within the limits of the ISA bus architecture, it's the only way to go.

The Bibliography appendix includes several books that define how the ISA bus operates and what your device must do in order to work correctly. What's missing

from those books is how to write the firmware that makes the hardware jump through hoops. We'll do that here, after we get the PC hardware nailed down.

Essential Hardware

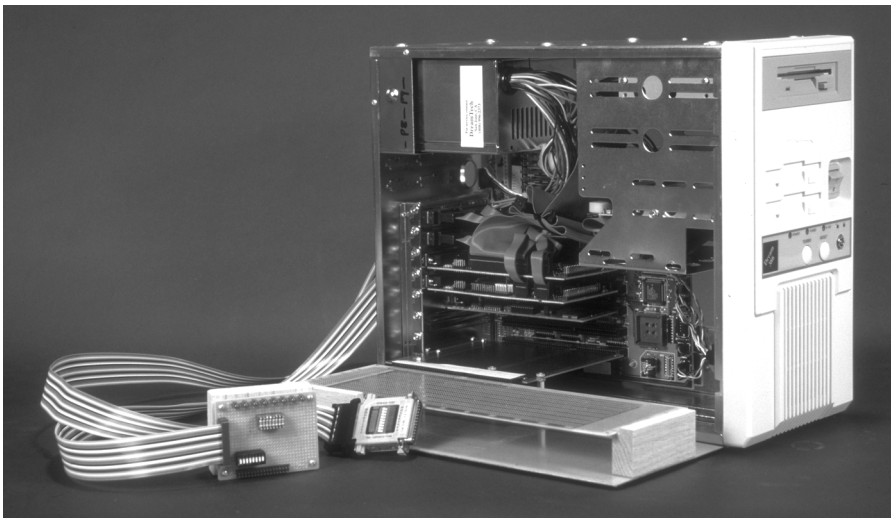
The code in this book will run on any '386SX or better CPU, but it is an Exceedingly Bad Idea to use your *real* PC as a guinea pig. A single slip of the scope probe or a minor programming error can literally put you out of business by wiping out either your hardware or your on-disk data. The cost of a minimal ISA bus PC system is tiny, compared to the aggravation of recovering your data. Trust me.

Besides, we will monitor the target system's operation through its standard PC serial port. You already have a comm program set up on your existing PC, so why make things harder than necessary? If you plan to build embedded systems, start working with separate hardware right away and debug the interconnections first.

You'll find that a mini tower case works better for these projects than a standard desktop case, because a tower extends the ISA bus boards horizontally over your desk. Photo 1 shows the target system arrangement I used, with a 16-bit ISA bus extender connecting the lowest bus slot to an ISA bus prototype board resting on a

Photo 1

A commodity PC provides a firm foundation for experimentation with the ISA bus. We will monitor our firmware through this target system's parallel and serial ports; note that it lacks a keyboard and monitor! Make sure your ISA bus prototype board has good mechanical supports, lest flaky connections drive you mad.



homebrew support. Although it's not shown in the picture, I machined a slot in the PC case around the bus extender, making a neat package that maintains normal airflow over the system board with the cover in place.

Throughout the book, you'll see photos and diagrams of ISA bus signals. After you work with the bus for a while, you won't need labels to find the signals, but, until then, some help is in order. See Chapter 3 for a label that identifies each signal: with each ISA bus pin name in plain sight, bus-level probing becomes a snap.

Because all PC system boards are set up for DOS or Windows, you must change a few BIOS CMOS configuration parameters in your target system. Attach the display and keyboard from your main PC, run the BIOS setup routine, tell it to run without a display or keyboard, enable booting from the diskette, nudge the Real-Time Clock into your time zone, and disable any virus checking. All this may be a nuisance, but it's much faster and far easier than having to write the whole BIOS yourself. Honest!

Booting Firmware

When you think of firmware, you probably also think of code stored in EPROMs or flash ROMs, developed with CPU emulators and other complex (and expensive) hardware. We can take an easier path: write the code on your host PC, inject it into the target PC through a floppy drive, then monitor and debug the results through a serial port. Think of it as squishy firmware.

Of course, emulators and simulators have their place. Some bugs simply can't be exterminated without the level of detail provided by in-circuit tools. However, we can cover quite a lot of territory without too much exotic equipment or esoteric software. To a large extent, you already have the tools on your desktop PC.

This may seem like cheating, but an all-in-one I/O board from nearly any PC supplier costs \$20 (less, in some cases) and a 3.5" floppy drive gives you 1.44 MB of storage for perhaps \$30. Price out the equivalent space on an EPROM or flash ROM board and report back; perhaps your project can use a floppy after all?

Remember, the rules are changing...

No, a floppy drive isn't appropriate for rugged, all weather, no-moving-parts applications. But many (most?) embedded systems run in shirtsleeve environments where a floppy that spins only during system resets makes a lot of sense. Even better, data collection applications can have PC-compatible data storage with essentially no overhead.

And, contrary to what you might think, loading a program doesn't require DOS, either. Loading a **COM** file involves nothing more than copying bytes from disk into memory, setting up a few registers, and jumping to the first instruction. **EXE** files are a tad more complex, but we'll tackle that problem in Chapter 11.

The BIOS on our ISA bus system board gets control after a hardware reset, tests and initializes the hardware, reads 512 bytes from Drive A, Track 0, Head 0, Sector 1 into RAM starting at address 0000:7C00, then jumps to the first byte at that address. If that sounds a lot like loading and starting a program, you're right: it's the standard bootstrap loader that starts your favorite operating system every time you turn on your PC, applied to a diskette rather than a hard disk.

The BIOS neither knows nor cares what the program in that first sector does. In our case, we'd like to load an embedded program rather than an operating system like DOS, Windows, or OS/2. Rather than the standard loaders that come with those operating systems, we need a custom bootstrap loader that can find our program on the diskette and read it into RAM.

Here's the trick that makes this all work out. If our bootstrap loader knows how to interpret the DOS FAT file system, we can produce the embedded PC program on a desktop PC, use the DOS **COPY** command to put it on the floppy, stick the floppy in the target system, and hit the Reset button. After our program starts running, it can do whatever we choose.

What could be easier?

The Boot's Strap

The first sector on each floppy disk contains both the bootstrap loader program and a table called the Diskette Boot Record. Each type of diskette has a unique DBR and thus requires a custom bootstrap program to read its data. Fortunately, a little assembler magic can paper over the differences.

Listing 1 shows the code that sets up a DBR. The **DISKSIZE** assembler macro selects one set of constants that are then plugged into the DBR variables. The values shown specify a standard, 720 KB, 3.5" diskette. Although you could create a bizarre diskette format by twiddling the constants, I recommend exercising some caution. My experience shows that choosing anything other than the standard diskette formats will get you in big trouble with at least one version of DOS, Windows, or the PC BIOS.

Listing 2 shows how the bootstrap loader calculates the directory's location from the DBR values and reads the first sector into RAM. Each directory entry holds

Listing 1

The Diskette Boot Record holds the information required to read the rest of the diskette. Assembler macros define the constants starting with “N_” to create a boot record for each type of diskette. The bootstrap loader uses these values to read the DOS file directory and load the embedded system program.

```

                                IF      DISKSIZE EQ 720
                                DISPLAY "Creating 720K diskette boot sector"
N_TRACKS      EQU      80
N_TRACKSIZE   EQU      9
N_HEADS       EQU      2
N_CLUSTERSIZE EQU      2
N_ROOTFILES   EQU      112
N_MEDIAID     EQU      0F9h
N_FATSIZE     EQU      3

                                ELSEIF
<<< other diskette sizes and data omitted here... see the source files >>>
                                ENDIF

N_SECTORS      =      N_TRACKS*N_HEADS*N_TRACKSIZE

SectorSize     DB      'Firmware'      ; 03 OEM name and ver (8 chars)
ClusterSize    DW      512              ; 0b bytes per sector
ClusterSize    DB      N_CLUSTERSIZE   ; 0d sectors per cluster
NumReserved    DW      1               ; 0e reserved sectors
NumFATS        DB      2               ; 10 number of FAT copies
MaxRootFiles   DW      N_ROOTFILES     ; 11 maximum root dir entries
NumSectors     DW      N_SECTORS        ; 13 total sectors on diskette
MediaID        DB      N_MEDIAID       ; 15 useless media descriptor
FATSize        DW      N_FATSIZE        ; 16 sectors in each FAT
TrackSize      DW      N_TRACKSIZE      ; 18 sectors per track (per head)
NumHeads       DW      N_HEADS          ; 1a number of heads
HiddenSectors  DD      0               ; 1c number of hidden sectors
              DD      0               ; 20 number of sectors if >32 MB
              DB      0               ; 24 internal DOS drive ID
              DB      0               ; 25 reserved
              DB      29h             ; 26 Boot signature
              DD      000FF00EDh      ; 27 volume ID number
              DB      'Firmware386'   ; 2b volume label (11 chars)
              DB      'FAT12'         ; 33 file-system type (8 chars)

```

one file’s starting cluster and size, so the loader uses the first entry to find the first file, reads it into RAM at address 1000:0100, and passes control to it.

There are, of course, a few restrictions: the program file must be the first one in the diskette root directory (although the loader will ignore a volume label and Windows 95 Long File Name entries), it must be contiguous (because the loader completely ignores the File Allocation Table), and it must be less than 64 KB long (as all COM files must be). In exchange, the loader sets up the registers so any COM program that doesn’t use DOS will work just fine. Fair enough?

Oddly enough, the standard DOS bootstrap loader has similar restrictions. The two DOS startup files (IO.SYS and MSDOS.SYS) must be the first two files in the

Listing 2

This excerpt from our custom diskette boot loader reads the DOS directory, skips over directory entries that don't correspond to files, locates the first file, and reads it into RAM. That file may be any *.COM program written with the constraints of our embedded target system in mind. Loading the program at 1000:0100 avoids problems with diskette DMA across 64 KB physical memory boundaries.

```
;--- locate and fetch first directory sector
;   this goes to 0000:7E00

        XOR     AX,AX                ; start with zeros
        MOV     AL,[NumFats]          ; account for sectors in both
        MUL     [FATSize]            ; ... copies of the FATs
        ADD     AX,[WORD LOW HiddenSectors] ; these are surely
        ADC     DX,[WORD HIGH HiddenSectors] ; ... unused
        ADD     AX,[NumReserved]      ; boot sector and other
        ADC     DX,0                  ; ... hidden sectors
        MOV     [AbsSector],AX        ; save start of dir

        CALL    CvtSectors            ; set up parameters
        MOV     BX,7E00h              ; set up target
        CALL    ReadSector            ; and get one sector

;--- skip volume labels and other debris
;   this also peels off win 95 long names...

SkipNonFiles:
        TEST    [ES:BX+DirEntry.FileAttrs],(MASK_FA_Label) OR \
                (MASK_FA_System) OR (MASK_FA_Hidden)
        JZ      HaveFile
        ADD     BX,SIZE DirEntry
        JMP     SkipNonFiles

HaveFile:

;--- find sector address of data area just after directory
;   we use directory size in bytes rounded to next whole sector

        MOV     AX,SIZE DirEntry      ; entry size in bytes
        MUL     [MaxRootFiles]        ; times entries
        MOV     CX,[SectorSize]       ; round upward
        ADD     AX,CX                  ; by sector size - 1
        DEC     AX
        DIV     CX                     ; get sectors
        ADD     [AbsSector],AX        ; add to dir start

;--- find file's starting sector from cluster number

        MOV     AX,[ES:BX+DirEntry.FileStart]; first cluster
        SUB     AX,2                    ; ... starts at 2
        MOV     CL,[Clustersize]
        XOR     CH,CH
        MUL     CX                     ; AX = sectors
        ADD     [AbsSector],AX        ; save for later
```

Listing continues on next page

Listing continued from previous page

```
;--- convert file length from bytes to sectors
;   we know the length must be < 64K bytes, so ignore high word

        MOV     AX,[WORD LOW ES:BX+DirEntry.FileLength]
        XOR     DX,DX                               ; this is always 0
        DIV     [SectorSize]
        CMP     DX,0                                ; any remainder?
        JZ      HaveLen
        INC     AX
HaveLen:
        MOV     [SectorCount],AX                    ; set up for reading

;--- now pull in the file to 1000:0100
;   1000:0100 = physical address 10100, or just over 64K
;   This allows up to FFF0 bytes of data before a DMA boundary crossing

        MOV     BX,1000h                             ; set up target address
        MOV     ES,BX
        MOV     BX,0100h

NextSector:
        MOV     AX,[AbsSector]                       ; set up starting sector
        CALL    CvtSectors
        CALL    ReadSector                           ; and get one sector

        INC     [AbsSector]                           ; step to next sector
        ADD     BX,[SectorSize]                       ; ... and next target

        MOV     CX,[SectorCount]                     ; continue until done
        DEC     [SectorCount]
        LOOP    NextSector
```

directory and they must be contiguous. You can only jam so much intelligence into 512 bytes of loader firmware, making some restrictions perfectly understandable.

Just to keep you on your toes, `MSDOS.SYS` has changed rather dramatically under Windows 95: it's now a text file rather than an executable program. Take a look and see... it's a hidden file in the root directory of your Windows 95 boot disk.

The files in this chapter's subdirectory include four bootstrap loaders: `Boot720.SEC` and `Boot1440.SEC` handle the 3.5" formats, while `Boot360.SEC` and `Boot1200.SEC` cover the 5.25" field. You can use these directly or modify `BootSect.ASM` to produce a customized version for your own system.

Listing 3 shows how to create a boot diskette for your target system using plain old `DOS Debug` on your host PC. The only tricky part arises because `Debug`'s **L** and **W** commands have changed slightly over the years; some DOS versions use a sector range, while others use a sector start and count. Check your documentation to see which one you've got. You can even run `Debug` in a DOS box under Windows 95, with no trouble at all.

Listing 3

DOS DEBUG, running on the host PC, provides the easiest way to create a bootable diskette for our embedded programming projects. Then, the bootstrap loader on the target system simply reads the first valid file from the diskette, pops it into RAM, and runs it. Note that DEBUG's L and W commands work differently on some system: the last digit may be either the final sector number or the number of sectors to transfer.

```
DOS•H:\ISABus\Code\Chap_01 >format a: /f:720          create a DOS data diskette
<<< normal formatting stuff happens here >>>
```

```
DOS•H:\ISABus\Code\Chap_01 >debug                  fire up DEBUG
-l 100 0 0 1          read first one or two sectors from Drive 0 (A)
-nboot720.sec         specify our boot sector
-l                   load it in
-w 100 0 0 1          write one or two sectors to Drive 0 (A)
-q                   back to DOS
```

You can use this shortcut with the version of DEBUG found on Windows 95 and recent MS-DOS systems:

```
DOS•H:\ISABus\Code\Chap_01 >format a: /f:720          create a DOS data diskette
<<< normal formatting stuff happens here >>>
```

```
DOS•H:\ISABus\Code\Chap_01 >debug boot720.sec         get loader into DEBUG
-w 100 0 0 1          write one sector to Drive 0 (A)
-q                   back to DOS
```

The First Program

The canonical First Program on any system displays a trivial message like C's "Hello, world" routine, but our target PC doesn't have a display yet. Think small: how about changing a single I/O bit? An LED connected to that pin will make the result visible. Unfortunately, our target system lacks even an LED...

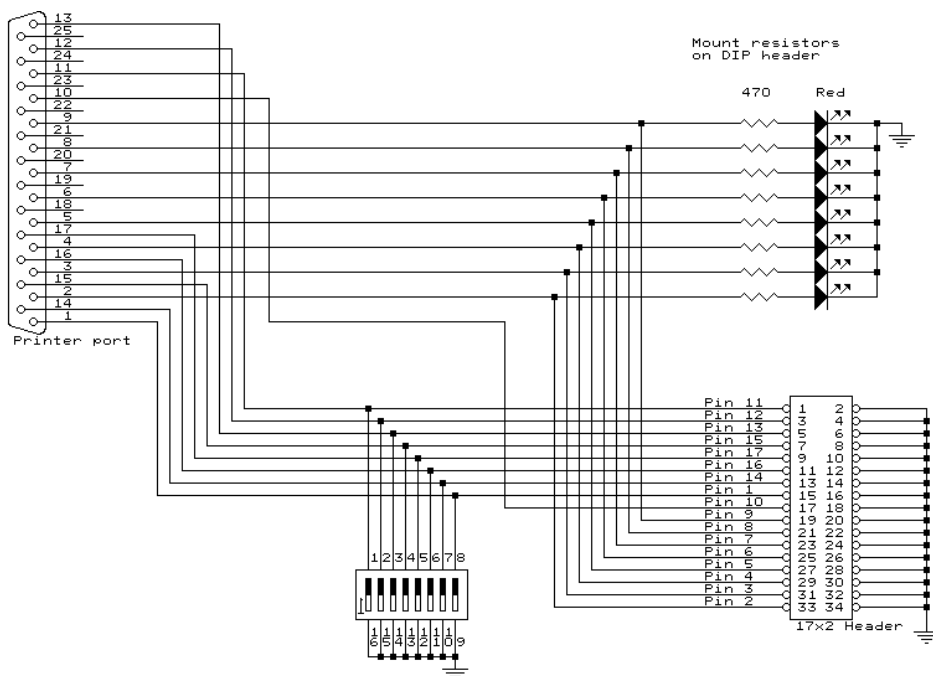
Schematic 1 shows how to get started: attach eight LEDs and eight switches to the target system's printer port. A PC rates at about one megaton of overkill for an LED blinker, but at least we can see if the program runs. Later on, we'll use those LEDs as trace outputs from more complex code; this is an easy way to get started.

Listing 4 shows how `CountSlo.ASM` displays a counting sequence on LPT1 slowly enough to be visible to a naked eye watching the LEDs. The key line, of course, is the `OUT DX,AL` instruction that writes to the output port. The `INC AL` creates the counting sequence and the `JMP` at the bottom runs the loop forever.

Although just those instructions would suffice, you wouldn't see more than a blur on the LEDs. I used the BIOS function `INT 15h, AH=86h` to generate a delay between counts. We'll find and use several handy routines in the BIOS throughout this book; it's a shame to let them go to waste!

Schematic 1

This circuitry converts the target system's printer port into one input and one output byte. The inputs are pulled up by resistors on the I/O board, while the outputs have (barely) enough drive for the LEDs. Note that the LEDs are ON when the outputs are HIGH. The header provides convenient access for scope probes, which will come in handy when we tackle performance issues.



I used Borland's **TASM** and **TLINK** to assemble **CountSlo.ASM** into **CountSlo.COM**, using the **MakeFile** you'll find on the source code diskette. Because I don't want to run the program, even by accident, even once, on my host PC, the **MakeFile** renames it to **CountSlo.BIN**. The bootstrap loader neither knows nor cares about the file's name or extension, so you may call it anything you like.

The **MakeFile** also copies the new file to the floppy in Drive A. Because the boot loader picks the first file, my **MakeFile** executes **DEL A:*.BIN** to wipe out any other embedded system programs, then copies the new **BIN** file to the floppy. While this reduces the amount of typing required to rebuild a program, be sure you've got the right floppy in the host system's drive before you start.

Listing 4

This excerpt from CountSlo, our first embedded program, displays a counting sequence on the parallel port LEDs so you can see something happening. The BIOS identifies the installed hardware during its power-on tests and puts the parallel port's address in the usual low-RAM location. This chunk of code recovers that address and blinks the LEDs. Refer to the complete source file on the diskette for the rest of the program.

```

        MOV     DX,0040h           ; aim at BIOS data segment
        MOV     ES,DX

        MOV     BX,2*(LPTPORT-1)+0008h ; aim at port entry
        MOV     DX,[ES:BX]         ; get LPT port base address

        XOR     AL,AL              ; start from zero...

eShow:
        INC     AL                 ; tick it
        OUT     DX,AL              ; send it out

        PUSH    DX                 ; save address
        PUSH    AX                 ; and count

        MOV     CX,COUNTRATE       ; delay for a while
        MOV     DX,0
        MOV     AH,86h
        INT     15h

        POP     AX                 ; recover count
        POP     DX                 ; and address

        JMP     ReShow             ; continue forever
```

Here's the process in a nutshell: set up your ISA bus target system and build the little LED and switch board shown in Schematic 1. On your host system, install the appropriate diskette boot loader using **Debug** as shown in Listing 3 and copy CountSlo.BIN to the floppy using the DOS **COPY** command. Pop the floppy into the target system's diskette drive, hit the target's Reset button, and watch the lights.

That's all there is to it... welcome to the wonderful world of embedded PC programming, without all the aggravation and hocus pocus you expected. Watch that first step, because it's all downhill from here!

Release Notes

If you haven't already done so, get the source code diskette accompanying this book out of its pocket, then check the **ReadMe.txt** file in the diskette's root directory for directions on how to install and decompress the source and binary files to your host system's hard disk. Separate subdirectories will hold the files for each chapter, along with other information you need to rebuild the programs.

The code snippets you see in the listings throughout the book represent only the tip of the firmware iceberg. You should refer to each chapter's subdirectory for the complete source code and executable files. Although the key points appear in the listings, you'll find many comments in the code that explain other topics and show how to accomplish some useful tricks.

See the `ReadMe.txt` in this chapter's subdirectory for more information on reassembling the files, which include several other programs that check out the switches and LEDs you just built. `BlinkSer`, `CopyStat`, `CopyCntl`, and `CountPar` are each simple enough that they should work on any PC, but, not surprisingly, some PC hardware isn't entirely compatible with the IBM spec.

For example, my ancient '386SX laptop PC lacks pullup resistors on the four control output bits of its parallel port, which means those input bits simply float low and the DIP switches have no effect. If the sample programs don't seem to work on your target system, do a little probing *before* you claim a software problem. You'll meet enough examples of real hardware problems later on in this book.

Identifying such problems is the hardest part of writing firmware, so starting with known-good programs on your new target system makes lots of sense. After you get some practice with embedded programming, you'll be able to start from scratch on your own... just keep reading and doing the projects and you'll be well on your way.

Consult the Sources appendix for vendors who can provide the bits and pieces of hardware and software that we'll use throughout this book.

Once again... **CAUTION!**

Do not run these programs on your host PC!

Get a cheap ISA bus PC system, modify its case, and get some experience building and debugging hardware gadgets on your new target system.

2 Chickens and Eggs

Now that we have a way to boot programs into our ISA bus target system, what should the First Program be? A multitasking, multiuser, multimedia, protected mode operating system featuring 32-bit flat memory model programming with no artificial segment limits?

Hardly!

The whole point of this book is providing the detailed knowledge you need in order to write good firmware and build custom hardware for an ISA bus embedded PC. While I suppose you can simply ignore all of the low-level details, I've found that this kind of knowledge is indispensable in ferreting out bugs, taking maximum advantage of the machinery, and generally doing a good job.

The First Program should be a debugger. The Second Program should be a simple C program. Next, we will look at the hardware used in typical ISA bus interfaces. After that... well, all things in due time.

Despite the killer GUI debuggers now in vogue for PC program development, a traditional command line debugger has a lot going for it, at least for simple projects. Those "primitive" debuggers are relatively small, you can drive them from a serial port, and they do what you need to get things started from scratch. If you have ever used the DOS `Debug` program (and who hasn't?) you know what I mean.

But, because our target system doesn't have DOS, we can't use good old **Debug**. Rather than write my own debugger, I'll take advantage of the Dunfield Development Systems 8086 Micro-C Developer's Kit. It includes the Micro-C compiler, an assembler, a pair of debuggers, and a host of other utilities useful for embedded x86 program development.

There are, of course, higher-powered ANSI standard C/C++ compilers that can generate code for embedded PCs and, as we'll see, ways to run GUI debuggers through serial ports and network links. Dunfield's programs are straightforward enough that we can see the fundamentals quite clearly. We'll get more complex in successive chapters, but, for now, let's get the basics up and running.

The Bare Essentials

The diskette boot loader I described last in Chapter 1 provides a simple and inexpensive way to load a program into the target system. In the next chapter, we will begin building a Firmware Development Board that will eventually hold some

nonvolatile memory that can make the diskette optional, but the code from this chapter will help us check out that board and the code that runs on it.

Recall that our target system includes an integrated I/O board with a pair of serial ports or a system board with all that built in. Because we are not using the keyboard or display, those two ports provide an ideal way to communicate with the firmware. It seems a shame to let a resource like that go to waste, so I'll use COM1 for the firmware and COM2 for the debugger that helps get the code running.

Dunfield's **HDM86** (which stands for Hardware Debug Monitor) is one of the slickest chunks of code I've seen in a while. It will run on any 80x86 CPU and uses no external RAM: *all* of its data stays in the CPU's registers, including the return address for one level of subroutine call. Talk about tight coding!

Although I won't require **HDM86** for our projects, it comes in handy to verify hardware startup functions such as DRAM refresh and memory mapping. Of course, **HDM86** must be in EPROM (or loaded from diskette, as we will do) and you can't download any code (where would it go?), but performing the actual I/O operations "by hand" can often reveal circuit problems.

HDM86 includes commands to run tight read/write loops that hammer on a specific I/O or memory address. This makes it easy for you to scope out problems, because the hardware strobes provide convenient sync points. Of course, you must reset the system to regain control, but that's why **HDM86** can live in EPROM.

Incidentally, CPU and system board hardware setup is one of the (many) problems we've sidestepped entirely by using a standard ISA bus PC system board as our target system. As far as we are concerned, the BIOS initializes everything and passes control to us through the diskette boot loader. While we can change anything necessary (and we certainly will, as you'll see later on), the essential startup details are a done deal.

Remember, the rules are changing. You give up detailed hardware diagrams to gain PC compatibility and a rock bottom per-unit price. All things in PC-dom are negotiable, so you can get schematics if you're using system boards in production quantities and you can write your own BIOS if you must have absolute control. However, the relentless pace of PC progress means any system you use today will be obsolete in six months, so don't plan on using any tricks that work on just one system. Stay generic and stay alive...

MON86, the larger of Dunfield's two 8086 debuggers, occupies a little over 5 KB (*gasp!*) and includes a disassembler, **HEX** file loader, up to eight code breakpoints, and the usual **Debug** style commands. It arrives set up for the PC's keyboard and

video display, but is easy enough to adapt to any standard PC serial port. Dunfield includes a detailed set of comments to help you through the process, which involves changing a few constants and reassembling the code.

The diskette boot loader handles standard COM programs starting at offset 0100, but MON86 assumes its programs begin at 0000. The only change required is to set MON86's initial User PC (Program Counter, which Intel fans know as the Instruction Pointer: IP) to 0100 instead of 0000.

MON86 disassembles code into 8086-level instructions, but this is not a serious impediment during the first few projects because plain old 8086 instructions provide all the functions we need. When instructions found on '386 and higher CPUs will benefit our code, we'll use an appropriate assembler and debugger.

Load and Go

You could burn the modified MON86 into an EPROM, but I put it on a diskette and boot it into the target system's RAM using the loader from Chapter 1. Remember that the whole point of this exercise is building the tools to build the board to hold that EPROM.

Your host system must have a serial port and communications program to talk with MON86 on the target PC. Schematic 1 shows the three possibilities for the null modem adapter between the host and target system serial ports. With standards like this, it's a wonder we get anything working at all. Be careful of commercial null modems, because I've seen some that cross-wire all of the CTS/RTS and DTR/DSR pins together. Hard to believe, but true.

When MON86 starts up on the target system, you should see a display similar to Listing 1 on your host system's comm program. If not, recheck the source in MON86.MAC to verify that you have all the options set correctly.

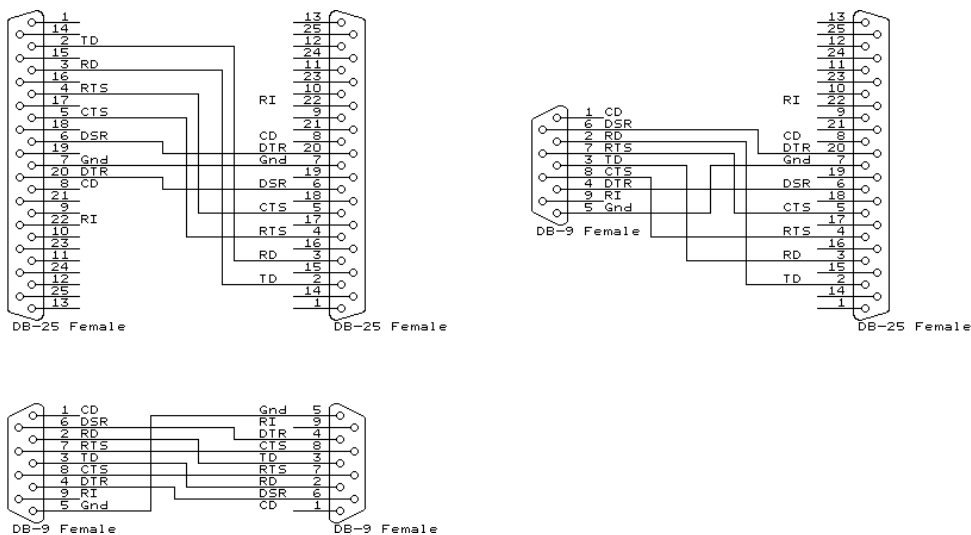
Listing 1

This is what MON86 sends from the target system to the comm program running on your host system. If you get something else (or nothing at all), make sure that both MON86 and your host system comm program are set up correctly... and doublecheck the serial cable! MON86 uses a * prompt rather than the more familiar - or > characters.

```
MON86 Version 1.1
Copyright 1991,1995 Dave Dunfield
All rights reserved.
*
```

Schematic 1

Connecting two PC serial ports together requires a null modem adapter to swap the input and output pins. Use a breakout box or continuity checker to verify the connections inside commercial null-modem adapters, as I've found some that were wired incorrectly. You need not connect the other pins. For example, because the communication program on the host PC will not be dialing or answering a modem, the RI and CD pins are unused.



Although I should not need to mention this, make *sure* you plug the serial cable into the right ports on both PCs. I speak as an expert, having gotten *both* ends wrong on the first crawl under the desk.

Once **MON86** boots up, use its **Output** command to write a few values to the parallel printer port at address 0378. The LED and switch circuit shown in Chapter 1 will come in handy to watch the bits toggle. This simple step verifies that **MON86** is loaded, the serial ports and your comm program work, and the parallel port is at the address you expect.

The next step involves loading a **HEX** program with, naturally enough, the **Load** command. **Rotate.ASM**, shown in Listing 2, has a simple loop that rotates a single bit across the parallel port. It uses a BIOS delay function to make the output visible, so you won't absolutely need a scope to check this code out.

Listing 2

Use MON86 to load Rotate.HEX through the serial port. The LED and switch hardware described in Chapter 1 will show you the results appearing on the parallel printer port. The code shown here should be assembled with Dunfield's ASM86 assembler; the syntax is slightly different than Microsoft's MASM or Borland's TASM, but no more unusual than some other assemblers.

	ORG	\$100	
*			
	MOV	DX,#0378	Select parallel port
	MOV	AL,#80	Set up the rotating bit
*			
ReShow	ROL	AL,1	Rotate it
	OUT	DX,AL	And display it
*			
	PUSH	DX	Save our values
	PUSH	AX	
*			
	MOV	CX,#1	Use BIOS delay function
	MOV	DX,#0	... to get 64K microseconds
	MOV	AH,#86	
	INT	\$15	
*			
	POP	AX	Recover our values
	POP	DX	
*			
	JMP	<ReShow	

Dunfield's assemblers can produce either Intel or Motorola HEX files and MON86 will accept either flavor without any special attention. However, if you plan to send the files to an EPROM programmer or use them as an input to other programs, be sure you select the appropriate assembler command line switches.

MON86 does not echo the characters it receives while transferring the HEX data, which means you may have to configure your comm program to *not* wait for the end of each line. If the comm program seems to hang after sending the first line, the two programs are probably locked in this deadly embrace. Cancel the transfer, reset the comm program's options, and try again.

Dunfield's HEXFMT utility will transform Rotate.HEX into Rotate.BIN, which the diskette boot loader will pull directly into RAM. This is the main advantage of changing MON86's default User PC: you can debug programs using MON86, then boot them directly from diskette with our loader, without making any changes.

The C compiler forms the last piece of the puzzle, as you might expect. It turns out that the effort is almost anticlimactic because Dunfield has done nearly all of the required work already.

Micro-C Startup

This may come as a shock to some C firmware programmers, but there's some startup code that runs immediately after the CPU comes out of a hardware reset, long before your `main()` function gets control. Regardless of whether you program in C on a microcontroller or an x86 CPU, you must understand the how, what, and why of that startup code. Only after the startup code runs successfully can your program execute your instructions.

On PCs running a standard operating system, of course, all this is taken care of for you. Now that our target system runs without even DOS, it's time to learn more about the machinery you've been taking for granted all along.

The "bare metal" startup begins when an x86 CPU emerges from hardware reset by fetching the first instruction. Each of the 8086, 80286, 80386, 80486, and Pentium CPUs start with slightly different `CS:IP` register values, but the system board maps the resulting physical address to `FFFF0` (a.k.a. `F000:FFF0`) in the ROM BIOS.

As I mentioned earlier, the PC's BIOS handles all the initializations required to get from that first instruction fetch to a system that can boot an operating system or, better yet, your embedded program. If you are writing x86 embedded code without the luxury of a BIOS, you must know the details of the target system's hardware so you can provide all the right startup functions.

Several vendors offer "BIOS kits" that reduce the tedium of rolling your own BIOS. Even if you don't use all the standard functions, starting with known-good code can trim months off your schedule. Check the Sources appendix and the ads in your favorite magazines; this is a small but significant niche market.

For our present purpose, however, the BIOS sets up the hardware, reads our boot loader from the diskette, sets up the segment registers, and branches to the loader. Our code reads in the first DOS file, sets up the segment registers, and branches to whatever is at offset 0100. That's where the C startup code must begin.

Listing 3 shows the top part of `8086RLPT.ASM`, Micro-C's Tiny memory model startup code. It sets up the segment registers (again), clears the uninitialized variables, and starts the C program with a `CALL` to the `main()` function. If the `main()` function returns, the default Micro-C startup code invokes the `INT $21` Terminate Program DOS function, which is appropriate for DOS-based programs.

I replaced the termination code with a simple `JMP` back to the top, but even that isn't strictly necessary. As you will see later, our embedded C programs never end

Listing 3

The Micro-C startup code for the Tiny memory model loads the segment registers, clears the uninitialized variables, and calls the `main()` function. I modified the code just after the `CALL` instruction to restart the program rather than invoke a DOS function we don't have.

```

*                ORG $100
Restart          EQU      *                *** restart here on main() exit
                MOV      AX,CS              Get CODE segment
                MOV      SS,AX              Set SS
                MOV      SP,#$FFFE          Initial stack pointer
                MOV      DS,AX              Set DS
                MOV      ES,AX              Set ES
* Zero uninitialized RAM area
                MOV      DI,#?temp          Point to uninitialized RAM
                MOV      CX,#?heap+1-?temp  Get size of uninitialized RAM
                XOR      AL,AL              Get a zero
                REP      STOSB              Repeat prefix
                CALL     main               Zero memory area
                JMP      <Restart           Execute main program
                *** bypass DOS exit!
* Exit function... Return to MSDOS
exit             XOR      AH,AH              Function 0 - exit
                INT      $21                Exit

```

and that `JMP` should never be executed. Nevertheless, I feel better with a valid instruction instead of a guaranteed trip to the weeds; call me compulsive.

Reloading the segment registers three or four times in quick succession is esthetically displeasing, but of no real consequence because the values are the same. The way I've set things up, each step in the chain is more or less independent of the others, so the new register loads appeared Just In Case the previous program isn't under my control.

Because the diskette boot loader handles only `COM` files, all the code and data must reside in one 64 KB segment. This implies all the segment registers have the same value, which is precisely the definition of the Tiny memory model. Micro-C can also produce `Small` model code, which uses separate Code and Data segments, but we will get along quite happily with a single segment until we explore BIOS extensions in Chapter 10.

Console I/O

The Micro-C runtime library includes the usual console I/O functions that connect `getch()` and `putch()` with the outside world. Despite the fact that they're in a file called `SerIO.ASM`, `getch()` delivers characters from the PC keyboard and `putch()` sends them to the video adapter. This makes sense for those embedded

applications that use PC hardware, but our code doesn't yet have the luxury of standard PC I/O. Remmber what's missing from Photo 1 in the previous chapter?

Fortunately, it's easy to substitute the corresponding serial I/O routines. While the BIOS serial functions are not adequate for real life serial I/O, they will suffice for our immediate purposes. Higher performance is, as Steve Ciarcia puts it, "a simple matter of software" that you can work on when the occasion arises.

Listing 4 shows the Firmware Furnace console I/O routines in `SerIOFF.ASM`. Use the `SLIB` utility from Micro-C's library utility to replace `SerIO.ASM` in `Tiny.LIB`:

slib i=tiny.lib a=serioff.asm r=serio.asm

Your code must call `serinit()` to specify the bit rate and COM port number. The data rate is limited to 9600 b/s, the maximum allowed by BIOS function `INT 14h`, `AH=00h`. For speeds up to 19200 b/s, use `AH=04h`. Beyond that, however, you must twiddle the serial port hardware directly.

Although `serinit()` can set up any one of the four standard serial ports, the typical integrated I/O board has two ports, so `MON86` uses `COM2`. You may use the same port for both your C program and the debugger, but the output can be confusing at times.

Finally, we are ready for a C program...

Listing 4

Micro-C's default console I/O routines use the target system's BIOS video and keyboard functions. This code substitutes the corresponding BIOS serial I/O functions, which, while not adequate for heavy-duty use, are good enough to get us started.

```
* Replacement for SerIO.ASM to use serial port
* Embedded PC's ISA Bus -- Ed Nisley
*
*-----
* Set up the serial port
* serinit(bitrate,COMport)
* Assumes 8N1 format
*
serinit      MOV      BP,SP           Get pointer to params on stack
              MOV      BX,DS           set up ES for SCAS
              MOV      ES,BX
              MOV      AX,4[BP]        fetch bit rate (stacked first)
              MOV      CX,#8           Set table length
              MOV      DI,#?bittab     Set table base
              REPNE    SCASW           Search for rate
```

Listing continues on next page

Listing continued from previous page

```

MOV     AL,CL
NEG     AL
ADD     AL,#7
MOV     CL,#5
ROL     AL,CL
AND     AL,$E0
OR      AL,$03
MOV     AH,#0
MOV     DX,2[BP]
DEC     DX
MOV     ?COMport,DX
INT     $14
* set up modem control outputs to ensure good communications
MOV     BX,?COMport
ADD     BX,BX
MOV     AX,$0040
MOV     ES,AX
MOV     DX,ES:[BX]
ADD     DX,$0004
IN      AL,DX
OR      AL,$03
OUT     DX,AL
RET

*
*-----
* Table of BIOS-acceptable bit rates
* The entry index is the BIOS bit rate code value
*
?bittab    DW      110      0
           DW      150      1
           DW      300      2
           DW      600      3
           DW     1200      4
           DW     2400      5
           DW     4800      6
           DW     9600      7

*
*
* Write a string to the serial port
putstr     MOV     BX,SP      Address stack
           MOV     SI,2[BX]   Get string
?putstr    MOV     AL,[SI]    Get char
           INC     SI         Skip to next
           AND     AL,AL       End of string?
           JZ      ?1         Yes, exit
           CALL    ?putch     Write it
           JMP     <?putstr   Do it all

*
* Write char with NEWLINE<>CR translation
*
putch      MOV     BX,SP      Address stack
           MOV     AX,2[BX]   Get character
?putch     CALL    ?putchr    Write value
           CMP     AL,$0A      Newline?
           JNZ     ?1         No, skip it
           MOV     AL,$0D      Add carriage return
           JMP     <?putchr    Write it out

```

Listing continues on next page

The Embedded PC's ISA Bus

Listing continued from previous page

```
*
* Write char with no translations
*
putchr      MOV      BX,SP           Address stack
            MOV      AX,2[BX]       Get character
* Write through IBM/PC BIOS call
?putchr     MOV      DX,?COMport    use our port
            MOV      AH,#$01        Write char to port
            INT      $14            Call BIOS
?1          RET
*
* Hang until a character arrives
* Return value is raw character in AL
*
waitchr     MOV      AH,#$03        Status request
            MOV      DX,?COMport    use our port
            INT      $14            Ask BIOS
            TEST     AH,#$01        Check data ready
            JZ       waitchr        Char is ready
            MOV      AH,#$02        Get character
            MOV      DX,?COMport    use our port
            INT      $14            Ask BIOS
            XOR      AH,AH          clear high byte
            RET
*
* Test for character (No translation)
*
chkchr      MOV      AH,#$03        Status request
            MOV      DX,?COMport    use our port
            INT      $14            Ask BIOS
            TEST     AH,#$01        Check data ready
            MOV      AX,-1          assume no char
            JZ       ?ccr           correct, bail out
            MOV      AH,#$02        Get char
            MOV      DX,?COMport    use our port
            INT      $14            Ask BIOS
            XOR      AH,AH          discard flags
?ccr        RET
*
* Receive char with no translations
*
getchr      CALL     waitchr         wait for character and get it
            RET
*
* Test for character (with CR<>NEWLINE translation)
*
chkch       MOV      AH,#$03        Status request
            MOV      DX,?COMport    use our port
            INT      $14            Ask BIOS
            TEST     AH,#$01        Check data ready
            MOV      AX,#0          assume nothing
            JZ       ?cch           correct
            MOV      AH,#$02        Get character
            MOV      DX,?COMport    use our port
            INT      $14            Ask BIOS
            XOR      AH,AH          discard flags
?cch        RET
```

Listing continues on next page

Listing continued from previous page

```

*
* Receive char with CR<>NEWLINE translation
*
getchr      CALL    waitchr      Wait for char and get it
            XOR     AH,AH        Zero HIGH
            CMP     AL,$0D       Carriage return
            JNZ     ?1          No, its OK
            MOV     AL,$0A       Convert to NEWLINE
            RET

*
* Receive a string: getstr(buffer, size)
*
getstr      MOV     BX,SP        Address stack
            MOV     SI,4[BX]     Get buffer address
            XOR     CX,CX        Zero CX
?2          CALL    getchr       Get character
            CMP     AL,$08       Backspace?
            JZ      ?3          Yes, handle it
            CMP     AL,$7F       Delete?
            JZ      ?3          Yes, handle it
            CMP     AL,$0A       End of line?
            JZ      ?4          Yes, handle it
            CMP     CX,2[BX]     Are we at end?
            JAE     ?2          Yes, don't accept
            MOV     [SI],AL      Write it
            INC     SI          Advance pointer
            INC     CX          Advance length
            CALL    ?putch       Write it out
            JMP     <?2         And proceed
* Delete character from buffer
?3          AND     CX,CX        At beginning?
            JZ      ?2          Yes, ignore
            MOV     AL,$08       Backup
            CALL    ?putch       Output
            MOV     AL,'#'       Space over
            CALL    ?putch       Output
            MOV     AL,$08       Backup
            CALL    ?putch       Output
            DEC     SI          Backup pointer
            DEC     CX          Reduce length
            JMP     <?2         And proceed
* Newline, terminate entry
?4          CALL    ?putch       Echo it
            MOV     >[SI],#0     Zero terminate
            MOV     AX,CX        Return length
            RET

$DD:?COMport 2

```

Why do some labels start with question marks? They mark local branch targets that will not be called from code outside this file. Refer to the Micro-C documentation for more details on the special treatment applied to those labels by the assembler.

Say "Hello!"

Back before C compilers (excuse me, Application Development Environments) required Windows 95, came on CD-ROMs, and soaked up hard disk space denominated in tenths of gigabytes, the first C program had one essential line:

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

Fortunately for us, Micro-C still retains that charming simplicity. `Hello.C`, shown in Listing 5, uses `printf()` to send out the obligatory message through the target system's serial I/O routines described above.

Our `Hello.C` program must not "fall out the bottom" of its main loop, because there is no operating system to regain control. As you'll see throughout this book, essentially all our embedded programs run continuously from power-on to shutdown. High octane embedded systems require real operating systems with dynamically loaded programs, but we don't need that level of performance just to explore the ISA bus.

`Hello.C` uses a `while` loop to hold the do-forever code. In this case we increment a counter, display the value, and call a BIOS function to waste about half a second.

Listing 5

The canonical First C Program prints "Hello, world!" and returns to the operating system. Our ISA-bus target system doesn't have an operating system, so, after this version of `Hello.C` displays its message, it begins an endless loop showing the contents of a simple counter. A BIOS function provides a half-second delay between displays.

```
#include <8086io.h>
#include "e:\mc86\custom\8086base.h"

#define LOOPDELAY 8 /* units of 64K microseconds */

unsigned int Counter;

main() {
    serinit(9600,1); /* set up serial port */
   _putstr("Hello, world!\n");
    Counter = 0;

    while (TRUE) {
        printf("Counter: %5u\r", ++Counter);
        asm {
            MOV     CX, #LOOPDELAY
            MOV     DX, #0
            MOV     AH, #86
            INT     15
        }
    }
}
```

The delay is essential because there is no point in sending continuous 9600 b/s data: each line would take only about 10 milliseconds, and you just can't read text off the screen that fast.

Micro-C includes a *Command Coordinator* that runs the preprocessors, compiler, assembler, linker, and so forth in the right order with the appropriate files, as well as a character-based Integrated Development Environment. I prefer the command-line compiler, mostly because that's what I started out with, and use this command line (in a batch file, of course) to recompile the program on the host system:

```
cc86 hello -ciklmop m=t
```

The result appears in file **Hello.HEX** with Intel **HEX** format. Boot **MON86** from the floppy in the target system, then use the **Load** command. Send **Hello.HEX** from your host system to the target system using your comm program's ASCII file transfer function, then do a **G 0100** to start **Hello**.

After you're comfortable with Micro-C, you may want to omit some of the compiler switches. The **-k** option keeps all the intermediate files, generating quite a clutter in your **\temp** subdirectory, and is of use mostly while you familiarize yourself with the compiler. The **-c** option includes all of your C language comments in the assembler output, which can inhibit the peephole optimizer's best efforts.

Because **MON86** uses COM2 for debugging output and **Hello** uses COM1 for normal output, you must have two serial ports on your host PC to monitor the action. This is one of the big advantages of a multitasking operating system like Windows 95 or OS/2: simply run two cables, start two comm program sessions, and watch their windows update simultaneously.

You can also use a PC with a single port and one comm program, but you must switch the cable from the target system's COM2 port to COM1 after starting **Hello**. You'll miss the **Hello, world!** output message, but the counter values should come through correctly and display on your host system.

You can use **MON86** to step through your C programs at the assembler level, set breakpoints, and dump memory. After you're sure the code is solid, use **HEXFMT** to convert the **HEX** file to a **COM** file just as you did for **Rotate.HEX**:

```
hexfmt hello.hex -b w=hello.bin
```

Copy that file to a diskette with the boot loader from Chapter 1, pop it in the target system's drive, and hit the Reset button. You should see the same output from COM1 as you did when running it under **MON86**.

At this point you have everything required to develop moderately complex embedded programs running on a standard x86 system board. The diskette boot loader reads a program from disk into the target system's RAM without using an operating system or specialized development hardware. Micro-C provides the programming essentials in a well documented, comprehensible package. Your program sends trace information through the serial port to your host system.

OK, ball's in your court...

Release Notes

The files for this chapter include all the demo program source code, `HEX`, and `BIN` files, as well as the serial port code in `SerIOFF.ASM`. There are several useful batch file to set up Micro-C's environment variables, too. See the `ReadMe.txt` files in both this chapter's subdirectory and the `\ISABus\Code\Micro-C` subdirectory.

You'll need Dunfield's 8086 Micro-C Developer's Kit to compile and assemble the source code. The debuggers described above are a part of the package, as well as numerous utility programs and debugging features that I haven't mentioned. See the Sources appendix for pointers to the DDS Web page.

Of course, you might prefer a different compiler. The disadvantage of the Borland, Microsoft, and similar high-end PC C compilers is that they are primarily intended for use with DOS and Windows. Adapting them to a DOS-less embedded system is possible, but it's not a trivial exercise. We'll explore this topic in much more detail in Chapter 11.

However, if you already have a favorite x86 C compiler, take a look at its `Tiny` model startup code and see what you can do. You must verify that the generated code has no DOS or Windows calls, which probably rules out the normal character I/O routines (if not all the startup code), and write your program without using any DOS-based library routines. Give it a try and see how it works!

3 Barnacles on the ISA Bus

When I first planned these projects, '386SX system boards ran at about 25 MHz. One month later, the slowest boards you could buy started at 33 MHz. Now, a few years later, 166 MHz Pentium boards are sliding toward obsolescence as the long awaited MMX CPUs arrive.

Even the rate of change is increasing!

Curiously, though, higher CPU performance has little to do with your firmware's I/O bandwidth. Just like barnacles on a ship's hull, a myriad of "PC-compatible" boards hold down the maximum speed of the ISA I/O bus. Yes, you can design a faster bus, but customers may not beat a path to your door.

The acronym ISA stands for "Industry Standard Architecture" which, in turn, is a euphemism for "IBM PC/AT Clone". Stripped of the hype, what you're getting is design dating from the late 1970's, tweaked several times over the years, and intended for a simple desktop microcomputer. If you demand blazing I/O performance, these barnacles are not for you.

The PCI (Peripheral Components Interconnect) bus used on all Pentium and Pentium Pro systems eliminates many of the restrictions found on the ISA bus. Previous attempts at improving the ISA bus, such as IBM's own Microchannel and the EISA (Enhanced ISA) bus, simply didn't have the right combination of timeliness, broad industry support, performance, and affordability. PCI, today's bus of choice for high-performance I/O, is certainly here to stay. Unfortunately, designing and building PCI bus I/O device requires far more technology, engineering knowledge, and test equipment than is reasonable for anyone other than a full time hardware designer.

But, as we'll see, the good old ISA bus runs entirely fast enough for many, if not most, embedded controller applications. In this chapter, I'll take a look at I/O performance as we begin constructing the Firmware Development Board that serves as the basis for the rest of the projects in this book. As with the firmware in the previous two chapters, I regard covering the fundamentals as more important than presenting a complete design in one shot.

The Intel and AMD CPU data books and Web sites mentioned in the Sources appendix provide many of the grim details required to actually design a system board, starting with a handful of chips. Fortunately for us, that level of detail is irrelevant because the PC Compatibility Barnacles define precisely what the CPU and its support chips must do to be *PC Compatible*. We can see the CPU only

through several layers of glue circuitry that must operate properly to keep all the various boards and programs working.

For our purposes, the ISA bus cycle definitions in Solari's *AT Bus Design* and *ISA and EISA Theory and Operations* books provide enough information to construct a useful ISA bus board. I used those books as the basis for much of the logic and timing information in these projects, although you should refer to the Bibliography appendix for more information on many other related books.

However, it behooves you to verify that these designs will actually work in your system: even when you follow a recipe exactly, you must still taste the soup before dishing it out. I can only hit the high spots here and trust that you will verify the detailed numbers for your own circuits.

I still recall making Oriental Pepper Soup from a recipe that called for 3 T of black pepper. All present agreed it was plenty hot enough, but perhaps three *teaspoons* would have sufficed. Could a “t” have turned into a “T” by mistake?

One Byte at a Time

The simplest ISA bus I/O operation occurs while writing a single byte to an output port. For simplicity, I'll use the target system's parallel printer port at address 0378 to illustrate the process, although it applies to any byte I/O operation. Connect the LED circuit shown in Chapter 1 to watch the parallel port's output in action.

Each iteration of the code snippet in Listing 1 produces two pulses on Bit 0 of the parallel port. With a 33 MHz system clock and a '386SX CPU, the first pulse lasts

Listing 1

This chunk of the porttime.c Micro-C program writes a pair of pulses to the parallel printer port. The first pulse uses C-level functions, while the second pulse shows how to use inline assembler to reduce execution time. You'll find the program and source code in this chapter's subdirectory.

```
while (!chkch()) {
    disable();
    outp(PortAddr,0x01);           /* rising edge          */
    outp(PortAddr,0x00);           /* falling edge         */
    asm {
        MOV    DX,PortAddr
        MOV    AL,$01
        OUT    DX,AL
        MOV    AL,$00
        OUT    DX,AL
    }
    enable();
}
```

3 μs and the second is only 1 μs . You should see a dim glow from the LED as the pulses repeat every 31.5 μs with a 13% duty cycle.

Although x86 CPU bus interfacing circuitry can be complex (just read the data book, yikes), the ISA I/O bus operates in a fairly straightforward manner. I connected a logic analyzer to the bus and took Photo 1 to illustrate this point.

The top trace is **BCLK**, the 8.33 MHz Bus CLock. The output operation begins with the rising edge of **BCLK** just before **BALE** (Bus Address Latch Enable), the second trace, goes high. **BALE** falls when the I/O address lines have settled. The third trace, the negative-active I/O Write signal, **-IOW**, clocks the data into the parallel port's output latch.

The port's output data, shown on the bottom trace, actually appears at the pin one **BCLK** cycle after **-IOW** ends. Evidently the integrated I/O chip uses **BCLK** to update its internal latches. As we'll see later, a latch built from discrete TTL chips presents its results immediately after the rising edge of **-IOW**.

Photo 1

This is the logic analyzer record of a single 8-bit write to port 0378. The output operation begins with the rising edge of BCLK just before BALE goes high and ends when **-IOW** goes high to latch the output data into the port. The integrated I/O chip used in this ISA bus system delays the actual output one additional BCLK cycle.

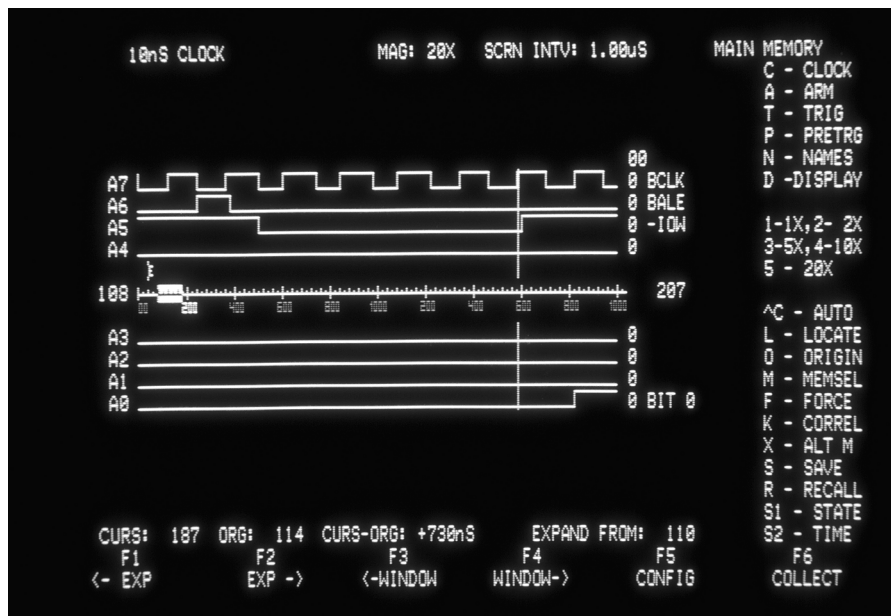
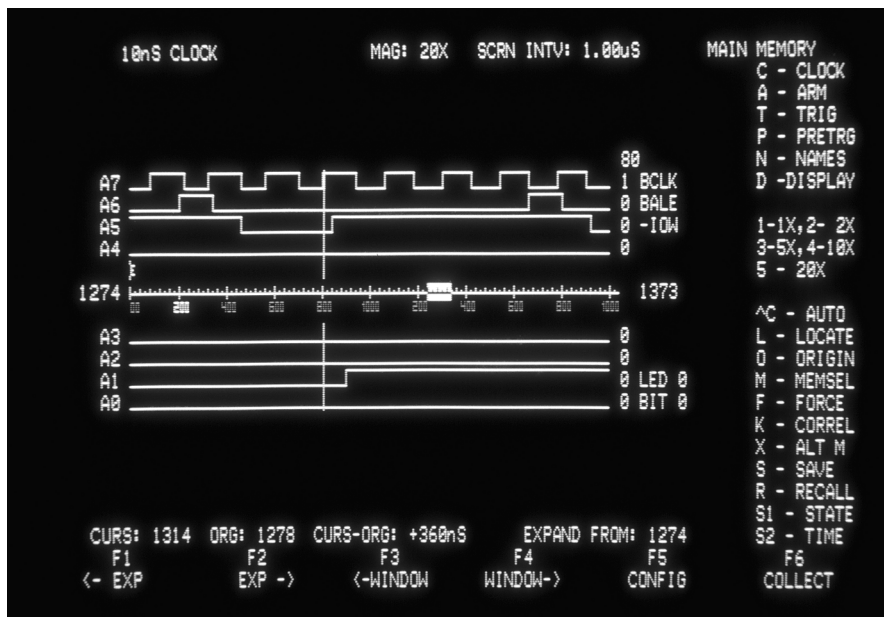


Photo 2

This photo records a single 16-bit write to the LED display on the Firmware Development Board. The operation is complete in three BCLK cycles and the data appears at the latch output immediately after -IOW rises.



Not counting the internal latch delay, the whole output operation takes six 120 ns bus clock cycles. The first two cycles set up the address and data, while the remainder provide enough time for the I/O device to accept the data. Byte reads require a similar amount of time, during which the device provides the data to the CPU in response to the -IOR bus signal.

The rather leisurely pace of a single-byte write (or read) may come as a surprise, but the nominal ISA bus cycle requires 720 ns. This ensures compatibility with older ISA bus boards designed for the original 4.77 MHz 8088 PC, while forming a real stick in the spokes for faster CPUs that can gnaw through dozens of instructions in that time.

Unfortunately, we have no way to speed up single-byte I/O cycles. The ISA bus includes a Synchronous Ready signal that can force no-wait memory accesses, but the system board ignores -SRDY during I/O operations. You will find some

references referring to **-SRDY** as **-ENDXFR** or **-NOWS**, but, whatever the name, pin B08 simply doesn't help.

IBM's PC AT designers widened the PC's I/O bus to 16 bits and, because none of the older 8-bit PC boards could respond on the new lines, the new cycles could run considerably faster. Photo 2 shows that writing a 16-bit value takes only three bus cycles from the first rising **BCLK** edge until the data appears in LED bit 0.

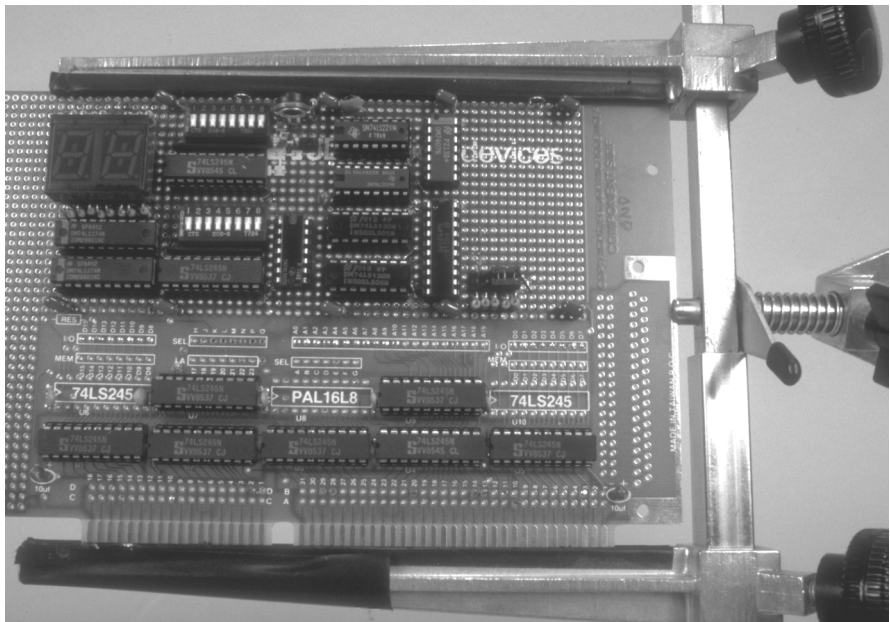
That's better, but we must first build some 16-bit hardware...

ISA Board Basics

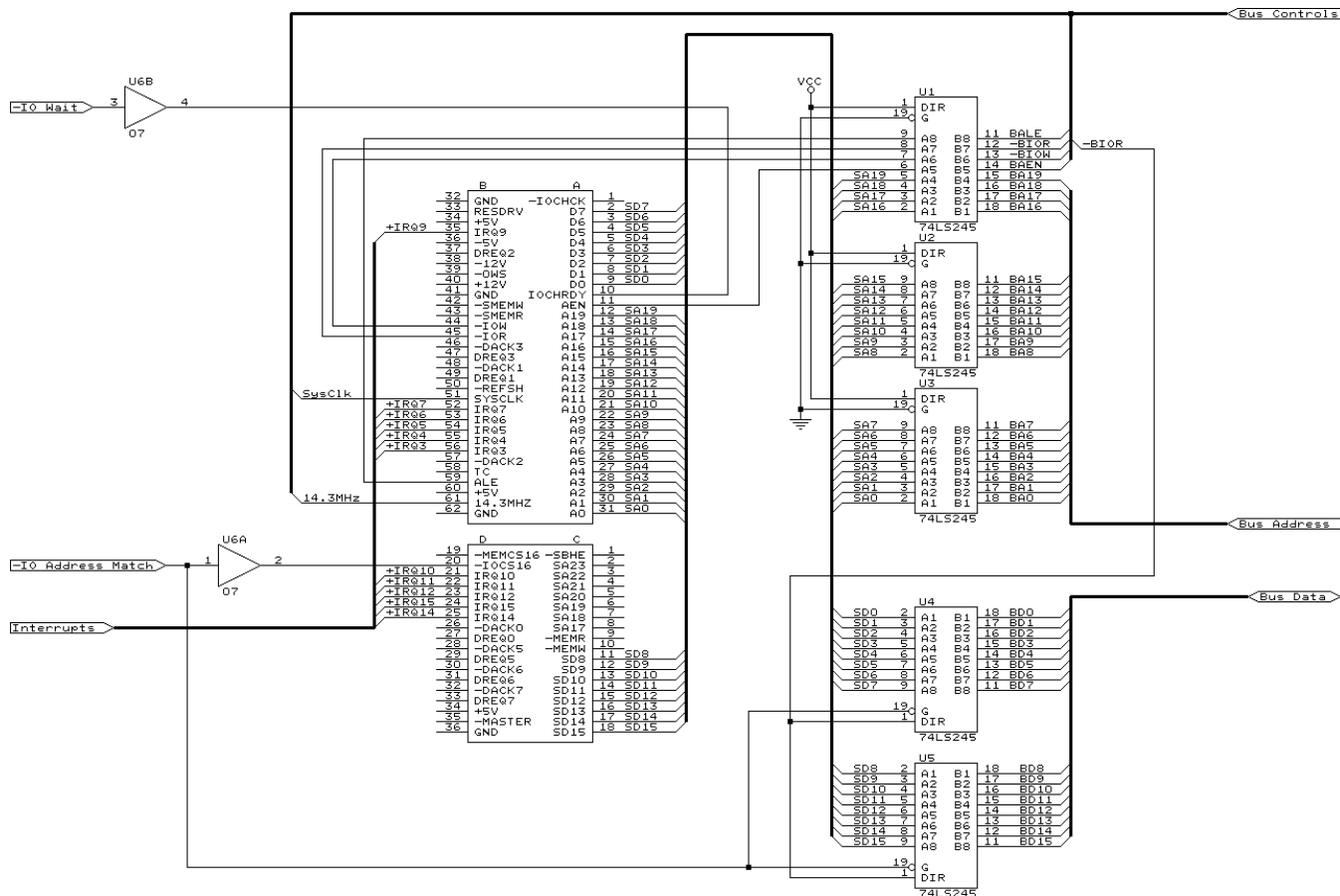
Photo 3 shows the end result: a 16-bit PC/AT prototype board with a handful of ICs, a pair of DIP switches for inputs, and a two-digit LED display for output. Despite using discrete chips, well over half of the board remains empty. We will have plenty of room for the rest of our circuitry, never fear.

Photo 3

I built the Firmware Development Board on a JDR Microdevices prototyping board, with the bus buffers arranged parallel to the connectors in the etched-circuit area and the address decoding and timing ICs near the top edge. The circuits you'll see in the rest of the book depend on the functions provided by these ICs, DIP switches, and LEDs.



The ISA bus signals can drive a maximum of two (or perhaps, four) LSTTL loads per slot. The LS245 transceivers on the address and control lines are hard-wired to drive signals to the board, while the address decoding circuitry controls the direction of the data line buffers. The system board provides pullup resistors for the IOCHRDY and -IOCS16 lines.



When you examine the Firmware Development Board's schematics, you'll be struck by the absence of "high tech" circuitry. I've deliberately used standard SSI gates and MSI decoders rather than PALs, GALs, PEELs, or other programmable logic devices. While those devices offer significant reductions in board space and wiring complexity, they tend to obscure the mechanics of what is really a fairly simple process.

Regardless of which programmable device I might pick, the majority of you folks won't have access to the programmer required to produce it. But you can implement these garden-variety circuits with whatever programmable logic devices you might have available. There have been enough articles about programmable logic devices in various sources that you should have no trouble getting from here to there.

The FDB (that's "Firmware Development Board") circuitry for this chapter divides neatly into three parts: a set of buffers to isolate the board from the I/O bus, address decoding and timing circuitry, and the LED and switch interfaces. The first two account for most of the logic; the remaining four ICs handle the actual I/O bits.

Buffering the Bus

Schematic 1 shows the FDB's ISA bus buffer circuits. The Original IBM PC AT bus specifications allowed up to two low-power Schottky TTL loads on each board, which the LS245 buffers handle quite nicely. The EISA bus spec allows up to twice that load on each board, but, if you are mixing logic families, you must calculate the actual currents for both high and low logic states. You should also consult the PC/104 signal loading specs, should your embedded project take that bus. In any event, keep your board's bus loading to the absolute minimum.

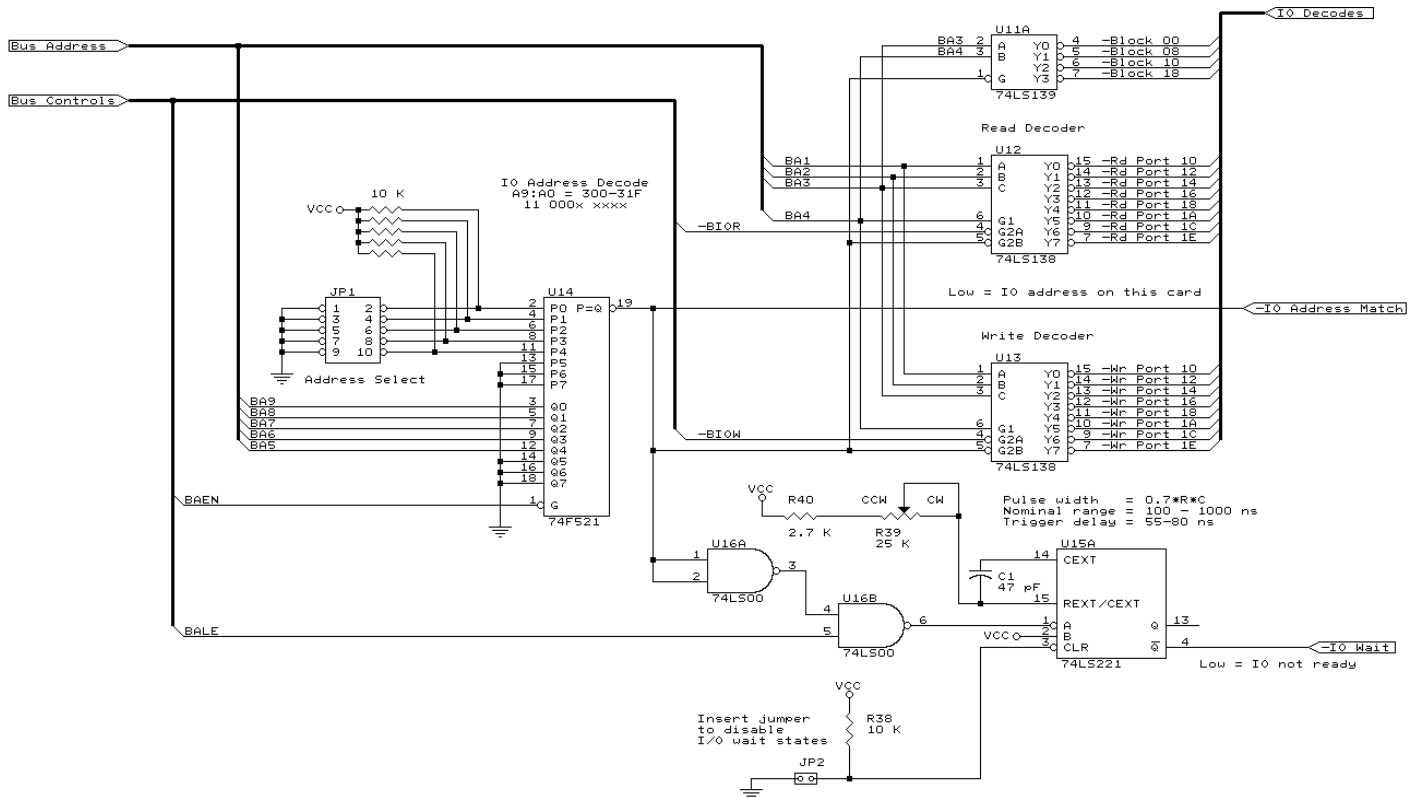
For example, in your own ISA bus designs, you may want to omit the buffers on some lines because you know the exact loading. In fact, you might think you can get away with excessive loading on your custom board in a specialized target system with, say, only two other ISA bus boards... that road is fraught with peril, as "just one more board" may push the bus loading over the limit. Rule of thumb: conservative design eliminates sleepless nights.

Notice that the three LS245 transceivers used for the address and control lines are always enabled and wired to send signals in only one direction. You may use LS541 buffers instead, but I prefer one less part number, particularly when LS245s are faster and less expensive. Check *your* suppliers for the latest prices and availability.

The signals on the inboard side of the LS245 drivers bear a **B** prefix to indicate that they are buffered from the bus. I prefix low-active signals with a minus sign, but as

Schematic 2

This I/O address decoder recognizes I/O port accesses to the FDB and activates the hardware corresponding to that port. Because the FDB contains only 16-bit ports (by definition), the card-selected signal activates the -IOCS16 signal shown in Schematic 1 for all addresses on the board. The firmware must avoid 16-bit accesses to odd addresses, although 8-bit accesses will work under conditions we'll explore in Chapter 4. You may replace the 74F521 with a 74LS688 if you note that the latter chip has a much longer propagation delay and will affect other circuits.



I mentioned in Chapter 0, you'll find that other references use leading or trailing asterisks, front or back slashes, overbars, and even more peculiar notations.

So much for standardization...

The two LS245 buffers on the data lines are enabled only when the address decoding circuitry detects an I/O access to a port on the FDB. When that happens, **-BIOR** determines the signal direction: the buffers drive data to the board, except, of course, when the CPU reads data from the board.

There are only two other signals leaving the board: **-IOCS16** tells the ISA bus interface that this board can handle 16-bit I/O accesses and **IOCHRDY** can stretch those cycles to accommodate slow devices. It may seem strange to talk about slowing the cycles when we're investigating faster I/O, but I'll explain this in Chapter 4.

Both of the control outputs use 7407 open collector drivers, because every ISA bus board can drive those signals. The PC system board includes the rather low value pullup resistors that each ISA I/O board must pull low to activate the signals.

Access Control

Schematic 2 should be more interesting, because the logic actually does something useful. These few ICs determine the I/O addresses of the devices on the FDB and control the bus access timing.

Although an *x*86 CPU can access 65536 (exactly 64×1024) I/O ports, ISA bus I/O boards decode only address bits **A0** through **A9**. As with the 8088's 1 MB of RAM, the 1024 ports defined by those ten bits seemed like a lot at the time.

Over the last decade, various manufacturers have assigned most of those 1024 I/O addresses to one device or another. Before you hardwire your new logic, you should check the references to see who else might be using *your* address. Often, you can overlay a device you're sure you (or your users) will never install. Be very careful when you produce a commercial product, though: those old widgets out there lie in wait for fatal assumptions like that!

The Firmware Development Board's default address range is 0300 through 031F, which corresponds to the ports on the original IBM prototype board. Just in case your target system already has something at those addresses (you aren't trying this in your *host* PC, are you?), a set of jumpers defines the address range. If you change the board's I/O address, you must modify and recompile the sample code, as we operate well below the level of Plug-and-Play hardware.

The CPU and the ISA bus can handle all combinations of 8-bit and 16-bit accesses to even and odd addresses, but each ISA board must include some logic to help out. Because we have complete control over the firmware, I decided to stay with 16-bit accesses at even addresses. It certainly simplifies the logic and, as you'll see in the next few chapters, does not mean the board cannot handle 8-bit devices.

U14, a 74F521 comparator, XORs address bits **BA5** through **BA9** with five jumper inputs and activates the **-IO ADDRESS MATCH** line when they coincide. That signal drives **-IOCS16** to enable the faster 16-bit accesses. That signal marks the only difference between 8-bit and 16-bit bus cycles.

Notice that U14 also activates **-IOCS16** during memory accesses when bits **BA5-BA9** match the jumper settings. That's the way it's *supposed* to work: the CPU's side of the ISA bus interface ignores spurious **-IOCS16** signals during memory accesses.

If you verify the circuit timings, you may be able to replace the 74F521 comparator with a 74LS688 comparator. Because 74LS circuits have significantly more delay than 74F, calculate and measure the changed timings carefully to ensure that the circuits we'll add in later chapters will work properly.

U12 and U13, the two LS138 decoders, combine U14's output with four more address bits to identify each of the sixteen 16-bit ports in the board's address range. The **-BIOR** and **-BIOW** signals enable the decoders, whose outputs indicate both address selection and bus timing.

U11A, half of an LS139 decoder, combines the comparator's output with two more address bits, making each output active for four consecutive 16-bit ports. This chip will come in handy later on when we add ICs with several internal addresses; the chip select signals can come directly from this decoder.

In a real design, you would use a programmable logic device to replace all four of these ICs. In effect, the PAL (or whatever) would implement a custom decoder with only the outputs your circuitry uses. Should you have access to a PAL programmer, feel free to adapt what you see here.

For boards that must handle both 8-bit and 16-bit accesses, consult the references to make sure you cover all the possibilities. That type of logic is hard to do with discrete gates and is tailor made for a PAL, just two of the reasons I avoided it. In truth, though, the design you see here is just another example of a tradeoff between hardware and firmware.

The remaining circuit, U15A, takes a step in the wrong direction. The LS221 one-shot creates wait states to slow down the faster 16-bit I/O cycles. It's not essential for the circuits in this chapter, but I decided to include an analog wait state generator. This way, you can vary the delay continuously and observe the resulting wait states, rather than be stuck with fixed delays from a digital delay line.

Essential I/O

Schematic 3 shows two I/O devices which (just barely) demonstrate that the FDB works. The LEDs provide visible indicators for sixteen output bits and the DIP switches control sixteen input bits. Both ports live at address 031E, presenting the firmware with write-only LEDs and read-only switches.

You may use sixteen discrete LEDs instead of the two-digit, seven-segment display I picked. The firmware in upcoming chapters will display error codes and status indications on these digits, but your application may be different.

Driving LEDs directly from an LS374 latch is, admittedly, unusual. Each latch output bit can sink up to 24 mA at normal logic levels, making the 10 mA for each segment entirely acceptable. The digits have separate common anode connections, both tied to the +5 V supply and current limiting resistors in series with each LED.

Note that LS374 outputs cannot *source* more than a few milliamps each, so they cannot drive common cathode LED digits.

Stepwise Construction

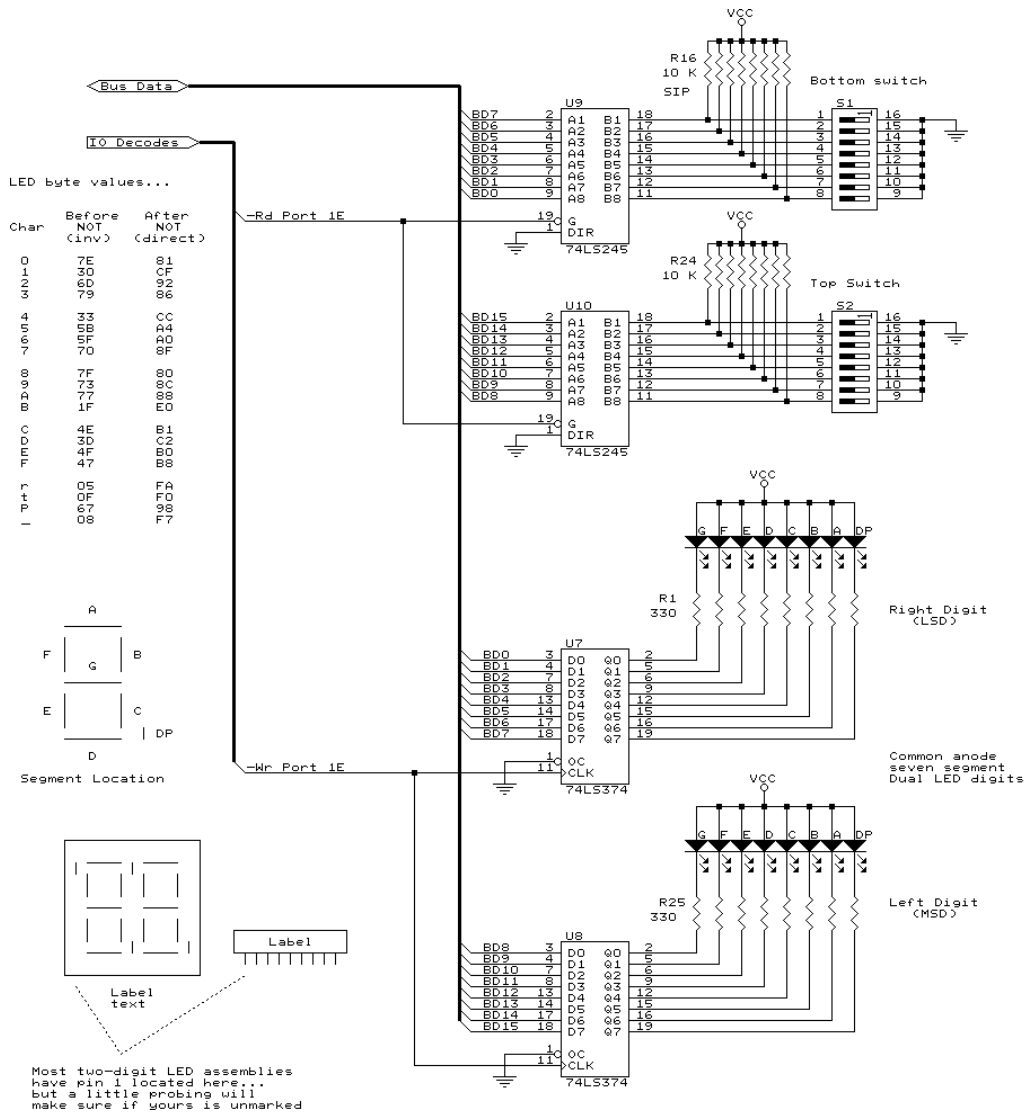
Reading about all this is one thing, but experience remains the best teacher. Fire up those soldering irons!

I used a JDR Microdevices prototype board as the basis for the circuitry shown in this chapter. Although the board etching includes a complete PAL-based bus interface, I decided to present a slightly different and somewhat more general design. As a result, my board has several cuts and adds that match it to the schematics shown here and in the Schematics appendix.

You can also use a bare board, without etched circuitry, similar to those made by Vector, but I *strongly* recommend ground and power planes covered by a good solder mask. Boards with etched buffer circuits reduce the number of wires you must cut, strip, and solder enough to make them worthwhile, even if you don't use their address decoding circuitry.

Schematic 3

The I/O circuitry shown here is the bare minimum required to verify that the FDB's buffers and address decoders are working correctly. The LEDs may be either common anode, seven-segment digits as shown or discrete diodes. The tables show the digital output values that produce some recognizable characters. You can use similar circuitry to build digital inputs and outputs that interface with the real world.



One of the big advantages of designing a plug-in board for a PC is that you can use the computer to debug the thing. I wrote `PortTest.C` to exercise each section of hardware as I built it. Proceeding step-by-step may not be as challenging as debugging the whole affair at once, but the job gets done faster with less hassle.

The code in Listing 2 should give you an idea of how to proceed. We start by exercising the LED digits with a visible test pattern, so you can tell at a glance whether the hardware works correctly. That's a debugging technique you won't find in the manuals, by the way...

You will, however, need an oscilloscope or logic analyzer to verify some of the other tests. For example, checking the F521 address comparator requires writing to addresses for devices on the board as well as for ones elsewhere in the address space. The only way to see those pulses (or the lack of them) is with an oscilloscope or well-handled logic probe. Of course, if your wiring and chips work properly, then you won't need to debug anything.

In any event, check out the code in this chapter's subdirectory before you begin construction. If you build the hardware in the same order as the tests, you should have no problems.

Listing 2

Writing firmware to exercise new hardware simplifies the debugging process. This section of `PortTest.C` blinks the LED digits, then lights each segment in turn. Because the firmware drives the LEDs directly from digital outputs, without intervening seven-segment decoders, it can control each segment independently. Simply watching the digits can reveal short-circuits or missing wires without any other test equipment.

```
case '3' :
    putstr("Single bit chasing through LED\n");
    putstr(" segment order is DP, then ABCDEFG\n");
    do {
        outpw(BOARD_BASE+LED_OFFSET,0xFFFF);          /* all segments off      */
        waitTicks(10);
        outpw(BOARD_BASE+LED_OFFSET,0x0000);          /* all segments on       */
        waitTicks(10);
        outpw(BOARD_BASE+LED_OFFSET,0xFFFF);          /* all segments off      */
        waitTicks(10);
        outp(SYNC_ADDR,0x01);                          /* sync on LPT port      */
        for (Counter = 0x8000; Counter; Counter >=> 1) {
            outpw(BOARD_BASE+LED_OFFSET,~Counter);      /* bit low = on          */
            waitTicks(7);
            Option = chkch();
            if (Option) {
                break;
            }
        }
        outp(SYNC_ADDR,0x00);
        waitTicks(10);
    } while (!Option);
    break;
```

Delayed I/O

The final section of `PortTest.C` exercises the LS221 wait state generator I mentioned earlier. Although the LS374 and LS245 ICs can keep up with normal 16-bit I/O rates, they also work with slower accesses. This makes them ideal experimental subjects: nothing can go wrong!

Most of the time you want I/O to run as fast as possible, but, once in a while, you must interface a device that cannot keep up with the normal bus rate. The **IOCHRDY** bus signal holds the key to slowing things down.

If **IOCHRDY** remains high (*I/O Channel Ready*) during an access, the ISA bus interface will complete the cycle normally. However, if it goes low (*I/O Channel **not** Ready*) the interface provides additional time. A pull up resistor on the CPU board holds the line active (*Ready*), allowing designers to entirely omit that circuitry from I/O boards that don't use the signal.

The board must pull **IOCHRDY** low within 76 ns after the leading edge of **-IOW** or **-IOR** during 16-bit accesses. The bus interface samples **IOCHRDY** on each rising **BCLK** edge and extends the **-IOR/IOW** signals by whole 120 ns, **BCLK** cycles.

The LS221 has a maximum trigger delay of about 80 ns, far too slow to fire it directly from the **-IOW/IOR** signals. Instead, I opted for a genuine kludge: it's triggered from the falling edge of **BALE**, when the I/O address comparator signals a match. Because the addresses don't stabilize during the early part of the **BALE** pulse, the LS221 may trigger on F521 comparator glitches that occur while the address settles.

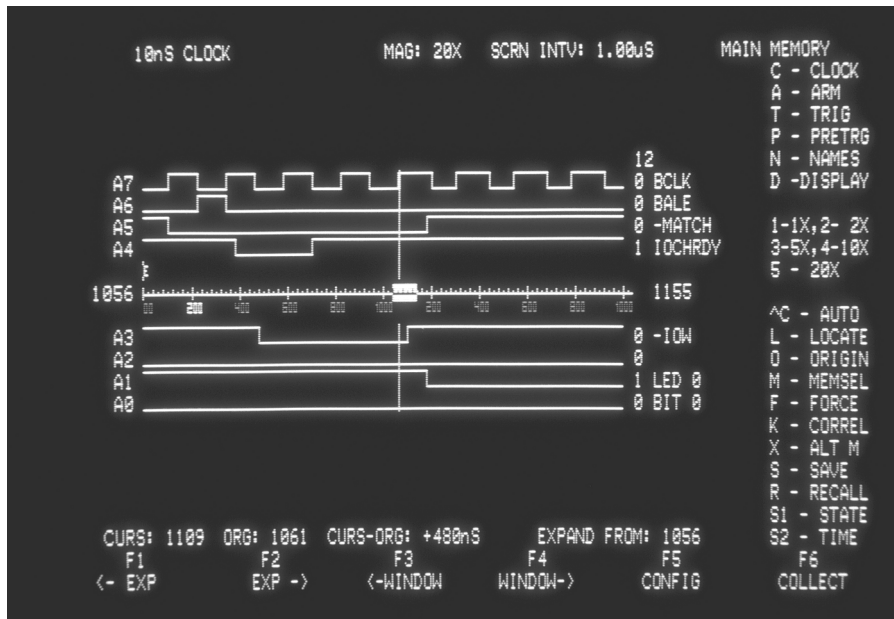
It turns out that this doesn't matter for our present purpose, because the only effect is to slow down a bus cycle that should run at full speed. Because there is no data loss and we're examining delays in detail, I decided to overlook the glitches in the interest of producing an easily variable delay for this chapter.

Photo 4 shows a 16-bit write with one wait state. Compare that with Photo 2 to see how **IOCHRDY** affects the results. Of course, you can't see the slowdown by eye, because the additional delay amounts to only 120 ns... those LEDs certainly won't look much different, even to a well-trained eye.

However, the wait state calibration routine in `PortTest.C` provides a way to adjust the number of wait states without a scope or logic analyzer. The test loop shown in Listing 3 counts the iterations completed in each 54.9 ms BIOS timer tick. As you increase the number of wait states by adjusting R39, the trimpot controlling the LS221 one-shot, the number of iterations decreases.

Photo 4

If the IOCHRDY line is low for at least 76 ns after the -IOW pulse starts, the ISA bus circuitry adds wait states to the 16-bit access. This write cycle takes four complete BCLK cycles due to the added wait state. Compare this cycle with the one in Photo 2.



Listing 3

Adding wait states to I/O accesses reduces the bus speed, allowing slower I/O devices to work properly. This test loop from PortTest.C displays the number of iterations completed in a single 54.9 ms BIOS timer tick. Increasing the number of ISA bus wait states decreases the number of iterations.

```
while (!chkch()) {
    outp(SYNC_ADDR, 0x01);          /* sync on printer port          */
    NextTick = waitTicks(1) + 1;    /* lock on next edge            */
    Counter = 0;
    do {
        outpw(BOARD_BASE, Counter++);
    } while (peekw(0x0040, 0x006C) != NextTick);
    outp(SYNC_ADDR, 0x00);
    printf("Count %5u\r", Counter);
}
```

The components I chose for the LS221 limit the **IOCHRDY** pulse between 100 and 1000 ns, which should be enough for most devices. **IOCHRDY** must not be held low more than 15.6 μ s during any access to prevent interference with the system's RAM refresh signals. We're safe from that problem.

The number of iterations obviously depends on the CPU speed. A 33 MHz '386SX completes 8080 (no, I'm not making that number up) iterations with no additional wait states. Remember to install the jumper that disables the LS221 before making this measurement, as component tolerances may produce a pulse long enough to trigger a wait state, even with the trimpot fully counterclockwise.

Simple math: 54.9 ms divided by 8080 iterations tells you that each iteration takes about 6.79 μ s. I counted 28 instructions in the assembler listing, indicating that the CPU trundles along at 4.1 million instructions per second or about 243 ns per instruction. Each instruction lasts for some multiple of the 30 ns CPU clock, implying about eight clock cycles per instruction.

Removing the jumper and adding one wait state drops the count to 7970 iterations and 6.89 μ s per loop. The difference, 100 ns, seems somewhat less than the 120 ns you'd expect.

I think the precise difference works out to 90 ns, three CPU cycles, due to the instruction prefetch queue being empty when the CPU encounters the **OUT** instruction. Because the CPU flushes the queue when it **CALLs** the **outpw()** routine and the instructions that get the address and data from the stack are fairly fast, the CPU actually goes idle during part of the nominally no-wait-state **OUT** instruction. The added wait state partially overlaps the prefetch delay for the next instruction, resulting in less lost time than the book indicates.

Adding a second wait state produces 7830 iterations or 7.01 μ s per loop. The difference here, exactly 120 ns, suggests that the prefetcher got itself back on track and had the next instruction ready on time.

I'll leave it to you to continue adding wait states and calculating the results. The bottom line, even in this contrived example, shows that each additional wait state adds less than 2% to the overall loop time. The CPU spends so little time in I/O operations compared to the other calculations in the loop that an additional 120 ns, just four CPU cycles, is no big deal.

Performance measurement on the complex CPUs found in the *x86* series is not at all trivial, but this simple example shows that if you *really* care about the results, you must make careful measurements. We'll see further examples in upcoming chapters, but for background reading you should get a copy of Abrash's *Zen of Assembly*

Language or his more recent *Zen of Code Optimization* and commit the lessons found therein to heart.

Of course, a digital delay line is the correct way to produce a short delay when you know precisely how long it should be. Simply send **-IOR/-IOW** through the delay line, XNOR the delayed output with the input, and drive **IOCHRDY** with the result. The pulse goes low when **-IOR/-IOW** goes low and returns high when the delayed signal arrives. You must include address decoding and so forth, but the output becomes a fast, low-delay, well-controlled blip that lasts just long enough for your I/O device.

The Digi-Key catalog includes a selection of digital delay lines that run \$10-\$20 in onesie-twosie quantities, depending on the delay, number of taps, and precision. You can see why I picked a one-shot for this adjustable delay, but I don't want you using an LS221 in your design just because you saw it here! Rummage through your catalogs and see what's available to produce the exact delays you need.

Release Notes

Refer to the code diskette for all the test programs. As before, I used Micro-C and you can run the **HEX** files on your target system with the **MON86** debugger or by copying the **BIN** files on a diskette with the appropriate **BootSect** loader from Chapter 1.

You can also copy the **BusLabel** file on the other side of this page and paste it on your ISA bus extender to simplify your ISA bus signal probing. The raw ASCII equivalent in **BusLabel.asc** will work with your favorite word processor, if you set the printer's line spacing at 0.1 inch to match the ISA bus connectors.

Bus Labels

This label for your bus extender identifies the ISA bus connector pins. The pins are exactly 0.1 inches apart, with 0.4 inches between the two connectors. Beware of copiers that slightly enlarge or reduce their images without warning.

B - Bottom	A - Top
01 Ground	-IOChCk
02 ResDrv	SD 7
03 +5 V	SD 6
04 IRQ 9	SD 5
05 -5 V	SD 4
06 DRQ 2	SD 3
07 -12 V	SD 2
08 -End Xfer	SD 1
09 +12 V	SD 0
10 Ground	IOChRdy
11 -SMemW	AEN
12 -SMemR	SA 19
13 -IO W	SA 18
14 -IO R	SA 17
15 -D Ack 3	SA 16
16 DRQ 3	SA 15
17 -D Ack 1	SA 14
18 DRQ 1	SA 13
19 -Refresh	SA 12
20 SysClock	SA 11
21 IRQ 7	SA 10
22 IRQ 6	SA 9
23 IRQ 5	SA 8
24 IRQ 4	SA 7
25 IRQ 3	SA 6
26 -D Ack 2	SA 5
27 TC	SA 4
28 BALE	SA 3
29 +5 V	SA 2
30 14.4 MHz	SA 1
31 Ground	SA 0

D - Bottom	C - Top
01 -MemCS16 -SHBE	
02 -IO CS 16	LA 23
03 IRQ 10	LA 22
04 IRQ 11	LA 21
05 IRQ 12	LA 20
06 IRQ 15	LA 19
07 IRQ 14	LA 18
08 -D Ack 0	LA 17
09 DRQ 0	-MemR
10 -D Ack 5	-MemW
11 DRQ 5	SD 8
12 -D Ack 6	SD 9
13 DRQ 6	SD 10
14 -D Ack 7	SD 11
15 DRQ 7	SD 12
16 +5 V	SD 13
17 -Master	SD 14
18 Ground	SD 15

4 I/O Time

Admiral Grace Hopper, of COBOL fame, often handed out one foot lengths of wire during her lectures, as she observed that light travels about one foot in one nanosecond. Those wires (now collector's items, of course) helped her put electronic speeds on a human scale: "So *that's* what 70 ns means!"

It turns out that electronic signals in open wire travel at about one third the speed of light in vacuum, covering four inches every nanosecond. One of the wait states we discussed in the previous chapter reaches to the far wall of your domicile, a 16-bit I/O access stretches across the street, and an 8-bit bus access ends well down the block.

In this chapter, we'll add an 8-bit 82C54 timer chip to the Firmware Development, explore several more ISA bus timing issues, and introduce a handy BIOS timing facility you may not have met before.

Keep Admiral Hopper's wires in mind...

Where Does Time Come From?

Timing on PC class machines has always been a problem, because the facilities are only slightly above rudimentary. The system board includes three timers (either a real 82C54 or, more recently, a smidge of circuitry on an LSI support chip), but only one channel can generate an interrupt. Worse, that timer is tied into such diverse features as the BIOS time-of-day routine and the diskette motor turnoff delay, so tinkering with it can be hazardous to your program's operation.

The IBM PC AT added an MC146818A Real-Time Clock to keep track of the date and time while the system power is off. The chip (or its moral equivalent in LSI) can produce periodic interrupts on **IRQ 8** at rates ranging from 8.192 kHz down to 2 Hz. The BIOS implements an alarm clock function based on that interrupt, which will come in handy for this code.

While you can certainly use the PC's system-board timers in your code, they are best left for the normal BIOS services (unless, of course, your code doesn't use the normal BIOS services, in which case anything goes). I decided to add a separate 82C54 timer chip to support the Graphic LCD Interface you'll meet in Chapter 12, but in the meantime, we'll put the standard PC facilities to good use.

Photo 1 shows the Firmware Development Board with all the circuitry thus far.

The 82C54 timer, essentially identical to the non-CMOS 8254 on the original AT system board, has an 8 MHz clock rating. You must, however, pay close attention to the entire part number. A -2 suffix indicates a 10 MHz clock speed and a slightly faster bus interface. A -5 suffix brands it as a 5 MHz slowpoke that won't work in this circuit. No, the suffixes *don't* make sense; these chips were born back in the bad old days before rational “dash” numbers appeared.

Although It Would Be Nice to have a sensible clock frequency, I decided to use the 14.318 MHz signal from the ISA bus. Pardon the digression, but I must explain why that frequency is what it is.

The Original IBM PC's designers used an Intel 8284A clock generator on the system board to produce the 8088's CPU clock. The 8284 includes a crystal oscillator and produces a CPU clock at 1/3 of the crystal's frequency. Out on the Color Graphics Adapter, they needed a 14.318 MHz signal for the composite video

The 82C54 timer chip shown here uses the address decoding and bus buffering circuitry presented in Chapter 3. The LS245 drives the ISA bus interrupt request lines, so make sure another device in your target system doesn't use the same signals!

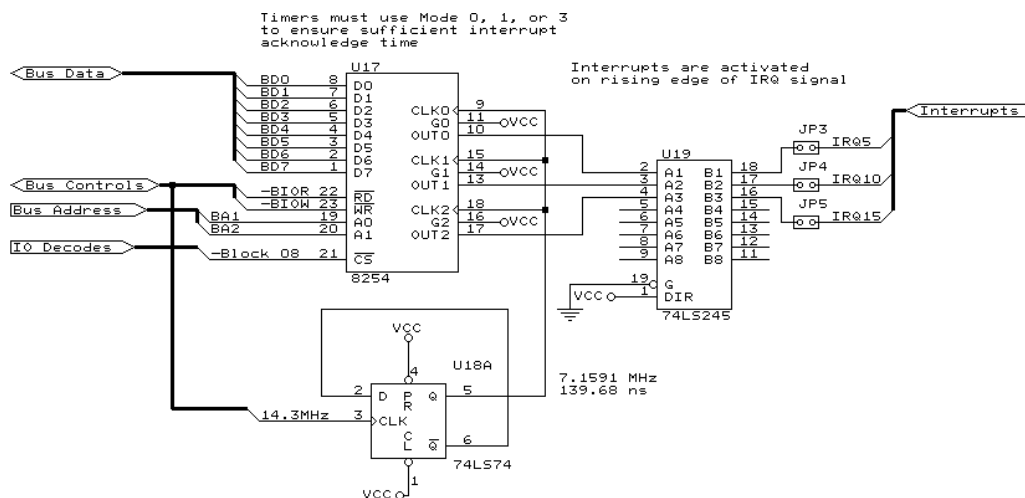
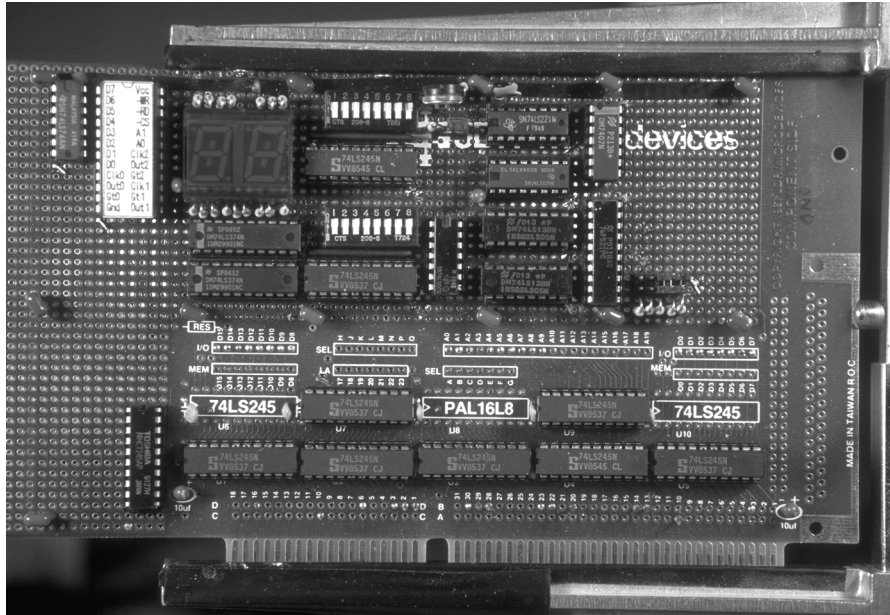


Photo 1

Adding an 82C54 timer, a bus buffer, and a few flipflops requires only three new DIPs. Comparing this with Photo 3 in Chapter 3 shows the additional circuitry to the left of the LEDs and bus buffers. Note the additional bypass capacitors on the ground plane.



signal that could drive a standard television. So... why not run the 8284 at 14.318 MHz and, thus, the 8088 CPU at 4.77 MHz?

The 8284 also produced a **PCLK** output at half the CPU clock rate, or 2.39 MHz. Although the Original PC's 8253-5 timer chip could handle that rate (it went all the way up to a blazing 2.6 MHz, despite the -5 suffix), the longest possible period between BIOS timer interrupts would then be 27.5 ms. That seemed a bit peppy, given that the PC's 8088 CPU could execute only 8000 instructions in that time.

So... they used a flipflop to divide **PCLK** by two, producing a 1.19 MHz clock, which the 8253 then divided by 64 K counts to generate an interrupt every 54.9 ms. That's why every PC in the universe has such a bizarre BIOS timer tick period.

The rest, of course, is history.

When the IBM PC AT came along, its designers could have boosted the BIOS tick rate to take advantage of the new-and-improved 80286 CPU's horsepower. By that time, however, far too many programs depended on that 54.9 ms rate. Only a whole new operating system can change the clock rate; anything less will be chained by historical necessity. And, as you might imagine, those new operating systems had better exhibit absolute backwards compatibility with existing programs.

But, even though the PC's Compatibility Barnacles limit the system board 82C54 to 1.19 MHz, we have no such restriction. The 82C54 has an 8 MHz clock rating, so half of an LS74 flipflop chops the 14.3 MHz bus signal down to 7.16 MHz.

Thus, each of our three timer outputs can produce interrupt periods from about 140 ns to 9.15 ms under firmware control. Admittedly, the faster rates aren't particularly useful, as even a firebreathing Pentium Pro can't accomplish much in 140 ns, but the improved timing resolution will turn out to be a Good Thing.

All three outputs drive interrupt request lines on the ISA bus connector through a LS245, which, in this application, serves as a simple buffer. I didn't include a way to disable these interrupts using software, because I assume that you're building this board for a particular reason and you'll hardwire the interrupts based on your requirements. You can, of course, install hardware jumpers that select different **IRQ** lines on the ISA bus, then disable them entirely by yanking the jumpers.

Because **IRQ 5** may be used by printer ports and sound boards, make sure your target system doesn't have any conflicts with the 82C54 outputs.

Figure 1

The Firmware Development Board now has three I/O devices: sixteen LEDs, a matching set of DIP switches, and an 82C54 timer. Because the address decode logic in Chapter 3 assumes 16-bit I/O accesses, each I/O port must appear at an even address. The 82C54 timer's four internal registers appear as the low-order byte in four consecutive 16-bit I/O ports and may be accessed by either 8-bit or 16-bit bus cycles.

Port	Read	Write
0308	Timer 0 Count/Status	Timer 0 Count
030A	Timer 1 Count/Status	Timer 1 Count
030C	Timer 2 Count/Status	Timer 2 Count
030E	nothing	Timer Control Register
031E	Switches (16-bit)	LED digits (16-bit)

I/O Addressing

The Firmware Development Board's address decoding logic requires 16-bit I/O operations directed to even addresses, but you don't need an I/O device for each and every data bit within a given port. For example, I wired the 82C54 to the low-order byte of the data bus and left the high-order bus byte disconnected. The hardware ignores the high byte during writes and the firmware must ignore it during reads.

The 82C54 has four internal addresses that are usually accessed as successive one-byte I/O ports. On this board, however, they must be located at successive *even* addresses, because the FDB's decoding logic cannot handle *byte* reads or writes to an *odd* address. As you can see in Schematic 1, the 82C54's **A0** and **A1** inputs come from the buffered address lines **BA1** and **BA2**, respectively.

Figure 1 shows the I/O port mapping for the devices we've added so far. The timer uses 8-bit I/O operations, while the LEDs and switches accept 16-bit operations.

You will recall that the only difference between 8-bit and 16-bit I/O operations is that the board activates **-IOCS16** to indicate that it can handle both bytes. If **-IOCS16** remains inactive, the system board breaks 16-bit operations into two 8-bit accesses by writing the low-order byte to the first address and the high-order byte to the next address.

The system board swaps the bytes around to put the proper bytes at the proper I/O locations, but the Firmware Development Board does not include the circuitry to handle its end of this dance. If you must support all possible I/O accesses, check the references for the details and prepare to spend a while designing that circuitry.

If the CPU is executing an 8-bit I/O operation, it ignores the **-IOCS16** signal and simply writes the byte to the specified address. Because the 82C54 connects to only the low-order byte of the ISA bus, it will respond correctly to either 16-bit or 8-bit I/O operations directed to even addresses. The high-order byte will be mush during 16-bit accesses, but in this case we don't care.

The only other difference is the bus timing, with 8-bit I/O operations requiring twice as many cycles. As it turns out, those extra cycles are critical to properly using the 82C54 on this board.

Timing from Both Sides

Because the 82C54 is the first nontrivial I/O device on the Firmware Development Board, we should explore some of the details that ensure it actually works. Even if you're in the firmware business, you must (at least) be able to interpret the hardware specs... perhaps to point out bugs that the hardware folks must fix?

The timing values and diagrams in this chapter come from Solari's *ISA & EISA Theory and Operation*, a weighty tome that replaces his earlier *AT Bus Design* book. Despite the fact that it costs about \$90, it's essential if you're doing this sort of stuff.

Figure 2a shows the (considerably simplified) timings for a 16-bit ISA bus I/O read operation and Figure 2b shows the corresponding timings for the 82C54's read operation. As is traditional in these hardware timing diagrams, the horizontal (time) axis is not drawn to scale and you can't compare intervals by eyeball.

Figure 2a

ISA bus timings for a 16-bit I/O read operation. Obviously, the time axis is not to scale!

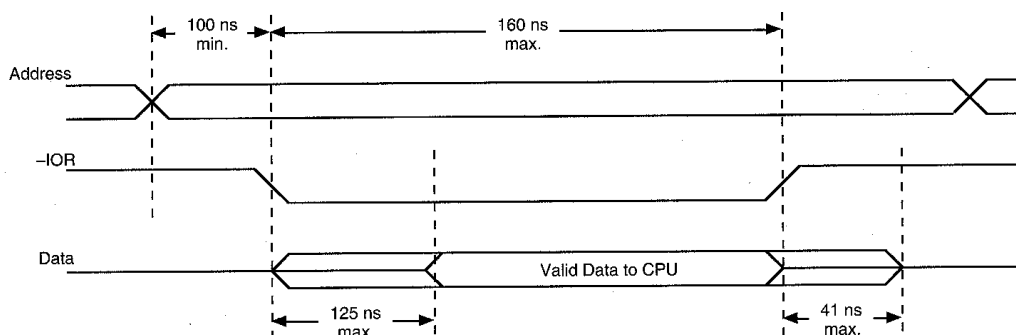


Figure 2b

82C54 read cycle timings, showing the 220 ns delay from the address lines and 120 ns delay from -RD to the first valid data output. This is barely within the limits shown above, but does not include the ISA bus buffer and address decoding times.

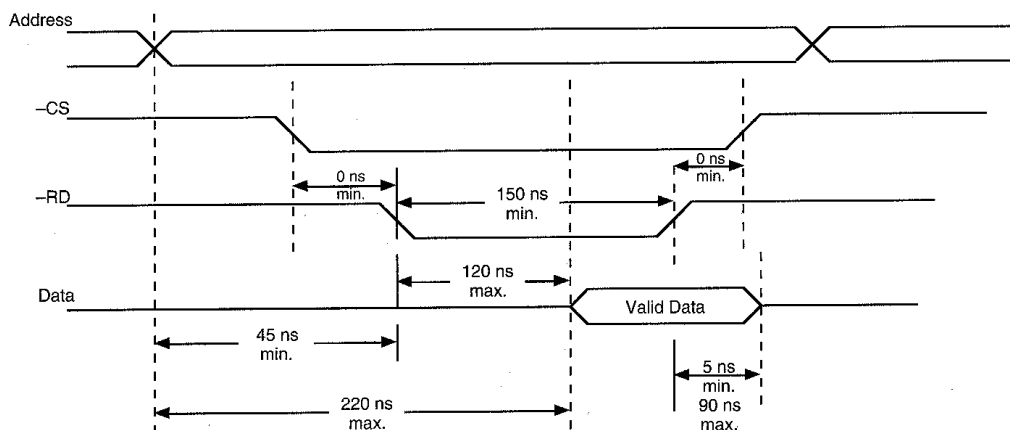
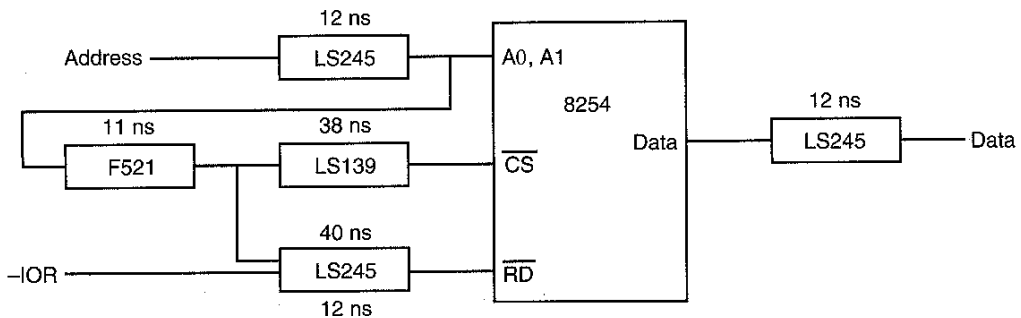


Figure 2c

This simplified Firmware Development Board schematic shows the delay through each IC involved in a ISA bus read operation. The two values shown for the LS245 in the Data path correspond to the DIR and Data inputs. You must add the delays shown in Figure 2a to the total delays along each path to find the total times.



According to Figure 2a, the ISA bus address settles at least 100 ns before **-IOR** goes active. At most 125 ns later, the data from the board must become valid and remain valid until well after **-IOR**'s trailing edge. However, the data drivers must turn off no more than 41 ns after **-IOR** goes inactive, thus preventing contention with the next cycle's signals.

As you can see in Figure 2b, however, the 82C54's output doesn't become valid until 220 ns *after* the address stabilizes or 120 ns *after* the beginning of the **-RD** pulse, whichever comes *later*. This may look close enough, but remember that the 82C54 isn't directly connected to the ISA bus.

Figure 2c shows the delays produced as signals pass through some of the key ICs supporting the 82C54. The LS245 buffers add a 12 ns delay to the address lines on the input side and another 12 ns on the data output, making valid data lag its address by about 244 ns. Comparing that with Figure 2A tells you that this setup just won't work.

That's precisely why wait states were invented. Recall, from Chapter 3, that a single wait state adds 120 ns to the duration of **-IOR**, ample time for the 82C54's data to arrive and settle down before the bus cycle ends.

There are several other paths though the logic that merit checking, such as the bus address to **-CS** path versus the **-IOR** to **-RD** path. In order to verify that the chip will work, you must analyze *all* the paths through the chips and add up their

timings. Hint: even if you work the *New York Times* crossword puzzle in ink, use a pencil for this job!

Obviously, if you must do a lot of this timing, figuring, and checking, you will depend on an automated timing analysis program that verifies every path in your schematic using built-in delay tables for each IC and interconnection. It's expensive, but... better you should find problems *before* the board goes into production, than suffer the slings and arrows of outrageous glitches.

Figure 3a
ISA bus timings for a 16-bit I/O write operation. Again, the time axis is unscaled.

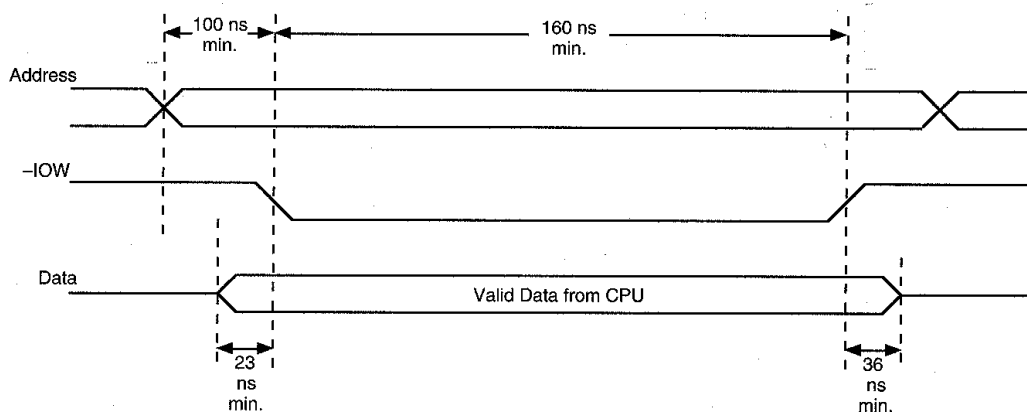


Figure 3b
82C54 write operation timings. This looks like it might work, if you ignore the buffer and decoding delays between the chip and the ISA bus signals.

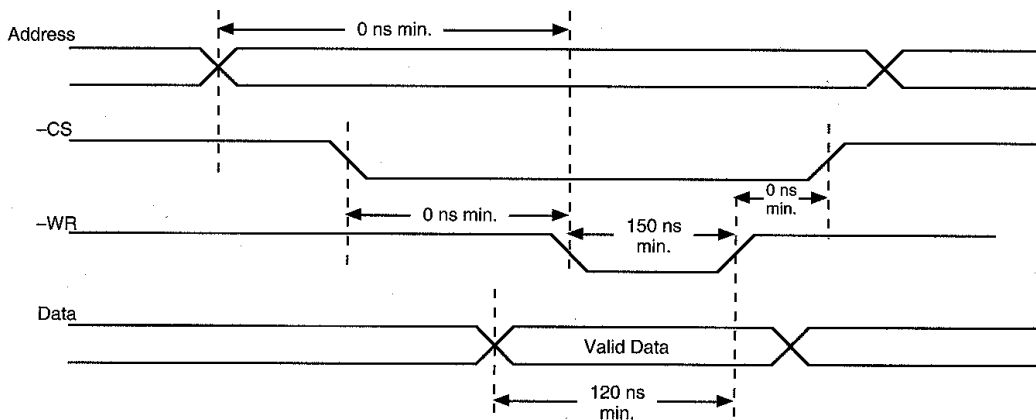
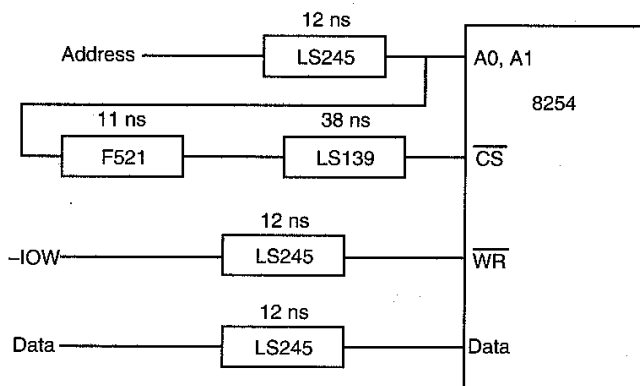


Figure 3c

These IC timings apply to ISA bus write operations. You must add the times shown in Figure 3a to the delays along each path to find the total times.



As an exercise, work through the write cycle diagrams shown in Figure 3 and see if an additional 40 ns of wait state will suffice here, too.

The simplified schematic in Figure 3c summarizes the circuit delays during a write cycle. Notice that **-IOW** gets there a bit faster than **-IOR** did during read cycles.

Now, here's the punch line. If you use 8-bit I/O operations, the ISA bus hardware inserts *three* extra wait states, with no extra effort on your part and no additional circuitry on your board. Basically, by picking the right I/O operation, we can tune the system to work correctly, even with this rather poky IC.

The down side: should you (or someone else, say, the poor soul maintaining your code after you've gone on to bigger & better things) substitute a 16-bit I/O operation, the access will fail. Worse, the timings seem pretty close, making it likely that some boards will work some of the time and others won't work most of the time. It's all a matter of the tolerances found on the system board and your chips.

Try to find *that* bug!

However, you'll find plenty of similarly sharp objects in this book, so I'll show how to use the hardware either way and trust you to do the right thing. This chapter's checkout program, `TimeTest.C`, can use either 16-bit or 8-bit I/O operations. You also have control of the FDB's wait state generator and can see just how well your hardware responds to various conditions. Oh, yes, comment your code, too.

Count the Ways...

Although the 82C54 registers and operations should be familiar to most of you, I'll provide a capsule summary to get us all to the same starting point. A complete "how to use the 82C54" exposition lies well beyond this book's charter... fortunately for us, we don't need quite *that* much detail.

The 82C54 contains three independent counter/timer channels, each with three pins: a **Clock** input, a **Gate** control input, and an **Output**. Unlike the counter/timers found in 8051 microcontrollers, these gizmos count *down*. Therefore, you just load them with the number of counts you want, rather than the two's complement of that value. If you prefer BCD to binary, you can have them count down from 9999 by decades instead of from FFFF in hexits. Honest.

Each 82C54 timer/counter channel can run in any one of six different modes, known, naturally enough, as Modes 0 through 5. Each mode uses the channel's three I/O pins in slightly different ways. You must match the channel's mode, register setup, and external hardware configuration to your application.

Mode 0 implements a simple counter: the firmware loads a delay value and the 82C54 decrements it by one count on the falling edge of each **Clock** pulse, as long as the **Gate** pin remains high. The channel's **Output** pin goes low when you load the counter and returns high when the counter hits zero. After that, even though the counter continues counting, the output pin won't change. This mode can generate precise delays for hardware gadgets, because the output can produce a single, clean, delayed edge with no additional hardware.

Mode 1 works the same way, except that a rising edge on the **Gate** input starts the count. In effect, this mode produces a precision delay starting from a hardware, rather than a firmware, trigger. Because I tied all the **Gate** inputs high on the Firmware Development Board, we can't try this mode without some rewiring.

Mode 2 divides the **Clock** input frequency by the programmed count value. It produces a single low **Output** pulse each time the count hits zero. The counter automatically reloads its initial value from an internal register and continues counting, giving you one pulse for every n **Clock** input pulses with **Gate** high. This comes in handy for dividing a clock frequency by a 16-bit value.

Mode 3, similar to Mode 2, produces a regular square-wave output with a period of n **Clock** cycles. Channel 0 of the system board's 82C54 uses this mode for the familiar 54.9 ms BIOS timer interrupt on **IRQ 0**. Note that the counter decrements by two, not one, on each **Clock** cycle. The **Gate** input can, thus, synchronize the **Output** phase to an external signal. I'll leave the **Gate** pins tied

Figure 4

Each Counter channel in the 82C54 uses configuration bits from the 8-bit Control Word written to the Control Register at address 030E. The Firmware Development Board, just like the PC itself, does not support all possible 82C54 modes and configurations!

The names for each Control Word bit are:

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

The SC bits select the Counter channel:

SC	Function
00	Counter 0
01	Counter 1
10	Counter 2
11	used for Read-Back commands

The RW bits specify the data I/O format for each counter. The counters are always 16 bits wide, but you can write or read a single byte in special situations:

RW	Function
00	used for Counter Latch commands
01	Read/write Counter MSB only
10	...LSB only
11	Read/write LSB followed by MSB

The M bits specify the Counter's operating mode, in the usual binary fashion, from 000 = Mode 0 to 101 = Mode 5. Specifying Mode 6 or 7 sets Mode 2 or 3, respectively.

The BCD bit selects the counting radix:

BCD	Radix
0	Binary, maximum count = FFFF hex
1	BCD, maximum count = 9999 decimal

high until we have something that demands synchronization, such as the Graphic LCD Interface circuitry.

Modes 4 and 5 are basically Modes 0 and 1 with a twist: the **output** remains high until the counter hits zero, at which point it produces a single low pulse.

If you plan to use an 82C54 with anything other than a regular, periodic, essentially square **clock** input, *read the data sheet carefully and pay attention to the clock specs*. It turns out that the 82C54 has several gotchas that become obvious only *after* you've designed the thing into a circuit and it behaves, well, strangely. See Chapter 13 for an example of how **clock** input irregularities can affect an 82C54 counter.

Setting up an 82C54 counter channel is straightforward: just write an 8-bit *Control Word* to the Control Register, which resides at I/O address 030E on the Firmware Development Board. Figure 4 shows the Control Word bit layout. You must write one Control Word for each channel, then write one or two bytes to the channel's address to set the initial counter value. On the FDB, the three counters reside at addresses 0308, 030A, and 030C.

Hardware Checkout

Until we begin wiring up the Graphic LCD Interface hardware in Chapter 13, the 82C54's Mode 3 will suffice to exercise the hardware and illustrate the firmware techniques. Listing 1 shows the code required to set up all three counters for square-wave output.

Listing 1

This Micro-C code fragment shows how to initialize all three 82C54 channels to Mode 3. The `#define` macro eliminates a good deal of repetitive typing and ensures that you write the proper bytes in the correct sequence to the appropriate ports. Note that this code uses 8-bit I/O operations to ensure adequate bus timing.

```
#define LOADTIMERB(i,c,t) outp(I8254_BASE+6,c); \
    outp(I8254_BASE+2*i,t & 0x00ff); \
    outp(I8254_BASE+2*i,t >> 8);

putstr("Timer 0 = 1 ms square wave @ pin 10\n");
putstr("Timer 1 = 2 ms square wave @ pin 13\n");
putstr("Timer 2 = 3 ms square wave @ pin 17\n");

outp(SYNC_ADDR,0x01);          /* sync at start          */

LOADTIMERB(0,0x0036,7159);     /* 1 ms @ 7.1591 MHz      */
LOADTIMERB(1,0x0076,14318);    /* 2 ms @ 7.1591 MHz      */
LOADTIMERB(2,0x00B6,21477);    /* 3 ms @ 7.1591 MHz      */

outp(SYNC_ADDR,0x00);
```

Figure 5

The 82C54 includes latches that capture both bytes of the counters, allowing two successive 8-bit read operations to return valid 16-bit data, regardless of the delay between the reads. A Read-Back Command Word written to the Control Register at address 030E activates the counter latches. Note that you can latch all three counters with one command. If bit 4 is zero, the first byte you read from the selected Counter channels will be the Status byte.

Bit	Function
7	Must be 1
6	Must be 1
5	0 = latch current Count of selected Counter channels
4	0 = latch current Status of selected Counter channels
3	1 = select Counter 2
2	1 = select Counter 1
1	1 = select Counter 0
0	Must be 0

Figure 6

The Status Byte read back from the counter channels gives you enough information to figure out what the counter is doing. Bits 5-0 echo the bits previously written to the counter's Control Word. In the code for this chapter, we use bit 7 to discover the state of the Output pin without connecting it to an input port.

Bit	Meaning
7	State of Output pin (0 = low, 1 = high)
6	Null Count (1 = not counting, 0 = counter loaded)
5	RW1
4	RW0
3	M2
2	M1
1	M0
0	BCD

The `#define` macro bottles up the three `outp()` functions that load the 82C54 Control Word and write both counter bytes for a single channel. If your code space gets unreasonably tight, for whatever strange reason, you can convert this macro to a function call.

After that code starts the timers, fire up your oscilloscope to verify the outputs. Should your test bench lack a scope, Listing 2 may be of more interest, because it measures and displays the results on your host system using the target PC's timers. If what you see on the host looks good, the hardware probably works OK... but, if it fails, prepare to cadge a scope from someone. You can use a frequency counter to

Listing 2

This routine computes and displays output frequencies by reading the status of each counter's Output pin. The code uses the BIOS alarm clock routine shown in Listing 3 to measure one second of real time while it checks for counter Output pin transitions.

```
putstr("Timers should produce 1000, 500, and 333 counts/sec\n");
putstr(" +/- one or two counts for sampling error...\n");
putstr("LEDS display iteration counter in hex\n");
putstr("Uses 8-bit I/O operations\n");

outp(SYNC_ADDR,0x02);          /* sync at start          */
LOADTIMERB(0,0x0036,7159);    /* 1 ms @ 7.1591 MHz      */
LOADTIMERB(1,0x0076,14318);   /* 2 ms @ 7.1591 MHz      */
LOADTIMERB(2,0x00B6,21477);    /* 3 ms @ 7.1591 MHz      */

Counter = 0;
while (!chkch()) {

    Status[0] = Status[1] = Status[2] = 0;
    Edges[0] = Edges[1] = Edges[2] = 0;

    if (SetAlarm(0x000F,0x4240,&Alarm)) { /* alarm in 1 second */
        printf("Cannot use BIOS alarm function, code %02x\n",Alarm);
        break;
    }

    outp(SYNC_ADDR,0x01);
    while (!Alarm) {
        outp(I8254_BASE+6,0x00EE); /* latch all status bytes */
        for (Timer=0; Timer<3; ++Timer) {
            PortValue = inp(I8254_BASE+2*Timer) & 0x0080;
            Edges[Timer] += (0 != (Status[Timer] ^ PortValue));
            Status[Timer] = PortValue; /* save for next pass */
        }
    }

    outp(SYNC_ADDR,0x00);
    printf("\r%5u %5u %5u counts/sec",
        Edges[0]/2,Edges[1]/2,Edges[2]/2);

    outpw(BOARD_BASE+LED_OFFSET,~ByteToSegs(Counter));
    ++Counter;
}
putstr("Counters continue to run...\n");
```

measure the pulses, although a scope will give you a better idea of the actual operation and reveal any marginal signal levels.

The 82C54 can return a *Status Byte* for each counter that indicates, among other things, whether the counter's **Output** pin is currently high or low. Figures 5 and 6 show the Control Word and Status Byte values you'll use. A short routine can easily count the number of **Output** flips per second to calculate and display the output frequency of each channel.

(Now, why do you write an 8-bit Control *Word* and read an 8-bit Status *Byte*? Because, just because... it didn't start with me!)

Listing 3

The BIOS INT 15h Function 83h provides an alarm clock for PC programs. This routine shows how to set the delay and specify the address of a byte that will change when the time expires. The code is mostly assembler within a Micro-C wrapper.

```

/*----- */
/* Set up BIOS real-time alarm clock to delay interval in microseconds */
/* Returns 0 if OK, error code if not (also sets *pbAlarm to return code) */
/* High-order bit of *pbAlarm is set when delay time expires */
/* Remember to split the microsecond count manually... 1 sec = 1000000 us */
/* The asm{} section starts with BP pushed and loaded with SP */
WORD SetAlarm(DelayHi, DelayLow, pbAlarm)
WORD DelayHi;
WORD DelayLow;
BYTE *pbAlarm;
{
    DelayHi; /* keep compiler happy */
    DelayLow;
    pbAlarm;

    asm {
        MOV     BX, SP          aim at the stack
        MOV     CX, 8[BX]      get delay high byte
        MOV     DX, 6[BX]      ... low byte
        MOV     BX, 4[BX]      get pointer to alarm byte
        MOV     AX, DS          ... set up segment
        MOV     ES, AX
        MOV     AX, #8300      Set Event wait Interval function
        INT     15

        JC      ?seterr        C set on error
        XOR     AL, AL          ... clear return flag

    ?seterr    MOV     BX, SP          set up for return code
               MOV     BX, 4[BX]
               MOV     ES: [BX], AL
               XOR     AH, AH        clear high byte of return value
    }
}

```

Remember the MC146818A Real-Time Clock I mentioned earlier? Here's one place it comes in handy. BIOS Function `Int 0x15, AH=0x83` uses the periodic interrupt from that chip to set a delay; when the delay expires, the BIOS interrupt handler flips bit 7 in the byte of your choice. This relieves your code of the need to worry about interrupts: after you set the alarm, you simply test the target byte whenever it's convenient during your program's action loop.

The code in Listing 2 sets up the three 82C54 timers with periods of 1, 2, and 3 ms, then enters a loop that ends when it gets a byte from the serial port. Until then, it sets up a 1 s alarm and enters an inner loop that latches and reads back the current status for each channel. Although the 82C54 latches all three channels at once, you must fetch the three Status Bytes with three separate read operations.

Determining when each **Output** pin changes requires nothing more than XORing the current state with its value during the previous iteration. The **Status** array holds that information for each timer, while the count in the **Edges** array increments on each change.

Listing 3 shows the code that invokes the BIOS alarm clock. That function expects a four-byte delay value, measured in microseconds, and the `seg:off` address of a byte to change when the delay expires. Because the Micro-C compiler doesn't support `long` (four-byte) variables, I put the high and low halves of the 32-bit delay value into two 16-bit parameters.

When the BIOS function sets bit 7 of the **Alarm** byte after one second, the inner loop ends. Each **Output** cycle produces two counts in the **Edges** array, making the displayed frequency half that amount. The code sends the results through the serial port to your host system, sets the BIOS alarm again, and begins accumulating another set of counts.

You will see slight variations in the displayed totals as the program runs. The software counts, not being synchronized to the hardware's **Output** pulses, may miss or gain one count at the beginning and end of each second. However, if the totals differ from the correct values by more than a few counts, look for something gone badly wrong.

Just for fun, I tossed in the `ByteToSegs()` function that converts a byte into a pair of hex digits on the seven-segment LED display on the Firmware Development Board. A line of code sends the iteration counter through that routine and gives a running count on the LEDs. As an exercise, adjust the code to display the digits backwards and upside down so they read correctly from the *top* of the board...

Release Notes

`TimeTest.C` includes a variety of tests I used to get the 82C54 running and verify the bus timings. You must have a scope to check some of the routines, but using just the self-testing code *should* get you on the air if you do careful wiring.

The circuitry shown in this chapter differs slightly from that in Chapter 13, where we'll adapt the 82C54 timer to drive a bitmapped, graphic LCD panel. You must check out the timer using the schematics and code from this chapter, then modify the wiring as shown in the back of the book.

Take stock of what you have working on your Firmware Development Board: I/O address decoding, a pair of LED digits, some DIP switches, and a trio of high-resolution hardware timers. So far, so good!

5 After This Brief Interruption

Stop me if this hasn't happened to you.

When the clothes dryer buzzed, I decided to take a break. The laundry room light went nova, so I detoured to the garage for a new bulb. On my way through the machine shop, I perched the laundry basket on the workbench while pocketing some outdoor pole lamp parts I'd fixed the previous evening.

I punched the garage door opener, dug a new bulb out of the lamp stash, walked down the driveway to reassemble the widget, then retrieved the day's mailbox treasures. Mary leaned out of the kitchen door to say the call was for me. I passed the mail to her and snagged the phone on the way by.

Several hours later Mary stuck her head in the office and asked, "Why is the garage door open and what have you done with the laundry?"

We pros call this a *blown stack*, although, to be fair, there are other interpretations...

The Firmware Development Board now has enough hardware that we can investigate something that often goes unmentioned: what *really* happens when a hardware interrupt occurs? I'll concentrate on real mode and leave protected mode for a different book.

Our first task must be nailing down the nomenclature. You have probably read about interrupt handlers, exceptions, traps, faults, **IRQs**, **INTs**, and vectors, but the definitions are often either vague or just plain wrong. After settling those issues, we can examine the code that responds to interrupts. Finally, I'll tell a war story about PC interrupts that should curl your keyboard.

Let's start this discussion at the beginning, all the way down at the bare silicon.

The Inside Story

Intel 80x86 CPUs handle interrupts from several sources: external events, instructions, and internal problems such as the dreaded divide-by-zero error. Fortunately, the CPU uses the same basic mechanism in all situations.

External interrupts occur when hardware outside the CPU raises its **INTR** (Interrupt Request) pin, which may happen at any time. The CPU hardware activates its **INTA** (Interrupt Acknowledge) pin and reads a single byte from the data bus. That byte identifies the interrupt source and can, in principle, funnel up

to 256 different external interrupts through the single **INTR** pin. In the PC, as we will see later, only 15 external interrupts contend for the CPU's attention. The CPU ignores **INTR** when the Interrupt Enable bit (a.k.a. the Interrupt Flag or **IF**) in the **FLAGS** register is zero.

Unlike external interrupts on the **INTR** pin, Non-Maskable Interrupts arriving at the **NMI** pin cannot be ignored. There is only one Non-Maskable Interrupt, because the CPU does not read an ID byte from the data bus. Hardwired circuitry (or, more precisely, microcode) inside the CPU causes **NMI** to produce **INT 02h**. Although the CPU cannot disable **NMI** inside the chip (hence the name), the PC system board includes circuitry external to the CPU for that purpose.

Software interrupts occur when the CPU executes one of a class of instructions devoted specifically to causing them. The instructions have mnemonics like **INT 10h**, **INTO**, **BOUND**, and so forth. Like interrupts on the **NMI** pin, software interrupts cannot be ignored. Unlike external interrupts on the **INTR** pin, they are entirely predictable: whenever the CPU executes the instruction, the corresponding software interrupt ensues.

Finally, errors or similar conditions within the CPU trigger exception interrupts. These interrupts are generally data dependent, so a given instruction may not cause an exception every time. However, they can be reproduced if you set all the hardware to precisely the same state. Exceptions cannot be ignored or suppressed.

Each interrupt, regardless of cause, corresponds to a number from 00h through 0FFh (in C, that's 0x00 through 0xFF) called, oddly enough, its Interrupt ID, interrupt type, or just plain interrupt number. By convention, Interrupt IDs use hex notation, although some sources prefer the decimal equivalent.

However, I've seen one reference that managed to convert a *hex* Interrupt ID into *decimal*, then listed the *decimal* value as *hex*. Moral of the story: you must have more than one book on your shelf to crosscheck things that seem out of whack. To help prevent a similar misinterpretation in this book, I'll always indicate that the interrupt ID is in hex: **INT 02h** or **INT \$02**, as appropriate.

An interrupt's *raison d'être* (you should pardon my French) is diverting the CPU's attention from its current task and setting it to work on something else, something that is, presumably, more important. That something, the code associated with each interrupt, bears the name *interrupt handler* or *interrupt service routine* (a.k.a. **ISR**). As a general rule, interrupt handlers should be short routines that cope with whatever triggered the interrupt, then return to the interrupted, lower-priority task as rapidly as possible.

An *interrupt vector* holding the handler's address provides the link between an interrupt source and its handler. In Intel 80x86 systems, the interrupt vector's address is simply the Interrupt ID multiplied by four: **INT 08h** uses the vector at address 0020 (hex, naturally). The collection of all 256 (decimal) vectors occupies 1024 bytes of storage and is referred to as the Interrupt Vector Table or, on '286 and higher CPUs, the Interrupt Descriptor Table.

Although *everyone* knows that the IVT starts with the **INT 00h** vector at address 0000:0000, it turns out that the **LIDT** instruction (Load Interrupt Descriptor Table) can set the table's starting address and maximum length, even in real mode. The power-on default, of course, matches the 0000:0000 and 1024 bytes found in the earliest 8086 CPU. While operating systems may find reason to relocate or resize the IVT, ordinary programs have little need of such shenanigans.

Your interrupt handler code must obey several rules, which we'll explore throughout this chapter, as it executes. The basic rule is simple: the interrupted program should resume execution as though the handler never got control, apart from the time required to complete the handler. Any changes the handler makes must occur to hardware or variables it controls, without affecting the bystanders.

Assuming you've written a good handler, the final piece of the puzzle involves returning from the interrupt handler. Exactly how you do that depends on which type of interrupt your code handles.

For all external and software interrupts, the CPU pushes the **FLAGS** register, the **CS** register, and the **IP** register onto the stack, with the stacked **CS:IP** indicating the address of the instruction that *would* have been executed had the interrupt *not* occurred. If an external interrupt arrived through the **INTR** pin, the CPU then clears the Interrupt Flag to suppress all further external interrupts. Although **NMI** interrupts do not actually clear **IF**, the CPU ignores the **INTR** pin until the **NMI** handler ends with an **IRET** instruction.

Exception interrupts, triggered by the CPU's internal hardware, come in three flavors: faults, aborts, and traps. You must spend some time with the references to understand the differences; you'll see only the first two in real mode, unless you're exceedingly unlucky. For our purposes, faults and aborts push the address of the *current* instruction, the one that failed. Conversely, traps push the address of the *next* instruction, just like external and software interrupts.

Although you can, in principle, fix up the condition that caused a fault and successfully re-execute the failed instruction, this can be difficult in actual practice. A trap, on the other hand, is over and done with by the time your handler gets control, meaning that you must continue with the next instruction, if possible.

With that in mind, the `IRET` instruction at the end of each handler restores the CPU's `FLAGS`, `CS`, and `IP` registers from the stack, precisely what we want regardless of the interrupt's cause. Restoring the `FLAGS` register from the stack restores the `IF` bit, which, if this was an external interrupt, re-enables further external interrupts.

To summarize: when the CPU detects an interrupt, it pushes the `FLAGS`, `CS`, and (usually incremented) `IP` registers on the stack, computes the interrupt vector address from the Interrupt ID (which may be supplied by external hardware, internal hardwiring, or the instruction), fetches the starting address of the interrupt handler from the IVT, and transfers control to it.

When the handler finishes its work, it executes an `IRET` instruction that restores the registers from the stack. The CPU picks up where it left off.

Piece of cake, yes?

The Rest of the Story

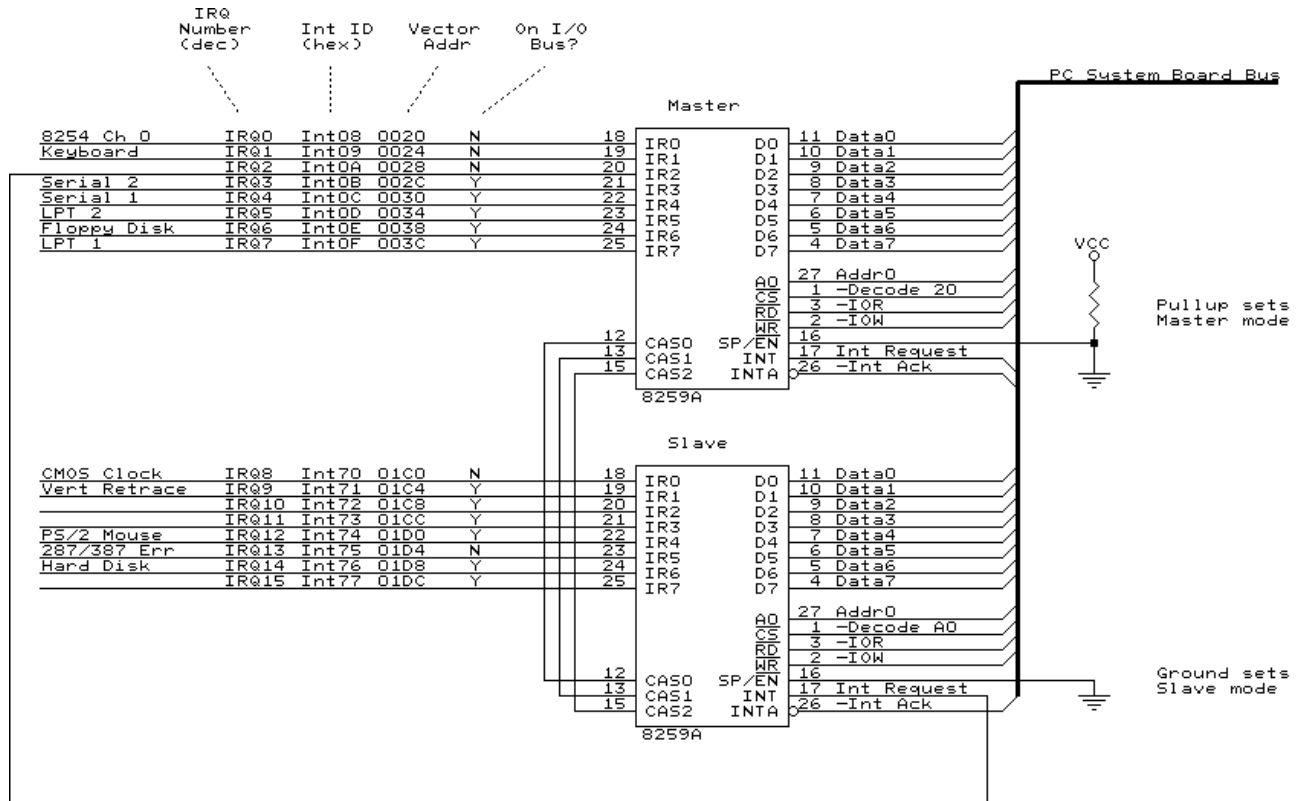
The Intel 8259 Programmable Interrupt Controller holds the key to understanding PC interrupts. As with any Intel peripheral bearing the term “Programmable” in its name, the 8259 is a maze of modes, options, control bits, and gotchas. I'll concentrate on how it behaves in a PC and leave the rest for an evening of data book spelunking. Note that, despite the obvious acronym, an Intel 8259 PIC has no relation to a PIC microcontroller made by Microchip Technology, Inc.

As shown in Schematic 1, the PC system board uses a pair of 8259s in tandem to provide 15 external interrupts. Of course, progress has long since subsumed the 8259 chips into an LSI package (along with all the other CPU support circuitry), but the PC Compatibility Barnacles dictate how that hardware must behave.

Word of warning: if you're contemplating doing anything at all out of the ordinary, remember that the 8259 data sheet does *not* necessarily apply to the LSI chips inside your PC. You should have the real specs on the actual LSI marvel you're using, as modes or functions not used in “normal” PC applications may not work quite correctly. Give them a go, but don't be surprised at unexpected results.

Indeed, even if you have the appropriate specs, don't be surprised if the hardware doesn't quite match the documentation. Often, the folks who design the hardware don't talk to the documentation writers and *neither* discuss matters with the testers who verify everything.

The system board circuitry devoted to interrupts boils down to a pair of 8259 Programmable Interrupt Controller chips or their LSI equivalents. This diagram summarizes the 8259 Interrupt Request (IRQ) numbers, the CPU Interrupt (INT) numbers, and the vector addresses for each external interrupt. Because the slave 8259 cascades through the master 8259's IRQ 2 pin, IRQ 8 through IRQ 15 have higher priorities than IRQ 3 through IRQ 7. The BIOS redirects IRQ 9 to the IRQ 2 vector maintain compatibility with older PCs; the pin that was IRQ 2 on PCs is labelled IRQ 9 on ATs.



Each 8259 is an I/O device with two internal addresses selected by the **A0** address line. Unlike the 82C54 circuitry in the previous chapter, you cannot use 16-bit I/O cycles with the 8259, because it expects separate I/O operations at each port. The master 8259 resides at addresses 0020 and 0021, with the slave at 00A0 and 00A1.

For what it's worth, the terms *master* and *slave* seem to have fallen out of favor lately. I must continue to use them here, because that's how Intel worded the 8259 data sheet. *Primary* and *secondary* may be more, ah, PC compatible, but you won't find them in the data books and references.

The BIOS initializes the 8259s for normal PC interrupt operation, which should suffice for most embedded PC code. Figure 1 shows an approximate diagram of an 8259 after the BIOS finishes its setup, minus all the control logic and special cases.

Each 8259 has eight Interrupt Request inputs called **IRQ 0** through **IRQ 7**, with the slave's **IRQ** inputs called **IRQ 8** through **IRQ 15** on the PC. By convention, the 8259's **IRQ** inputs bear decimal numbers. These **IRQ** numbers are simply labels with no mystical significance. As you'll see later, though, the slave's **IRQ** numbers are particularly inauspicious.

A rising edge on any **IRQ** input constitutes an interrupt request, so the 8259 turns on the corresponding Interrupt Request Register bit. You can force the 8259 to ignore specific combinations of **IRQ** inputs by turning on the appropriate Interrupt Mask Register bits. A 1 bit in the IMR *disables* the corresponding **IRQ**.

Assuming IMR does not mask the **IRQ**, the 8259 then activates its **INT** output, which raises the CPU's **INTR** input. As described above, that triggers a hardware interrupt if the CPU's Interrupt Flag is set. A 1 bit in IF *enables* the interrupt. The CPU blips the **INTA** once to tell the 8259 to resolve its highest priority interrupt.

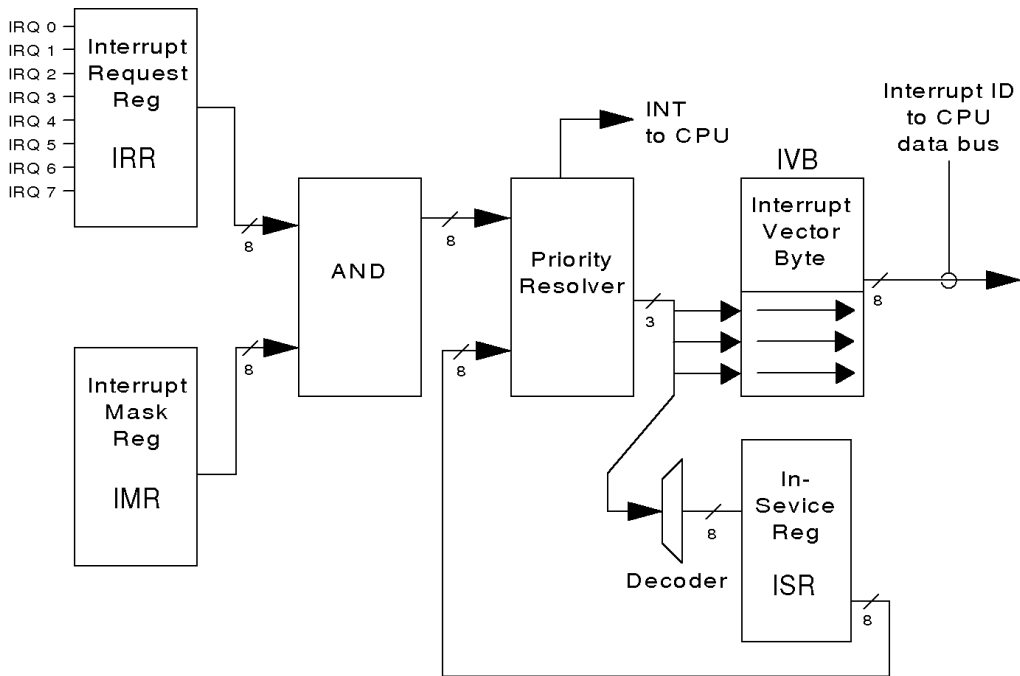
If two or more IRR bits are 1 simultaneously, the 8259 figures out which has the highest priority and saves the rest for later. The priority rules can be complex and may change on the fly if you reprogram the 8259, but, in normal PC operation, the rule reads *lowest numbered IRQ wins*.

A considerable amount of time may elapse between an IRR bit going active and the CPU's response. If additional IRR bits become active in the meantime, the 8259 will recognize the highest priority interrupt at the time of the first **INTA** pulse from the CPU. The other IRR bits remain active and will cause additional interrupts as they become the highest priority inputs in turn.

In any event, the winning IRR bit turns on the corresponding In-Service Register (also an ISR, not to be confounded with the Interrupt Service Routine described

Figure 1

Although the 8259 Programmable Interrupt Controller has a bewildering variety of modes and settings, this diagram shows roughly how it works in normal PC operation. A rising edge on an IRQ line sets an IRR bit. If the corresponding IMR bit doesn't mask it, the priority resolver decides if the new IRR has a higher priority than any of the bits currently set in the ISR. If so, it activates the 8259's INT output. When the CPU acknowledges the interrupt, the 8259 combines the IVB with the IRQ number and sends an Interrupt ID to the CPU, then turns on the appropriate ISR bit. At the end of the interrupt handler, an EOI command from the CPU resets the ISR and IRR bits. If any other IRR bits remain active, the entire process starts over again.



above) bit to indicate that the CPU has acknowledged the interrupt request. Several ISR bits can be active at any one time, all but one indicating that its lower-priority interrupt handler has been interrupted by a higher-priority interrupt.

So far, so good?

The CPU blips the **INTA** line once more to tell the 8259 to put the Interrupt ID on the data bus. This bus activity is *not* a standard I/O operation, because the **-IOR** and **-IOW** lines remain inactivate. The **INTA** and **INTR** control lines do not appear

on the ISA bus, which means that you cannot put an 8259 on an I/O board and produce more vectored interrupts for your own use, no matter how delightful that prospect may seem or how much it would simplify your embedded life.

The one-byte Interrupt ID seen by the CPU has two fields: five high-order bits from the 8259's Interrupt Vector Byte register and three low-order bits identifying the current ISR bit. The BIOS loads a different value into each 8259's IVB register during the power-up sequence: the master gets 0x08 and the slave gets 0x70.

Once the CPU reads the Interrupt ID, it proceeds as I described earlier: pushes registers, turns off **IF**, converts the ID to a RAM address, fetches the corresponding interrupt vector, and starts the interrupt handler. These operations occur in lockstep sequence, without any further interruption.

Meanwhile, the 8259 can accept a new interrupt on any **IRQ** without an active **IRR** bit. If an **IRQ** goes active, the 8259 compares its priority to the highest ISR bit and recognizes only higher priority **IRQs** by turning on the **INT** output. As before, the CPU responds to or ignores its **INTR** input depending on the Interrupt Flag's state.

The interrupt handler routine must write an **EOI** (End Of Interrupt) command to the 8259 to clear the ISR bit and re-enable any lower-priority interrupts. The BIOS sets the 8259 up so that what's called a **Nonspecific EOI** will reset the highest-priority ISR bit. You can also issue a **Specific EOI** to reset a different bit, but this isn't usually desirable.

The **Nonspecific EOI** command is 0x20. Yes, folks, that's identical to both the master 8259's base I/O address *and* the first address of its interrupt vectors in RAM. You can see why named constants are such a good idea... you cannot tell what the number 0x20 is, represents, or will do, just by staring at it.

Interrupts on the slave 8259 work slightly differently. As you can see from Schematic 1, the slave's **INT** output connects to the master's **IRQ 2** input and the two chips have different **SP/-EN** input settings. The slave **IRQ** activates its **INT** output after resolving its own **IRQ** priorities. That signal triggers the master's **IRQ 2** and forces the usual priority resolution. When the CPU responds to the master's **INT**, however, the *slave* provides the Interrupt ID.

In this case, two ISR bits become active: one in the slave 8259's ISR that identifies the actual interrupt and ISR bit 2 in the master 8259 that indicates an active slave 8259. The interrupt handler (forgive me for not calling it an ISR) for each slave **IRQ** must send an **EOI** to each 8259 to clear both ISR bits.

Collision Alarm!

The default setting for the master 8259's Interrupt Vector Byte register may be the second-worst idea in the whole PC kingdom. I had long believed it represented a classic case of what happens when you ignore what's printed in the data books, but the actual story has a surprising twist.

As you saw earlier, events inside the CPU generate exception interrupts. Intel now reserves Interrupt IDs 00h through 1Fh for those exceptions, but the 8086 and 8088 did not use all 32 IDs (nor do the 80486, the Pentium, and the Pentium Pro, for that matter). The PC BIOS code used several of those reserved interrupts for its own software functions and hardware interrupts. Inevitably, when subsequent Intel CPUs triggered exceptions using interrupts already used by the BIOS, the two functions collided with a loud crash.

In late 1996, Dave Bradley presented a history of the IBM PC at an IEEE session in Research Triangle Park, NC. When he mentioned that he wrote the Original PC BIOS, I asked him why he used those particular BIOS software interrupt numbers. He just rolled his eyes. Obviously, he'd gone through this before.

He said it was a simple case of Hobson's Choice. Microsoft had already laid claim to all the interrupt vectors upward from 20h for their own software. In late 1980, Intel had not yet reserved *any* of the unused interrupts below 20h; that came later, when Intel codified their 80186/80188 designs. So, Dave simply put the BIOS functions and hardware interrupts in the only vacant spots shown by his data books.

Then, in 1982, Intel published those 80186/80188 design manuals and laid claim to those interrupts. Allowing for the usual publication leadtimes, they had no way of knowing that the IBM PC was about to become the tail wagging their dog.

As usual, there's enough blame to go around. However, the moral of this story remains the same: don't tread on reserved ground! If you *know* of a restriction, you *must* assume somebody will eventually use it... even if your design lacks the staying power of the IBM PC architecture.

In any event, Figure 2 summarizes the conflicts between the CPU, BIOS and hardware interrupt IDs. Many of the CPU exceptions occur only in protected mode, where the real-mode BIOS becomes irrelevant. BIOS and DOS hardcode the remainder in a thick layer of PC Compatibility Barnacles.

You can avoid the conflicts at INT 08h through INT 0Fh by writing a different Interrupt Vector Byte into the master 8259. Although only protected-mode operating systems require this, we embedded systems types may find it a handy

Figure 2

Intel's documentation now reserves Interrupt IDs between 00h and 1Fh for CPU exceptions, but the IBM PC BIOS and interrupt hardware got there first. This table summarizes the collisions. CPU exceptions marked with an asterisk occur only in protected mode, making real-mode BIOS conflicts nearly irrelevant. You can change the 8259's Interrupt IDs by writing a new value into the Interrupt Vector Byte register.

Int ID	Intel CPU Exception	BIOS Function	PC Hardware IRQ
00	Divide-By-Zero Error		
01	Step/Debug		
02	NMI pin		
03	Breakpoint		
04	INTO (Overflow)		
05	Bound Check	Print Screen	
06	Invalid Opcode		
07	x87 Not Available		
08	Double Exception		IRQ 0 RTC
09	287 Segment Overrun		IRQ 1 Keyboard
0A	Invalid TSS *		IRQ 2 / IRQ 9 Video
0B	Segment not present *		IRQ 3 COM2
0C	Stack fault		IRQ 4 COM1
0D	General protection		IRQ 5 LPT2
0E	Page fault *		IRQ 6 Diskette
0F	Reserved		IRQ 7 LPT1
10	x87 Error (-ERR pin)	Video functions	
11	Alignment Check (486+) *	System info functions	
12	Machine Check (Pentium+)	Get memory size	
13	Reserved	Disk I/O functions	
14	Reserved	Serial I/O functions	
15	Reserved	System functions	
16	Reserved	Keyboard functions	
17	Reserved	Printer functions	
18	Reserved	Cassette BASIC (!)	
19	Reserved	Bootstrap loader	
1A	Reserved	Time Functions	
1B	Reserved	Ctrl-Break handler	
1C	Reserved	System timer tick	
1D	Reserved	Video parameter tables	
1E	Reserved	Diskette param tables	
1F	Reserved	Pointer to 8x8 graphic font	

trick. The standard alternate seems to be `INT 50h`; if you're going to be nonstandard, you may as well pick the standard method.

I'll leave those machinations as exercises for you, as we already have entirely too many details in this chapter. You should certainly peruse the data books before trying anything fancy. But... you knew that already, didn't you?

While you're doing that, notice that the slave 8259 **IRQ** inputs, **IRQ 8** through **IRQ 15**, use the same digits as the master 8259's Interrupt IDs. Many references list them both in decimal, adding to the confusion: it sure looks like **IRQ 15** (decimal) ought to match up with `Int 15` (decimal), doesn't it? At least now you have a fighting chance of seeing why **IRQ 15** isn't related to `Int 0fh`...

Three Timers Ticking

Having stunned the subject, let's back up and run it over. The `TimeTest.C` demo program produces three external interrupts at known rates using the Firmware Development Board's 82C54 timer, then reports on the handler response times. As before, I wrote the code in Micro-C. You can load it on your target system using either the diskette boot routine or `MON86`'s serial `HEX` transfer command.

The program puts all three 82C54 timers into Mode 2, which produces a short blip when the counter reaches zero. I picked a 5 ms period for the timers to simplify scope sync: 200 traces per second make it easy on the eyes and allow plenty of time for the handlers to finish up before the next interrupt.

Timers 0, 1, and 2 drive **IRQ 5**, **IRQ 10**, and **IRQ 15**, respectively. The master 8259 drives **IRQ 5**. Both **IRQ 10** and **IRQ 15** have higher priorities, because they come through **IRQ 2** by way of the slave 8259.

The first order of business is writing an interrupt handler. Although, for reasons that will soon become evident, I favor assembly language over C for this task, Listing 1 shows a perfectly serviceable Timer 0 handler written in Micro-C. It resembles a standard C function, except for the `INTERRUPT` macro appearing before the function name.

You should remember that C functions and their inline assembly code may change nearly anything, including the CPU registers, before returning to their caller. This is obviously inappropriate behavior for an interrupt handler that must return control to the interrupted program with all registers intact. Although the Micro-C compiler saves and restores `BP`, we must handle the remaining registers by ourselves.

Listing 1

Interrupt handlers are generally written in assembly language, but you can get away with C in some cases. This handler turns on a parallel printer port bit, records the current timer value in an array, sends an EOI to the 8259, and turns off the parallel port bit. The `INTERRUPT` macro on the first line is the key to using a standard Micro-C function as an interrupt handler. See Listing 2 for the macro definition.

```
/*----- */
/* Timer 0 hardware interrupt handler */
/* Assumes interrupt on master controller */
INTERRUPT(Timer0Handler) HandlerT0() {
    outp(SYNC_ADDR,inp(SYNC_ADDR) | 0x01);
    if (!(Response[0][RESP_SET]++)) {
        Response[0][RESP_NOW] = ReadTimer(0,I8254_BASE);/* fetch times */
    }
    outp(I8259A,NS_EOI);
    outp(SYNC_ADDR,inp(SYNC_ADDR) & ~0x01);
    return;
}
```

Dave Dunfield suggested the `INTERRUPT` macro shown in Listing 2. It `PUSHes` all the CPU registers and the `temp` variable used by the Micro-C compiler and runtime routines, then `CALLs` the C function, which can be as rude as it chooses. When the function returns, the `INTERRUPT` macro `POP`s the saved values off the stack and performs the obligatory `IRET` instruction after restoring all the registers.

Different compilers turn standard C functions into interrupt handlers using different methods, but they all boil down to essentially the same thing: save and restore the CPU state around the C code. Check your compiler's manual for the details, then write a few testcases to verify that you understand what's going on.

It should be obvious that hardware interrupt handlers cannot have any parameters, nor may they return results in the CPU's registers. By definition, all the CPU registers and the stack will be in an unknown state (as far as the C code knows) when the interrupt occurs, so the handler must restore them to the same condition as it exits. Those restrictions leave no place for parameters nor return values!

With that in mind, `HandlerT0` in Listing 1 turns on bit 0 in the parallel printer port, reads Timer 0 and records the value in a global array, sends the all-important EOI to the 8259, and shuts the printer port bit off. Triggering your scope on

Listing 2

A C function invoked by a hardware interrupt must not change anything used by the main-line C code. This Micro-C macro saves all the CPU registers and the compiler's ?temp scratch variable, calls the interrupt handler, then restores everything before returning.

```
#define INT_PROLOGUE 30 /* size of prologue code */
#define INT_ENTRY(fn) (&fn-INT_PROLOGUE) /* start of prologue */

#define _SPACE_
#define INTERRUPT(fn) asm { \
fn PUSH AX \
  PUSH BX \
  PUSH CX \
  PUSH DX \
  PUSH SI \
  PUSH DI \
  PUSH ES \
  PUSH DS \
  PUSH CS \
  POP DS \
  MOV AX,?temp \
  PUSH AX \
  CALL fn+INT_PROLOGUE \
  POP AX \
  MOV _SPACE_?temp,AX \
  POP DS \
  POP ES \
  POP DI \
  POP SI \
  POP DX \
  POP CX \
  POP BX \
  POP AX \
  IRET \
}

#undef _SPACE_
```

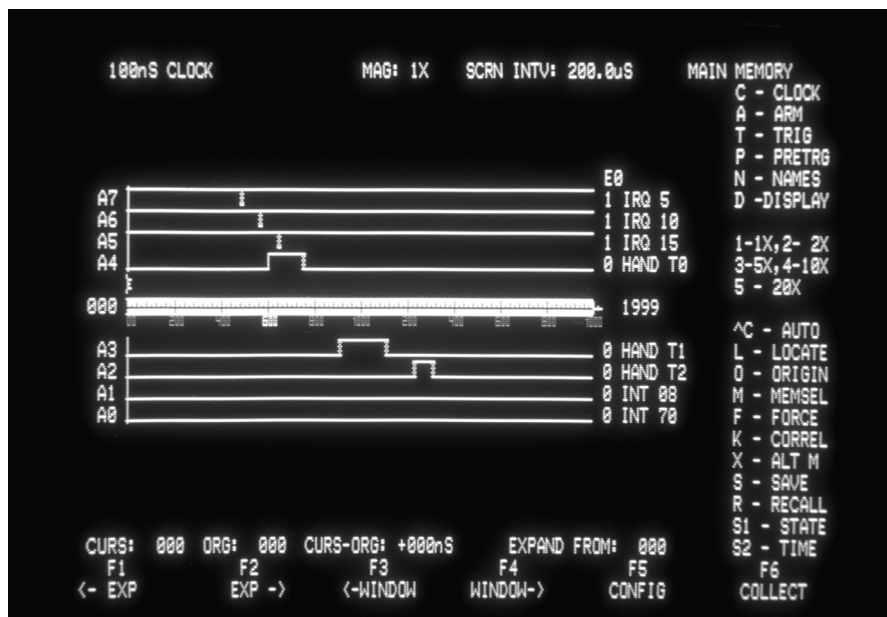
IRQ 5 and observing bit 0 on the printer port should give you a good indication of how much time your CPU takes to respond to an interrupt.

The **IRQ 10** and **IRQ 15** handlers are similar, with a few wrinkles I'll discuss in a moment. Photo 1 shows the three **IRQ** pulses triggering their interrupt handlers. Notice that the **IRQ 5** handler runs to completion, even though the other two handlers have higher priorities (remember: **IRQ 10** and **IRQ 15** trigger **IRQ 2** on the master 8259, which outranks the master's **IRQ 5**). **HandlerT0** runs with interrupts off, because the CPU disables other external interrupts when it accepts the first one: those other handlers don't stand a chance.

Listing 3 shows the **IRQ 10** handler. Compiling with **NEST_1** defined causes the code to send EOIs that enable interrupts immediately after setting the parallel port bit. Photo 2 shows the result: **HandlerT2** now interrupts **HandlerT1**. Even

Photo 1

The IRQ signals shown in the top three traces trigger three interrupt handlers that report their activity through the target system's parallel printer port. This logic analyzer record shows that HandlerT0 will run to completion, even with higher priority interrupts pending, because the interrupt handler code does not set the CPU's Interrupt Flag bit.



though **IRQ 15** has a lower priority than **IRQ 10**, the 8259 thinks that the IRQ 10 handler has finished when it receives the corresponding EOI.

One implication of this situation: a second **IRQ 10** hardware interrupt would start another copy of **HandlerT1**! As far as the 8259 is concerned, **HandlerT1** is history when it sent an EOI command to shut off its In-Service Register bit. In our case, the handler has finished long before the next **IRQ 10** pulse, but it's something to bear in mind if you have fast interrupts and slow handlers.

Photo 3 shows a somewhat more complex event: **HandlerT1** can be interrupted by more than one other interrupt handler. In this case, the BIOS timer tick on **INT 70h (IRQ 8)** and **HandlerT2** get into the act. The timer tick has a higher priority than **IRQ 15**, so even though **IRQ 15** remains pending, the **IRQ 8** handler runs first.

Listing 3

If NEST_1 is defined, this interrupt handler will send an EOI command to the 8259 interrupt controllers and turn on the CPU's Interrupt Flag bit. That allows other interrupt handlers to gain control in the middle of this code. If the 82C54 timer were to produce another IRQ before this code returns, the CPU would push its registers again and start executing the handler from the top, which is generally not what you expect.

```

/*-----*/
/* Timer 1 hardware interrupt handler */
/* Assumes interrupt on secondary controller */
INTERRUPT(Timer1Handler) HandlerT1() {
    outp(SYNC_ADDR,inp(SYNC_ADDR) | 0x02);

#ifdef NEST_1
    outp(I8259B,NS_EOI);          /* tell slave we are done */
    outp(I8259A,NS_EOI);          /* tell master we are done */
    enable();                     /* and allow other interrupts */
#endif

    if (!(Response[1][RESP_SET]++)) {
        Response[1][RESP_NOW] = ReadTimer(1,I8254_BASE);/* fetch times */
    }

#ifdef NEST_1
    outp(I8259B,NS_EOI);          /* tell slave we are done */
    outp(I8259A,NS_EOI);          /* tell master we are done */
#endif

    outp(SYNC_ADDR,inp(SYNC_ADDR) & ~0x02);

    return;
}

```

The source code includes the routines I wrapped around the BIOS interrupt handlers to activate parallel port bits for those pictures. This trick comes in handy, even for application programmers. Well, application programmers who care about their code's realtime performance... firmware folks, just like you and me.

Digital Readout

Even if you don't have a scope or logic analyzer, you can still experiment with interrupt handlers using the code from this chapter. Each handler reads back the current value of its 82C54 timer channel and stores it in a global array. Once per second, the mainline code calculates the minimum, running average, and maximum values of those times for each channel, then sends the results to the host system through the serial port, where it appears as shown in Figure 3.

Photo 2

A different version of HandlerT1 sends EOIs to the 8259s and sets the CPU IF bit as soon as it gains control. This record shows HandlerT2 running while HandlerT1 is suspended, even though HandlerT2 has a lower priority. As soon as the 8259 receives an EOI, it resets the highest priority ISR bit and generates a new INT output based on whatever IRR bits remain active, which can result in precisely the situation shown here.

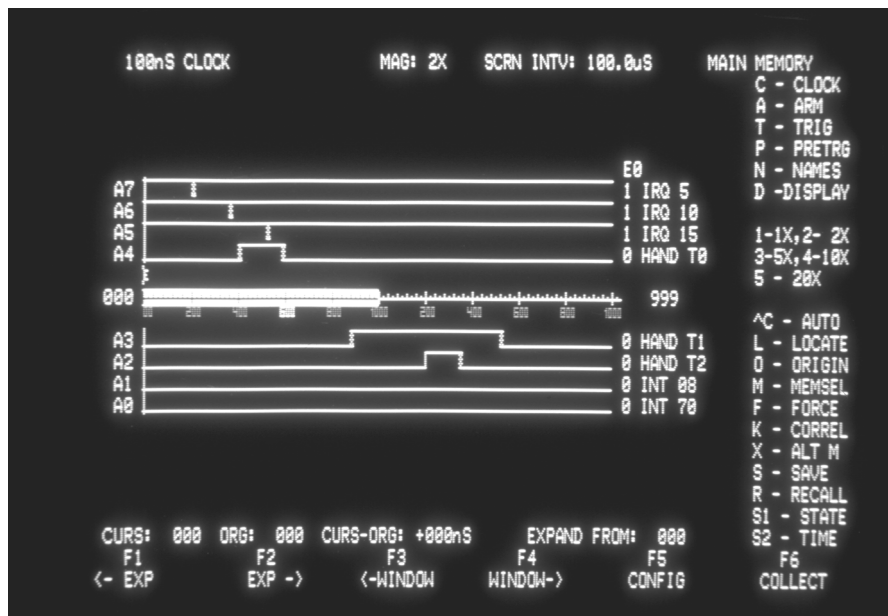


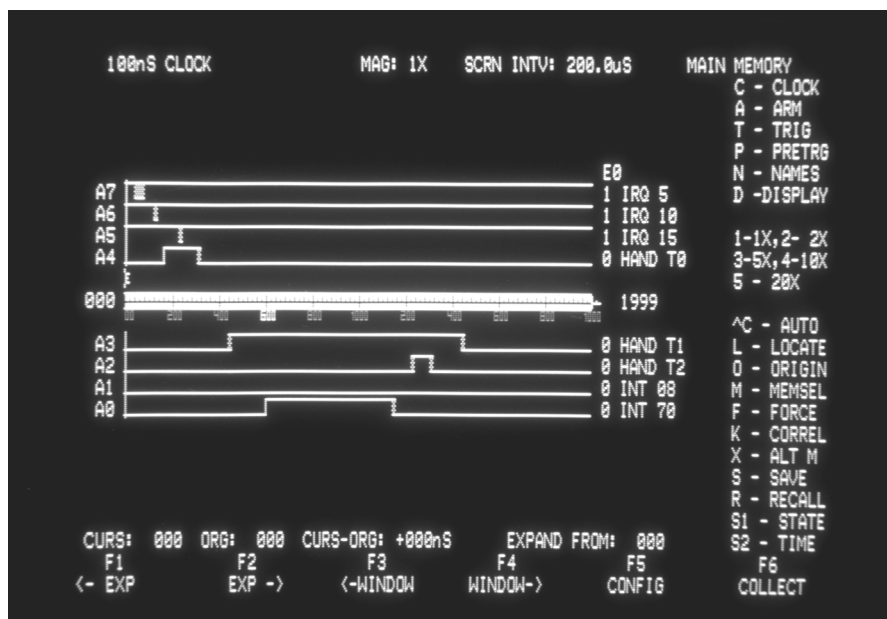
Figure 3

The interrupt handlers record the elapsed time between the IRQ signal and the 82C54 command required to latch the timer. The main program summarizes those values, showing you how the different interrupt handlers respond. Displaying this response time table on your host PC requires a terminal emulator capable of handling ANSI cursor control strings, which is usually a matter of selecting the right option from a dialog box.

```
Interrupt Handler Timing Exerciser
Embedded PC's ISA Bus Chapter 5 -- Ed Nisley
Timers are not synchronized
<<< some text omitted in this listing >>>
Interrupt response times in 139 ns ticks:
Timer Current Minimum Average Maximum
0          102      96       97      545
1          114     108      108     597
2           40      35       35     524
Max stack used: 00C8
```

Photo 3

This photo shows both the Real-Time Clock and HandlerT2 gaining control during HandlerT1's execution. The RTC interrupt on INT 70 runs first because it has a higher priority than HandlerT2 and does not re-enable interrupts until it ends.



Recall that 82C54 timers running in Mode 2 count down to zero, generate an output blip (which we wired to an I/O bus **IRQ** line), then automatically reload their maximum count value and continue to tick. The difference between the maximum and current timer values equals the number of ticks since the reload, precisely the elapsed time since the **IRQ** line went active.

Thus, the minimum and maximum values shown in Figure 3 give you an estimate of how fast your handler can possibly respond to an interrupt and how long it may take under less-than-ideal conditions. These figures change as the program continues to run and the handlers interrupt each other in different patterns.

You can easily see that HandlerT2, triggered by **IRQ 15**, runs considerably faster than HandlerT0 on **IRQ 5**. Listing 4 shows the reason: assembly language code living inside a standard Micro-C function.

Listing 4

This interrupt handler uses assembly language to reduce the overhead caused by saving and restoring the CPU state. The result is a much faster handler that is also much harder to understand, a familiar tradeoff in embedded systems work.

```
/*-----*/
/* Timer 2 hardware interrupt handler */
/* Assumes interrupt on secondary controller */
/* CAUTION -- all the constants are hardcoded in here... */
/* ... and the registers are saved manually */
/* Micro-C inserts PUSH BP and MOV BP,SP before the first line of code */
HandlerT2() {
asm {
        PUSH    AX           save bystanders
        PUSH    DX
*
        MOV     AL,$80       latch Timer 2
        MOV     DX,$030E
        OUT     DX,AL
*
        MOV     DX,$0378     flag startup
        IN      AL,DX
        OR      AL,$04
        OUT     DX,AL
*
        MOV     AX,CS        set up data segment addressing
        MOV     DS,AX
*
        PUSH    SI           save more bystanders
        PUSH    DS
*
        MOV     DX,$030      read the latched LSB
        IN      AL,DX
        JMP     <Punt4
Punt4
        MOV     AH,AL
        IN      AL,DX
        XCHG    AH,AL        ... and the MSB
                           rearrange them
        MOV     DX,AX        save for later
*
        MOV     SI,#Response+(2*7*2)    aim at our slice of Response
*
        MOV     AX,4[SI]     fetch _SET element
        AND     AX,AX        previous entry processed?
        JNZ     Done         nonzero says skip this one
*
        MOV     2[SI],DX     stash it away
        INC     ASWORD(4[SI]) account for this sample
*
Done    MOV     AL,$20       send the EOIs
        OUT     $20,AL       ... to master
        OUT     $A0,AL       ... and slave
*
        MOV     DX,$0378     flag shutdown
        IN      AL,DX
        AND     AL,$FB
        OUT     DX,AL
*
}
```

Listing continued on next page

Listing continued from previous page

```
        POP        DS            restore bystanders
        POP        SI
        POP        DX
        POP        AX
        POP        BP            saved at function entry
        IRET                preempt normal return
    }
}
```

Unlike the other two handlers, this one need not save and restore all the CPU registers, because it knows precisely which ones it will use. Therefore, it gets started faster, runs faster, and exits faster. In round numbers, it runs in a third of the competition's time... while becoming at least three times harder to write and understand. You may choose which end of that tradeoff you prefer.

The main line code monitors the total stack used by the routines, which changes while the program runs. This will be particularly noticeable with nested interrupts, as the varying combinations use an unpredictable amount of stack space.

Unlike my blown cerebral stack, it's essentially impossible to blow the stack in this program, because Micro-C's setup code puts the stack at the far end of the 64 KB code and data segment. It's worthwhile to check the program's stack requirements occasionally, but 64 KB marks a big change (and welcome relief) from an 8031 microcontroller's cramped quarters. For that matter, how much of a program could you write with only 128 bytes of RAM?

Although I used a few other tricks in the source code, this overview should get you started. I heartily recommend spending some time with interrupt handlers to discover what your system can do... and what it *can't* do, no matter how good your firmware skills may be.

Speaking of skills, here's a scary story for you...

A Cautionary Tale

Before you start writing firmware, you *must* read the hardware data sheets carefully. Generally, I do a lot of reading before starting a project, but once in a while, well... This tale shows that, verily, hell hath no fury like that of an unjustified assumption.

Once Upon A Time, back in 1988, I wrote MC-Net, a control program that connected 8052-class microcontrollers and PCs into a networked system. You could write data collection firmware on the microcontrollers, store the results on

the PC's disks, monitor and operate the microcontrollers from the PC, and so on and so forth.

To summarize the details, MC-Net used an RS-485 serial link running at 19.2 kb/s between (up to) 32 nodes. A **Monitor** program, running on a PC AT, served as a remote console for all the 8052-BASIC nodes and provide an overall network debugging display. I described the code in my *Firmware Furnace* column in *Circuit Cellar INK* magazine Issues 10 through 12. All in all, a neat project.

Because the network's design point specified an 8 MHz IBM AT with an 8250 UART, handling a 1920 bytes/second network posed some interesting challenges. A stock AT runs at about 1 MIPS and can execute about 500 instructions between each incoming byte. If you get distracted for a millisecond or so while doing something else, you will certainly lose data, because the 8250 has no FIFO buffers.

Each transmitted byte generates two interrupts, because the RS-485 network echoes outgoing data back to the receiver. The first interrupt occurs when the transmitter buffer goes empty and the second, very shortly thereafter, blinks on when the receiver buffer fills with same character. The elapsed time varies, but it can be as little as one stop bit time, about 50 μ s at 19.2 kb/s.

Because the two interrupts occur so close together, I checked for pending interrupts at the end of the handler to eliminate the lengthy interrupt exit and entry overhead when a byte was ready. My scope showed that this worked quite well: most of the time each character produced only a single interrupt as the handler absorbed the second interrupt request.

About a year later, though, some customers reported sporadic problems with network errors that we simply couldn't duplicate. Some of the problems occurred due to cabling errors, some to termination problems, others came from severe noise... but there remained a very small minority of customers with everything set up right and everything going wrong.

The problems seemed more severe on faster machines. Finally, one customer installed MC-Net on his new 66 MHz 486DX2 and reported that it failed in a matter of minutes. *Ah-ha!* The bug *must* be related to CPU speed, because, in this case, we'd carefully eliminated everything else.

I set up a test network on my then-new 33 MHz '386SX, activated the trace outputs built into all my code, hitched up the logic analyzer, and waited to see what happened. After a long wait, the outputs reported that the TSR had jammed in an absolutely *impossible* state.

Although my system wasn't as fast as the latest '486 CPU, the logic analyzer showed that each outbound character generally produced two interrupts. The CPU now ran fast enough that the interrupt handler exited before the second interrupt occurred. Progress had eliminated the need for my interrupt polling trick, although the code was still in place for slower CPUs. However, very, *very* rarely, the second interrupt, the one caused by the receiver buffer, produced a suspiciously long blip.

I modified the TSR code to produce unique trace outputs for each possible interrupt source and discovered that the abnormally long interrupts were caused by a change in the Modem Status Register (MSR). That was certainly peculiar, as the TSR code never enabled MSR interrupts... and the Interrupt ID Register (IIR) should *never* report a disabled interrupt.

Essentially all PCs use National 8250, 16450, or 16550 serial interface UARTs, or a fragment of LSI that works just like them. I pored over the data sheets in search of something I'd missed three years earlier. What could possibly cause an invalid IIR? I assumed that my code was at fault, as genuine hardware problems remain very, very few and very, very far between.

In the 8250 family, you clear the transmitter interrupt flag when you read the IIR or write a new character, but the only way to clear the receiver interrupt is reading the pending byte. You would expect, as I did, that the chip updates the Interrupt ID Register almost immediately. You would be nearly right.

The IIR reports the highest-priority pending interrupt as a number in Bits 1 and 2. Bit 0 presents a summary status bit that, when zero, means "there is at least one interrupt active." My code read the IIR and used it as an index into a decoding table. Only two other interrupts can occur in the MC-Net TSR, but, being a belt-and-suspenders type, my table had all possible entries. That saved my skin!

Upon closer scrutiny, the 16450 data sheet revealed two key timings. The hardware updates the summary bit within 250 ns of reading the IIR for transmitter interrupts. Should the *receiver* cause the interrupt, however, the hardware may take up to 1 μ s to reset the summary bit after you read the character. The interrupt request output pin has the same timings, so the IIR bit must be wired to the output driver rather than the actual logic gate on the chip.

As Sherlock puts it, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

A sufficiently fast CPU can respond to the interrupt, read the IIR, branch to the receiver handler, read and process the byte, and check the IIR again *before* the

interrupt summary bit changes. Because the receiver interrupt bit clears almost immediately, the IIR contains invalid data.

As you might guess, an all-zero IIR indicates a modem status interrupt.

I checked my references again to see if anyone else knew about this. The only hint appeared in Mark Nelson's *Serial Communications: A C++ Developer's Guide*, published in 1992 (well after I needed it). In the *8250 Oddities* section, he states:

One annoying bug found in both the original National Semiconductor chips as well as some clone chips is the false modem status interrupt. The IIR can report a modem status interrupt when none has occurred. This could easily lead to trouble with the ISR code.

Indeed!

The serial port in your system may be just one corner of an LSI chip, but the PC Compatibility Barnacles dictate exactly how it must work. In this case, the Barnacles required my new silicon to precisely duplicate the same old bug.

There being no good fixes for this situation, I used a time-honored kludge: a delay loop. The code measures the CPU speed when it installs the TSR and sets up a delay loop that occupies at least a microsecond. After each receiver interrupt, the code stalls for long enough to ensure that the IIR interrupt summary flag becomes valid before testing it again. Not pretty, but it worked OK... and so did the TSR.

This story has a twofold moral: RTFM (Read The *Fine* Manual) first, then build trace outputs into your code to show you what's going on in actual real time.

But... you knew that already, didn't you?

Right?

Release Notes

I've recoded several of the support routines in assembler and moved them into the `FirmDev.ASM` library file. Use **SLIB** to update `Tiny.LIB` on your system.

The program files for this chapter include several batch files that create the executable files. You'll surely need to tweak them for your system, but that's all part of the learning experience. Check the `ReadMe.txt` file in this chapter's subdirectory for more details.

6 Memories Are Made of This

Although the PC BIOS has all the code required to boot a program from disk or diskette, some applications cry out for just a smidge of nonrotating, nonvolatile storage. Small embedded programs can run entirely without moving parts (assuming you have a power supply without a fan!) and larger programs can store configuration, identification, or logging data without disk I/O.

Compared to the confines of an 8031 microcontroller, the nearly 1 MB of address space available in a PC (ignoring protected mode for now) seems almost limitless. As you'll discover in this chapter, though, there isn't that much space left for our embedded applications: a contiguous 64 KB block may be hard to come by.

I'll start by reviewing the PC's memory layout, explore ISA bus memory timing, then describe the circuitry that adds an EPROM or EEPROM to the Firmware Development Board. With the hardware in place, a little firmware can load a program and make it a part of the BIOS that runs whenever the PC starts up.

Where Does Memory Come From?

Every PC's memory organization pays homage to The Original IBM PC and its 8088 CPU. At this late date, we can only quibble about the details, because a thick crust of PC Compatibility Barnacles renders the Big Picture impervious to change. Figure 1 shows the major divisions in the first megabyte of storage.

The (in)famous 640 KB block devoted to user programs and data forms the first and largest chunk. If your real-mode application requires contiguous RAM, this is as good as it gets. While there are ways to extend this block, none are particularly attractive or generally reliable in all systems.

The video RAM buffers occupy the next 128 KB, starting at A0000. The old CGA board freed the space below B8000 and allowed a glorious 736 KB of contiguous user RAM, but the VGA's 128 KB buffer renders that trick essentially useless. Although a VGA in CGA mode can release the space below B8000, simply changing back to VGA mode will lock up the system as the hardware buffer collides with the system RAM.

Of course, if your application doesn't use video at all, you can yank the board and devote its entire address space to whatever you'd like. The Bad News: you can then never, ever, install a video board along with your hardware. That seems a shame, given the utility of built-in, standard video with BIOS support, but it's your call...

Figure 1

The first megabyte of PC memory serves many different functions, defined both by the BIOS and by convention. You should think long and hard about compatibility problems before you devote a chunk of address space to a nonstandard use! In this chapter, we'll build the hardware that puts an EPROM or EEPROM at C8000, then write the code that turns it into a BIOS extension.

Address Range	Size	Type	Function
00000 - 9FFFF	640 K	RAM	Programs & data
A0000 - BFFFF	128 K	Video RAM Buffer	Video buffers
C0000 - C7FFF	32 K	Video ROM	BIOS Extension
C8000 - CFFFF	32 K	ROM or RAM	BIOS Extension
D0000 - DFFFF	64 K	ROM or RAM	BIOS Extension
E0000 - EFFFF	64 K	ROM	BIOS or BIOS Extension
F0000 - FFFFF	64 K	ROM	System Board BIOS

The system board BIOS may occupy either 64 KB starting at F0000 or 128 KB starting at E0000. Early PCs sported an empty EPROM socket or two on the system board, unused hardware that relentless cost reduction eliminated in short order. If your PC doesn't have BIOS code at E0000, which is quite unlikely in this day and age, you can use that address space for your own purposes.

Some I/O boards, notably video adapters, SCSI hard disk controllers, and network adapters, include EPROMs that modify, extend, or completely replace some system board BIOS functions. For example, plugging in any display adapter more complex than a CGA or MDA (aren't they all?) also installs a BIOS extension that replaces the BIOS video routines.

To accommodate those new functions, the BIOS scans the 128 KB of address space between C0000 and DFFFF to find those EPROMs and execute their startup code during power-on initialization. Fortunately for us, this entire process follows well defined steps, without much magic at all, allowing us to add our own BIOS extensions for our custom hardware.

I/O boards may include ordinary RAM in addition to the EPROM, but the cramped address space restricts that RAM to small buffers and scratchpads. The history of Lotus-Intel-Microsoft Expanded Memory Specification boards

(remember LIM EMS?) shows what can be done when you're desperate for more RAM. Bank-switching 32 MB of storage through a 64 KB peephole wasn't pretty, but LIM provided the only standard mechanism to do it when it needed doing.

Embedded applications enjoy far more freedom to chop up the lower 640 KB than standard PC apps running under DOS, so when we need big buffers, they need not be crammed between the video buffers and the system BIOS. Some applications can also make use of the vast extent of RAM beyond the lower megabyte, even when running in real mode, and we'll consider that in due time.

The Firmware Development Board can accommodate either 8 KB or 32 KB of solid state storage, mapped into the PC's address space between C8000 and CFFFF. While the decoding circuitry allows you to plunk it at any other address in the first megabyte, you can now see just how constricted your choices are.

Before we link up with the BIOS, though, we must get that memory running...

Confronting the ISA Slows

Not only do the PC Compatibility Barnacles determine the memory layout, they also set the minimum memory access time. You've probably noticed that current PC boards favor memory and I/O on the CPU's local bus and PCI boards rather than the ISA bus. After reading this chapter, you'll know why.

Schematic 1 shows the falling-off-a-log simple circuitry that puts an 27C256-style EPROM on the Firmware Development Board. The LS245 buffers isolate the ISA bus data and control lines, the F521 activates the EPROM's **-CE** input when the CPU reads or writes a byte in the desired address range, and the EPROM, of course, holds the data.

The tradeoff for this simplicity comes, as usual, at the expense of performance. Because I used a byte-wide EPROM, the CPU can fetch only one byte at a time. The ISA bus defaults to the same achingly slow, six-cycle, 720 ns access for memory as it does for I/O, with the results shown in Photo 1.

Figure 2A shows ISA bus timings for an 8-bit memory access and Figure 2B shows typical access times for a 200 ns EPROM or EEPROM. It's easy to see that the EPROM has its data ready *long* before the end of the bus cycle. With that in mind, I won't go through the same analysis as with I/O ports in Chapter 4.

It may appear obvious that the best way to improve performance is by reducing the ISA bus memory access cycle time, but sometimes appearances can be deceiving.

Schematic 1

Adding an EPROM to the Firmware Development Board requires only a few additional bus buffers and an address decoder. A 744LS688 may be easier to find than a 74F521, but be careful of its longer signal delay.

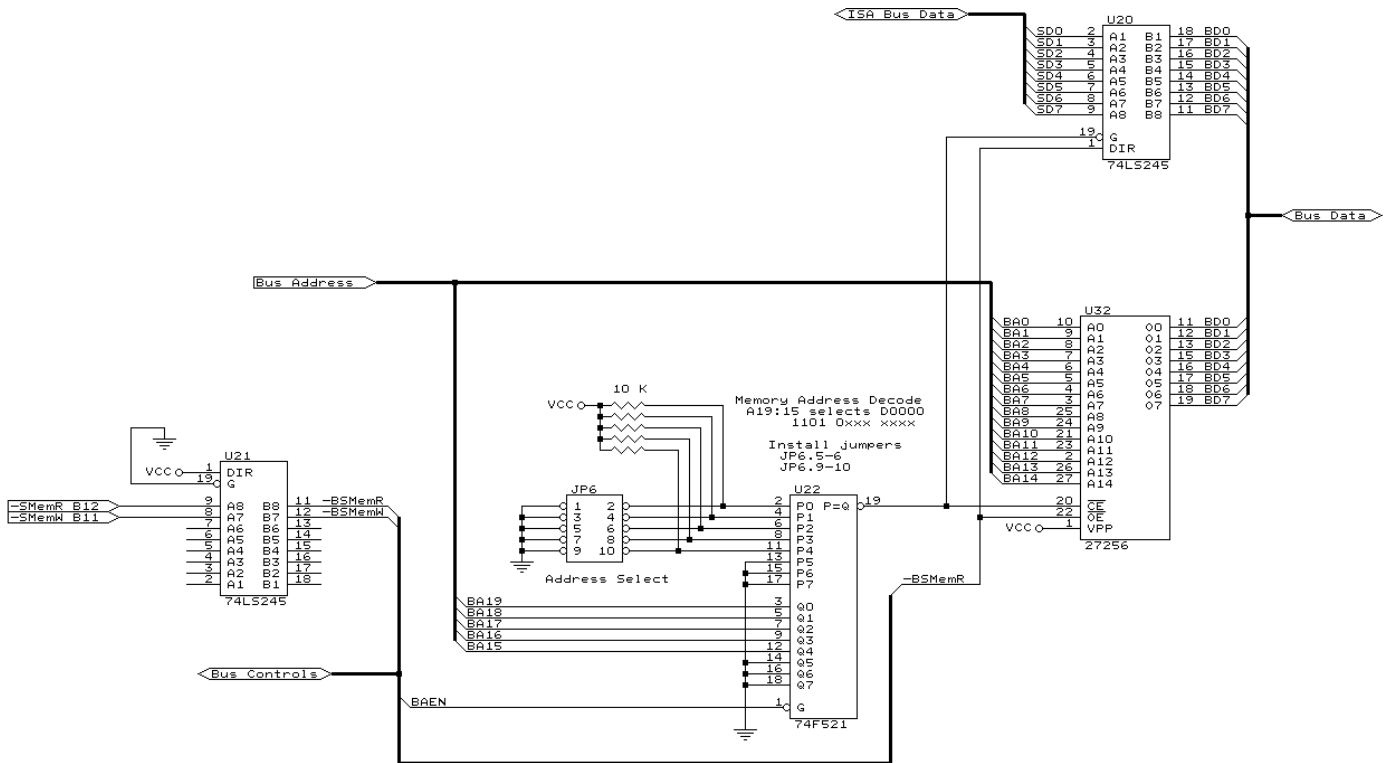
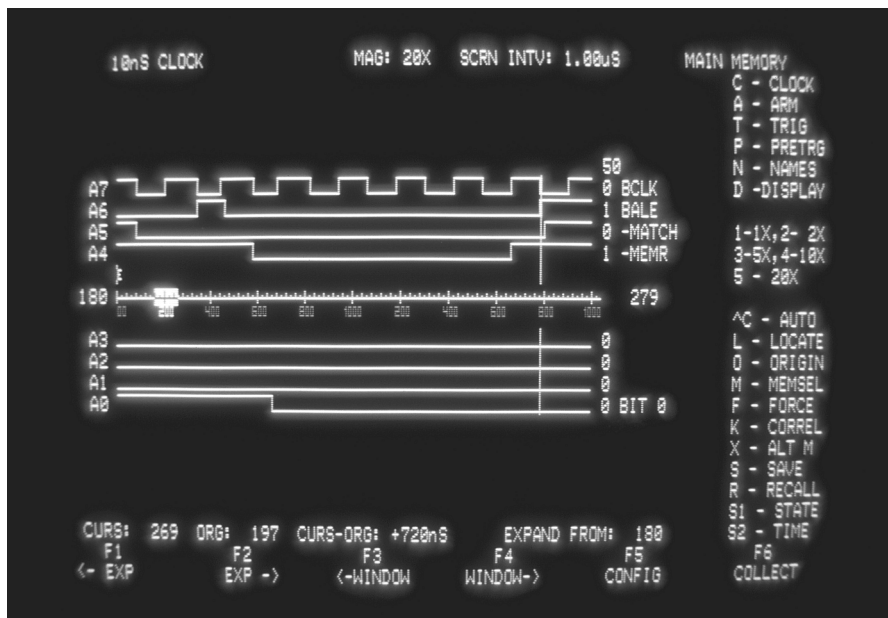


Photo 1

This logic analyzer trace shows what happens when the CPU reads a single EPROM byte. The access begins with the rising edge of BALE and ends six BCLK cycles later when -SMemR (mislabelled -MEMR here) goes high.



We have two ways to speed up ISA bus memory accesses. A pair of byte-wide EPROMs (or a 16-bit part) that provide 16-bit accesses, plus circuitry to activate **-MEMCS16**, yields three-cycle memory accesses. If that isn't good enough, additional circuitry can activate **-SRDY** to cut one cycle out of the access, although Solari's books are replete with cautions and compatibility hazards about doing that.

It turns out, though, that the PC provides an even better way that makes the bus access time irrelevant without any extra hardware. The *ROM shadowing* feature available on nearly all current system boards copies the EPROM contents into RAM, disables the EPROM, maps the RAM to the EPROM's address range, and write-protects it. Poof: fast EPROM made possible by cheap RAM!

Contrary to popular opinion, this has nothing whatsoever to do with the *x86* CPU's protected mode memory management hardware. It's entirely a function of the system board LSI hardware, meaning that your programs continue to run in

Figure 2a

This diagram shows the signals and timing involved in an 8-bit ISA bus memory read access. The timings remain compatible with boards designed for the Original IBM PC and seem painfully slow by contemporary standards.

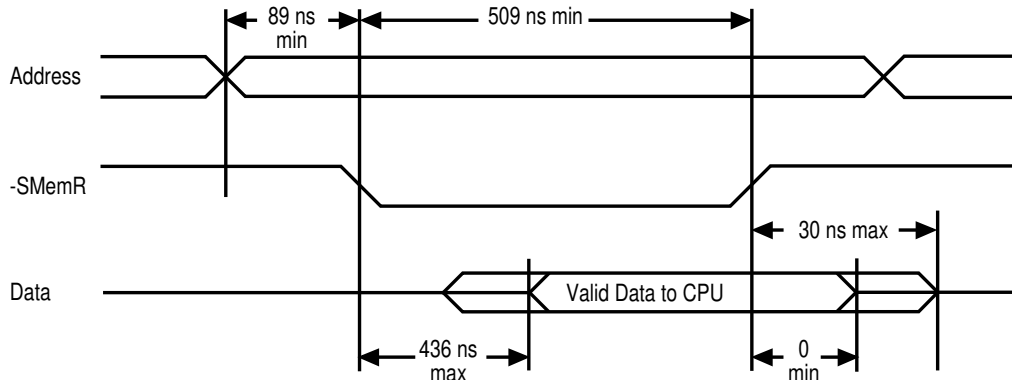
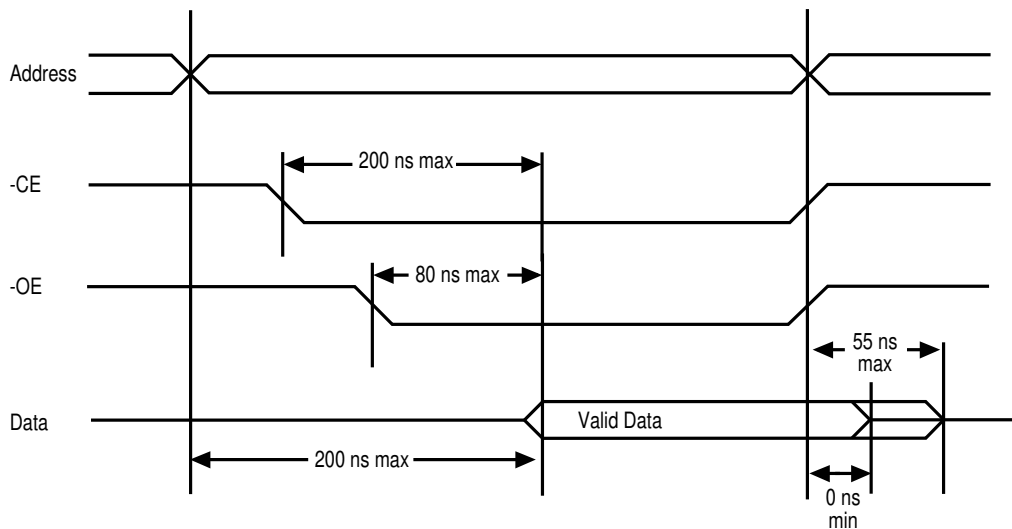


Figure 2b

These timings show the read cycle for a 200 ns 28C64A EEPROM and are typical of EPROMs as well. The bus buffers shown in the schematic truncate the rather long maximum data hold time.



real mode. The relocation hardware may chop up the memory above the first (or 16th or 64th) megabyte enough that protected mode operating systems have trouble using it, but that's a separate design issue.

The BIOS performs all the copying and remapping during the power-on reset sequence, so by the time your code gets control, the EPROM has fallen out of the picture. The system board circuitry runs much faster than the ISA bus, giving every operation that depends on EPROM code or data a significant boost.

How significant?

Listing 1 shows a section of `MemTest.C` that reads the 32 KB block of storage with a single `REP LODSB` instruction. This takes 33 ms per loop with shadowing disabled, about 960 ns per `LODSB` step. Enabling ROM shadowing cuts the loop to 7.3 ms, or only 210 ns per step. Simple division shows that ROM shadowing reduces the elapsed loop time by about 80%.

Even if you could somehow get a “no wait state” ISA bus interface running, each bus access would still take 240 ns. To judge from my logic analyzer traces, the CPU adds two data-access bus cycles to the minimum required for each `REP LODSB` instruction, so even an optimized interface would take 480 ns per byte.

There you have it: a four chip, warp speed, no hassle EPROM storage system for your embedded system. Ain't science grand?

Listing 1

This test loop uses a single `REP LODSB` instruction to read the entire 32 KB EPROM address space. The elapsed loop time, measured either by stopwatch or oscilloscope, gives a good indication of how fast the ISA bus can handle memory accesses.

```
RAMSize = 0x8000;                /* repeat for 32K block          */
RAMSeg = NV_SEGMENT;
Counter = 0;
while (!chkch()) {
    outp(SYNC_ADDR, 0x01);        /* scope sync                  */
    asm {
        MOV    CX, RAMSize       set up count
        PUSH   DS
        MOV    AX, RAMSeg        set up address
        MOV    DS, AX
        XOR    SI, SI
        REP    LODSB
        POP    DS
    }
    outp(SYNC_ADDR, 0x00);
    outpw(LED_ADDR, ~ByteToSegs(Counter)); /* show count on FDB LEDs */
    ++Counter;
}
```

The `MemTest.C` program has the test routines I used to get the memory circuitry working, including a `HEX` file with 32 KB of pseudorandom numbers from Micro-C's `rand()` function. Burn `PSR32K.HEX` into a 27C256 EPROM, plug it in, then run `MemTest` to read and verify it... that should give you confidence in your wiring. It's not an absolute test, but I doubt you'll find a bug that can hide from it.

Writing to EEPROM

There are times, however, when an EPROM just isn't the right hammer for the job. Whether you have frequent code changes, use nonvolatile storage for data logging, or just don't want to hassle with an EPROM programmer, an EEPROM may solve your storage problem.

EEPROMs come in several different flavors, but for our purposes they're all pretty much alike. I'll use the Microchip 28C64A 8 KB EEPROM as an example because it's readily available from the usual mail order sources. Feel free to use something else, but remember the address space limits *before* you spring for a megabyte part.

Schematic 2 shows the changes that put a 28C64A in place of the 27C256 EPROM. Because the EEPROM has only 8 KB, the CPU will see four identical copies in the 32 KB address range decoded by the F521 comparator. You can either add **SA14** and **SA13** to the comparator for full decoding or just ignore the ghosts, as I did in the code for this chapter.

As shown in Figure 2b, reading an EEPROM goes just like reading an EPROM: the data appears at the CPU in plenty of time. Once again, we need no special tricks to extract data from the part.

Writing, on the other hand, becomes considerably more complex. The 28C64A EEPROM requires up to 1 ms (that's 1000 microseconds, one million nanoseconds, about 63 miles on Admiral Hopper's scale) to erase and reprogram each byte. The 28C64A, as with all useful EEPROMs, has internal latch and timing circuitry to relieve the CPU of the details, but it cannot accept a new byte until the previous write cycle is completed.

Figure 3a shows the ISA bus write timings and Figure 3b shows the 28C64A's requirements. Obviously, we cannot jam the ISA bus for an entire millisecond, because (among other things) that would disable RAM refreshing and lose some data somewhere. Remember that normal RAM refresh cycles occur every 15.6 μ s and the maximum wait-state delay shouldn't exceed 1 μ s.

If your code has other things to do, simply ignore the EEPROM for at least a millisecond after each write. The timers on the Firmware Development Board can

Schematic 2

Installing a 28C64A EEPROM requires a few more parts in addition to those shown for the 27C256 EPROM in Schematic 1. That same 74F521 comparator produces -Mem Addr Match and these Bus Data lines use those buffers. We will replace the jumper on the LS00 gate with a firmware controlled write enable signal in Chapter 7.

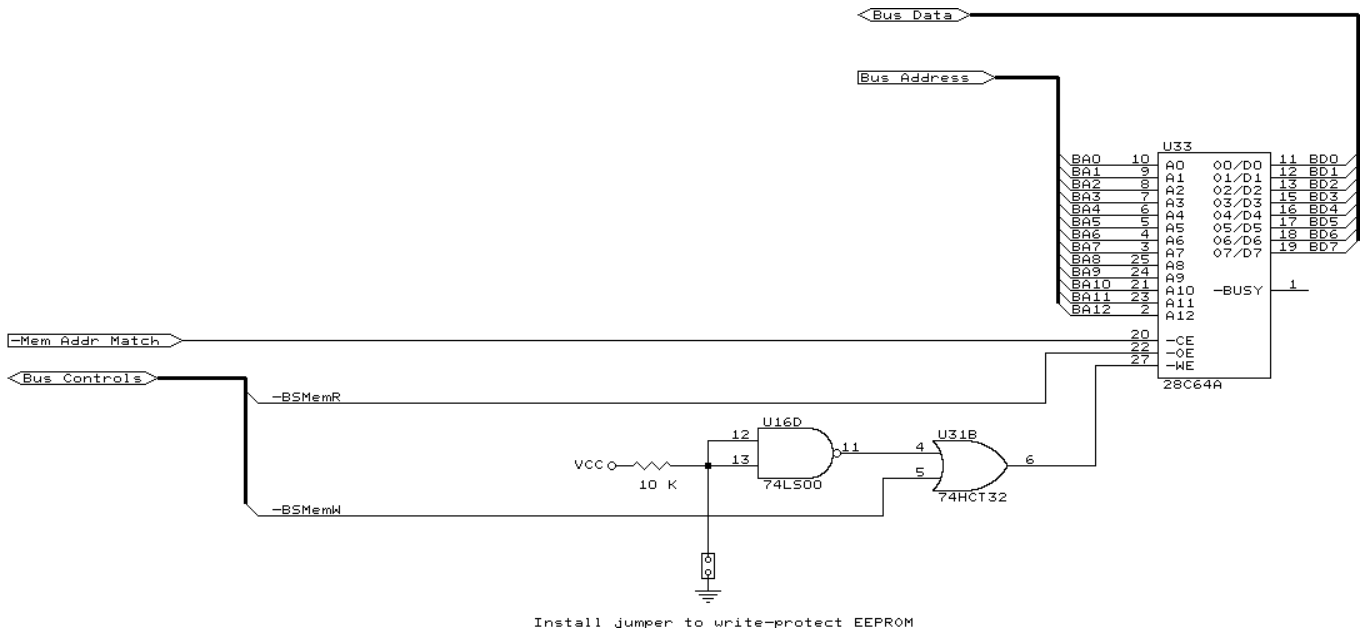


Figure 3a

ISA bus memory write accesses are similar to reads. Note that the data may not be valid for quite a while after the leading edge of -SMemW , so the destination must rely on the trailing edge for precise timing.

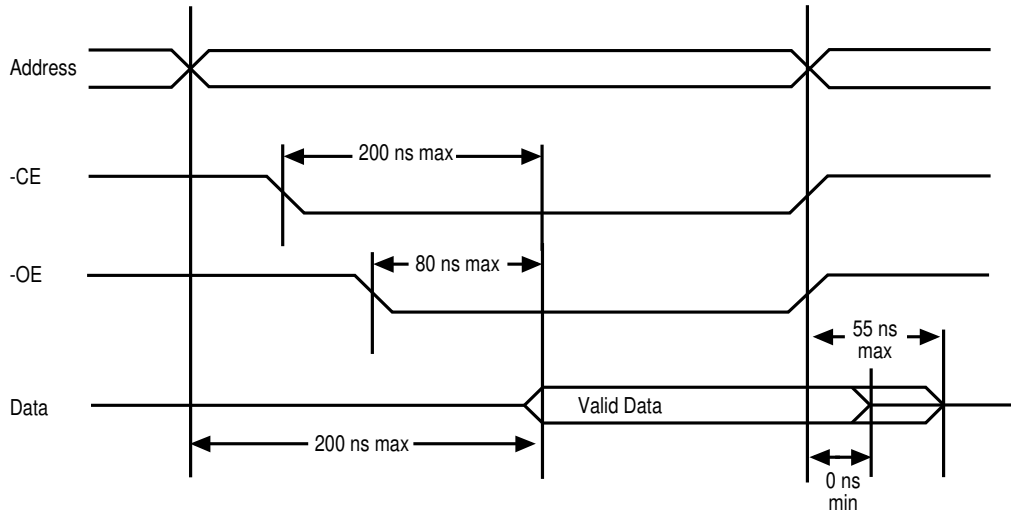
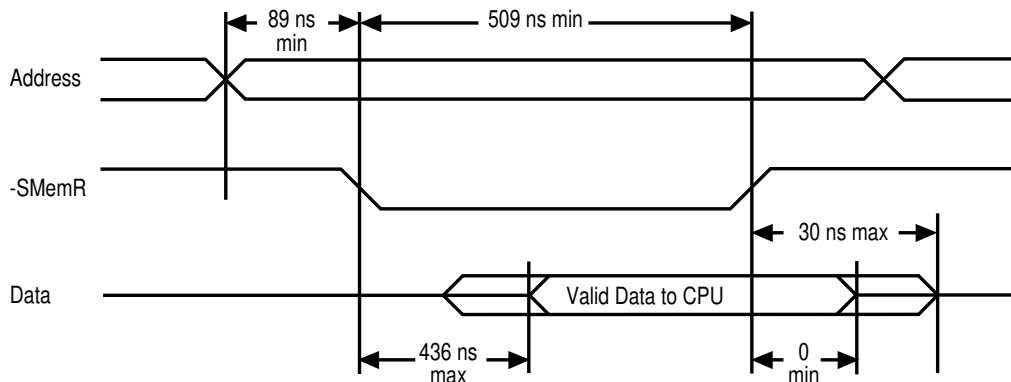


Figure 3b

The write cycle for a 28C64A EEPROM seems essentially identical to a standard RAM, but the EEPROM cannot accept more data for about a millisecond after beginning a write. This timing diagram shows the signals for the first part of the process; the text in this chapter describes how the firmware detects the end of the cycle.



easily meter out this kind of delay with a little additional code. The disadvantages should be obvious: more hardware and a write cycle running as slowly as the slowest possible chip. Fortunately, we need not go to such great lengths.

The 28C64A includes a polling mode that signals when the write cycle finishes. Without any additional hardware, you can write to the chip as fast as it can process

Listing 2

Writing a byte into the 28C64A is easy, but you must then poll the chip until the write cycle completes, which can take up to one millisecond. The code must also include error handling for timeouts and defective parts. Note that this will work perfectly with a standard static RAM in place of the 28C64A, which is a good way to debug the code without using up the EEPROM's maximum number of write cycles... a million writes at full throttle don't take very long at all, even on the ISA bus!

```
for (RAMAddr = 0x0000; RAMAddr != 0x2000; ++RAMAddr) {
    PRSData = peek(PRSsegment, RAMAddr);
    outp(SYNC_ADDR, 0x01);                               /* mark the write */

    poke(NV_SEGMENT, RAMAddr, PRSData);                  /* start it */
    if (SetAlarm(0x0000, 0x2710, &Alarm)) {              /* error after 10 ms */
        putstr("Cannot set BIOS timeout!\n");
        break;
    }

    do {
        TestData = peek(NV_SEGMENT, RAMAddr);
    } while ((0x0080 & (PRSData ^ TestData)) && !Alarm);

    if (Alarm) {
        printf("Timeout programming %04x to %02x, is %02x\n",
            RAMAddr, PRSData, TestData);
        if (MAX_ERRORS < ++ErrorCounter) {
            putstr("Too many errors, giving up!\n");
            break;
        }
    }

    TestData = peek(NV_SEGMENT, RAMAddr);
    if (TestData != PRSData) {
        printf("Cannot program %04x to %02x, is %02x\n",
            RAMAddr, PRSData, TestData);
        if (MAX_ERRORS < ++ErrorCounter) {
            putstr("Too many errors, giving up!\n");
            break;
        }
    }

    if (CancelAlarm(0x0000, 0x2710, &Alarm)) {
        putstr("Cannot cancel BIOS timeout!\n");
        break;
    }

    outp(SYNC_ADDR, 0x00);
    outpw(LED_ADDR, ~ByteToSegs(HI(RAMAddr)));
}
```

the new data. After each write, just read the data back from the same address and compare bit 7 with the original data. The chip inverts bit 7 while it's busy writing and returns valid data when it's done. Bits 0-6 are undefined, so you should mask them off before comparing the bytes.

Typical writes can run *much* faster than the 1 ms maximum spec, meaning that you can save a significant amount of EEPROM programming time by monitoring the chip's status. MemTest's programming loop writes all 8 KB in about four seconds for my EEPROMs, making the average write cycle well under 500 μ s. That includes the overhead around each write: the chip itself runs even faster.

Listing 2 shows how this works. After writing the byte using Micro-C's `poke()` function, the code sets up a 10 ms time-out and enters a loop waiting for bit 7 to match the `poke()`ed bit. After the XOR returns zero (or after the time-out), the code checks for errors before continuing with the next byte.

Note that you must perform one additional read *after* the 28C64A reports that it's done, because the data may not be valid on the same read that shows bit 7 changing. You can get around this by comparing the entire byte each time, without masking the high bit, during the polling loop. Just keep trying until your code makes a final comparison that verifies everything worked correctly.

The 28C64A EEPROM uses much the same pinout as an 8 KB static RAM. I *strongly* recommend that you work with a RAM instead of an EEPROM until you are *entirely* sure your hardware and code come up to par. The 28C64A specs say it will endure 10^4 write cycles, which works out to perhaps ten seconds at full throttle.

For example, one of MemTest's routines runs in a tight write loop that displays the control signals on a scope. This will slaughter an innocent 28C64A in short order. Be careful what's in the socket before you boot that code!

Pin 26 provides a second chip enable line (**+CE2**) on 8 KB RAMs. On 28C64A EEPROMs, that pin has no connection and you can simply wire it high. Conversely, pin 1 reports the 28C64A's **-Busy** status and has no connection on 8 KB RAMs. Similar reasoning applies to EEPROMs larger than the 28C256, but please match up the data sheets before trying anything silly.

Being able to program an EEPROM in the system can be an advantage, but that convenience makes it distressingly easy to clobber your precious code or data with an errant write. If you plan to use an in-circuit-programmable EEPROM, the jumper and gates shown in Schematic 2 provide simple, manual write protection.

Even if (you think) your code can't possibly make a mistake, other routines may scribble all over *your* address space. For example, the BIOS power-on code in one of my systems *writes* AA every 2 KB or so throughout the entire address range: an unprotected EEPROM in that box won't survive a single boot.

Incidentally, the same advice holds true for the system BIOS. My Pentium Pro system stores its BIOS in flash ROM and allows updates from diskette without opening the box. While setting up Windows 95 (why not NT? It's a long story, trust me), the initial scan for I/O devices wiped out the flash ROM.

After that happened several times under different conditions, I built a small interposer socket that holds the **V_{PP}** programming supply pin firmly low. No more problems, although I must now open the box and remove the chip to update the BIOS. A fair tradeoff, I'd say... but I already own a device programmer.

Installing the write protect jumper forces the EEPROM's **-WE** line high and prevents any attempted writes. **MemTest**'s loop will report a timeout error after 10 ms and any other code will simply conclude that the address holds an unchanging EPROM, which is precisely what we want.

In the next chapter, we'll add a software controlled write protect bit for the battery backed RAM, but a manual jumper suffices for now. That circuitry requires an HCT32 gate instead of an LS32, for reasons I'll go into when we get there.

Figure 4

The BIOS examines the start of each 2 KB block between C0000 and E0000 for BIOS extension code. If the first two bytes contain a valid signature and the block checksum equals zero, the BIOS executes a FAR CALL to offset 0003 to execute the extension's initialization code. The BIOS requires only the signature, length byte, and an instruction at offset 0003; the structure of the code block itself is not specified. The diskette EEPROM loader described in this chapter puts the checksum at offset 0005.

Offset	Contents	Definition
0000	55	First signature byte
0001	AA	Second signature byte
0002	xx	Overall length in 512 byte units
0003 - 0004	EB 01	JMP SHORT \$+3 (to 0006)
0005	ss	Checksum
0006...	code	BIOS extension code

The BIOS Connection

Now that the Firmware Development Board has that smidge of nonvolatile memory, we can bolt code onto the target system's BIOS to run after each hardware reset. This opens the door to diskless systems that boot with no mechanical motion.

Actually, given the limited space available (what can you do in 32 KB these days?), it's more likely that the (E)EPROM will hold essential hardware interface routines or configuration values, rather than a complete embedded application. The rest of the code can reside on a diskette with the write protect tab missing.

In any event, we'll start small: once again, the end result will be a few blinking LEDs. The weight of knowledge behind them should make you feel good, though.

As I mentioned earlier, the BIOS scans through memory between C0000 and DFFFF to find the distinctive BIOS extension signature shown in Figure 4. The 55 and AA bytes on a 2 KB boundary mark the start of an extension, which must have a valid checksum over the block of storage specified by the length byte.

Listing 3 shows how the search works. I wrote `ROMScan.C` to examine my system's address space and tell me where to put the Firmware Development Board's nonvolatile memory. It turned out that the only BIOS extension was a 32 KB EPROM on the VGA board at C000:0000, but your system may have other ROMs on other boards.

`ROMScan` cannot identify anything that isn't a ROM holding a BIOS extension. If your system includes LIM EMS boards, network adapters, or other oddities (you aren't doing this on your host PC, are you?) it won't show their RAM buffers or other ROMs. On the other hand, neither can the BIOS, so we're even.

When the BIOS finds a valid extension, it executes a `FAR CALL` to the instruction at offset 0003. You *must* put the first byte of your routine at that address. When your code finishes initializing itself, it should return control to the BIOS with a `RETF` (`FAR RETURN`) instruction. The BIOS then seeks out other extensions and, after calling all of them, continues with the normal disk boot process.

The length byte counts in units of 512 bytes starting from offset 0000, so a length of 0x02 indicates a 1024 byte block. If your routine occupies only 600 bytes, it must still have a length code of 0x02 and the *next* extension must start at offset 0800. An 8 KB EEPROM filled with a single extension requires a length code of 0x10. A zero length code, 0x00, *should* mean an extension that occupies 128 KB, but as we'll see in Chapter 16, at least one BIOS thinks a zero-length extension is perfectly OK, regardless of its checksum value.

Listing 3

The ROMScan program mimics the BIOS signature search through the address space between C0000 and E0000. It displays the header for valid extensions, which should help you find a vacant spot for the Firmware Development Board's nonvolatile memory.

```
BaseSeg = 0xC000;
BaseOffset = 0x0000;

do {
    BlockFlag = peekw(BaseSeg,BaseOffset);
    BlockSize = peek(BaseSeg,BaseOffset+2);
    if ((0xAA55 == BlockFlag) || SHOWALL) {
        printf("%04x %04x %04x %02x ",
            BaseSeg,BaseOffset,BlockFlag,BlockSize);
        if (0xAA55 == BlockFlag) {
            TestAddr = BaseOffset;
            CheckSum = 0;
            for (BlockSize *= 0x0200; BlockSize; --BlockSize) {
                CheckSum += peek(BaseSeg,TestAddr++);
            }
            printf("%02x OK!\n",CheckSum);
        }
        else {
            putchar("--\n");
        }
    }
    BaseOffset += 0x0200;
    if (!BaseOffset) {
        BaseSeg += 0x1000;
    }
} while ((BaseSeg < 0xE000) ||
        ((BaseSeg == 0xE000) && !BaseOffset));
```

The checksum does not include the three signature header bytes and the BIOS does not care where you put the checksum byte. As long as the (length×512) - 3 bytes starting at offset 0003 add up to zero, the BIOS concludes the block holds a valid extension. The code for this chapter puts the checksum in offset 0005, just after a `JMP SHORT` to the rest of the code. You may use any other location, assuming you modify the `LoadExt` diskette boot loader to suit.

Listing 4 shows a simple BIOS extension with two functions. If the low-order DIP switch on the Firmware Development Board is ON, it blinks the LEDs forever to indicate that it has seized control. If it finds that switch OFF, it turns both decimal points on and returns to the BIOS, which continues its normal boot sequence.

Before loading this into the (E)EPROM, we must compute the checksum. I modified the diskette boot loader from Chapter 1 to perform that function on a system with the Firmware Development Board installed. After reading the file from diskette as usual, `LoadExt` writes it into the EEPROM, computes the checksum, and plunks it into offset 0005.

Listing 4

BIOS extensions normally do something useful, but this is just a demonstration. If the low-order DIP switch is ON, it blinks the LEDs forever. If the switch is OFF, it turns both LED decimal points on and returns to the BIOS for normal booting.

```
CODESEG
STARTUPCODE

DB      055h          ; signature
DB      0AAh

DB      1             ; length in units of 512 bytes

MainEntry:
JMP     SHORT Booter  ; force two-byte jump
DB      000h          ; zero checksum until loaded

Booter:
MOV     DX,SW_ADDR    ; should we run?
IN      AX,DX
MOV     DX,LED_ADDR   ; set up for display

TEST    AL,01h        ; low switch ON?
JZ      Onward        ; zero says yes, so stay here

MOV     AX,08080h     ; show we were here
NOT     AX
OUT     DX,AX         ; ... just decimal points!

Onward:
RETF                    ; and return to BIOS!

ReShow:
MOV     AX,0FFFFh     ; all LEDs go off
OUT     DX,AX

Wait1:
MOV     CX,0
LOOP    Wait1

MOV     AX,00000h     ; all LEDs go on
OUT     DX,AX

Wait2:
MOV     CX,0
LOOP    Wait2

JMP     ReShow         ; continue forever
```

After `LoadExt` finishes, install the EEPROM's write protect jumper, pop the diskette out, and hit the Reset button on the PC's front panel to start your new BIOS extension. It's that easy!

One gotcha that became painfully obvious only in retrospect: if your code doesn't fill an exact multiple of 512 bytes, the checksum must include whatever junk lies beyond your code in the last block. You *cannot* compute the sum on just your code,

because the length byte includes more than that. Listing 5 shows the code that figures the EEPROM checksum.

In a blank EPROM, of course, unprogrammed bytes hold 0xFF and you can compute the checksum correctly without actually touching those bytes one by one. Because the HEX file doesn't include the unprogrammed bytes, you must use the extension's length to figure out how many bytes the BIOS will include in its calculations. I suggest that you take the easy way out by padding your extension to a multiple of 512 bytes.

The loader could, of course, automatically fill the rest of the last block with 0xFF bytes before computing the checksum. Choices, choices...

BIOS Extension Hints & Tips

Although I'll cover BIOS extensions in Chapters 8 and 10, a few caveats are in order here, if only to keep you from getting your hopes up.

First of all, I still think it's impractical to write nontrivial BIOS extensions in C. After all, fitting a big pile of stuff into a *very* small bag is a perfect job for assembler code. In Chapter 10, we'll modify the Micro-C startup code to run from the

Listing 5

The BIOS extension checksum excludes the three header bytes, but includes everything else. This routine, taken from the LoadExt boot sector EEPROM loader, computes the checksum based on the extension plus whatever is in the last 512-byte block beyond the end of the code. It writes the result at offset 0005 in the EEPROM, which must be a zero in the disk file.

```
MOV     AX,EEP_SEG      ; aim at EEPROM
MOV     DS,AX
MOV     SI,00002h       ; ... length byte

LODSB                   ; pick up length code
MOV     BX,512           ; convert to bytes
MUL     BX               ; ... in AX
SUB     AX,3             ; adjust for header length
MOV     CX,AX            ; set up for loop

MOV     AH,0             ; set up checksum

MakeCSum:
LODSB                   ; pick up data byte
SUB     AH,AL            ; tick checksum
LOOP    MakeCSum         ; over entire block

MOV     AL,AH            ; set up checksum
MOV     DI,00005h        ; ... address
CALL    WriteEEP         ; ... do it
```


EEPROM, but that requires the `Small/ROM` memory model and some additional code. Dave Dunfield's comments in the `Micro-C` startup code files, along with `Micro-C`'s support for ROM code, will prove quite helpful.

The code in `COM` files produced by PC linkers starts at offset 0100, but BIOS extensions must begin at offset `CS:0000` with code starting at `CS:0003`. Probably the easiest way to relocate the code on the fly is by subtracting 0010 from the `CS` register, then using an indirect branch that increments `IP` by 0100 immediately after the BIOS calls the extension. For example, the byte at `C800:0003` is also located at `C7F0:0103`. That's the same trick I used in the boot sector loaders, so refer back to Chapter 1 to see how it works.

Any RAM used by the EEPROM code should be in the lower 640 KB, but you must ensure that separate BIOS extensions, as well as your own routines, don't step on each other's storage. Remember that the familiar DOS memory allocation routines don't exist during the BIOS boot sequence... and won't exist at all for a target system that doesn't boot DOS.

The BIOS does, however, keep track of the RAM size and you can allocate space by reducing that value. In effect, your data will lie beyond the end of memory because you've moved the `Dead End` sign inward a few kilobytes. We'll cover this in Chapter 10, when we turn a C program into a BIOS extension.

The BIOS invokes all of the extensions after it has performed some, but not all of its initialization and before it attempts to boot from diskette. Your initialization code can hook any interrupts that it will use to regain control later on: timer ticks, serial ports, whatever it takes. As we'll see in Chapter 10, though, some initializations occur after the BIOS calls the last extension and we must take some drastic steps to preserve our data and interrupt vectors.

Want a diskless workstation? Here's a start: the BIOS invokes `INT 19h` after initializing all the extensions. `INT 19h`, of course, handles the normal disk boot load from either floppies or hard disks. If you capture `INT 19h`, your code regains control when the BIOS expects to boot from the disk. That means you can boot from, say, code in the EPROM or a program loaded from the serial port. Now you know one trick involved in making network PCs!

The BIOS will invoke `INT 18h` after `INT 19h` concludes that the system has no bootable diskettes or hard disks. By capturing `INT 18h`, you have the option of booting from diskette to update the firmware (for example), while running without a diskette in normal operation. Just check a pushbutton switch so you don't get stuck with a machine that can't boot without loading your extension. Hint, hint.

According to my references, nearly all non-IBM, PC compatible systems do *not* support INT 18h, as that software interrupt originally fired up good old IBM Cassette BASIC. This probably means that it does something specific on *each* system, but not the same thing on *every* system. Take care, lest you depend on a response that you get only on your development system.

Release Notes

The code for this chapter includes the source and HEX files for everything you've seen here, as well as slight modifications to the FirmDev.H and ASM files. The EEPROM boot loader and ExtDemo BIOS extension require Borland's TASM, with everything else in Micro-C.

Be careful with diskettes containing the LoadExt boot loader and boot them only in your dedicated target system. Unlike the BootSect diskettes we used in Chapter 1, the new version uses the hardware found on the Firmware Development Board. If you boot LoadExt in an ordinary system, it will hang without any error indication. While this won't damage the PC, it can be pretty scary.

Remember, set bit 0 of the FDB's DIP switches ON to run ExtDemo and OFF for a normal boot.

7 Absolute Power Corrupts

Power tends to corrupt and absolute power corrupts absolutely.

While Lord Acton surely didn't consider firmware back in 1887, his epigram certainly applies to our current task. Putting vital data in RAM requires absolute control of both the power supply and the code... because preventing data corruption is what it's all about.

Replacing the EPROM chip we added to the Firmware Development Board with a RAM chip may look easy, but protecting its data turns out to be considerably more difficult. You'll find the power monitoring and watchdog functions helpful, even if your project doesn't require a nonvolatile RAM with battery backup power.

Prepare to RAM

The new hardware we'll build for this chapter has three main sections: a static RAM chip with its support circuitry, a Maxim MAX691 Microprocessor Supervisory Circuit, and a 16-bit I/O port. Refer to the schematics in Chapter 6 and the Schematics appendix for the complete wiring diagrams, because the circuitry we will add here depends on the chips we've already debugged.

Schematic 1 shows the RAM and its support logic, in a layout quite similar to the (E)EPROM hardware we built in Chapter 6. As promised, we now have firmware controlled write protection with an LED indicator.

The Firmware Development Board's memory socket can hold RAM, EPROM, or EEPROM memory in either 8 KB or 32 KB sizes. Although the memory chip pinouts were designed with compatibility in mind, Figure 1 shows the connections that adapt a single socket to the various chips. My board sprouted five jumper blocks that you may not need if you pick just one chip and stick with it.

As you saw in Chapter 6, 8-bit ISA bus accesses allow more than 500 ns from the start of the **-SMemR** or **-SMemW** pulse. I used a Hitachi HM62256LP-150 RAM, which, with its 150 ns access time, beats that spec by a wide margin. The LP suffix indicates a Low Power, standby mode that preserves data, while drawing a very small current from an external backup battery.

Unlike EPROMs and EEPROMs, CMOS static RAM chips require continuous power to maintain their data. Normal operation requires +5 volts, but the RAM cells can retain their contents down to about 2 V. Disabling the chip by raising **-CE** activates the low power mode and reduces the total current drain. For example, a

chip that draws more than 50 mA during a normal access may pull only 10 nA in low power mode.

However, simply disabling the chip may not be enough. The data sheets specify the minimum **-CE** voltage to guarantee a maximum supply current. Because the supply voltage will vary with the battery's condition and temperature, the **-CE** pin voltage spec actually defines the maximum difference between the voltages on pin 28 (**V_{CC}**, the power supply) and pin 20 (**-CE**). A 200 mV maximum differential means that pin 20 must be no more than 0.200 volts below the supply voltage: >4.8 V for normal power and >2.8 V while running from a (nominally) 3 V lithium cell.

Schematic 1

Adding a RAM chip to the Firmware Development Board requires circuitry similar to that presented in Chapter 6. The two CMOS HCT32 gates control the Chip Enable and Write Enable pins to prevent data loss during power failures. A lithium backup battery powers those gates to ensure that the RAM properly enters power down mode.

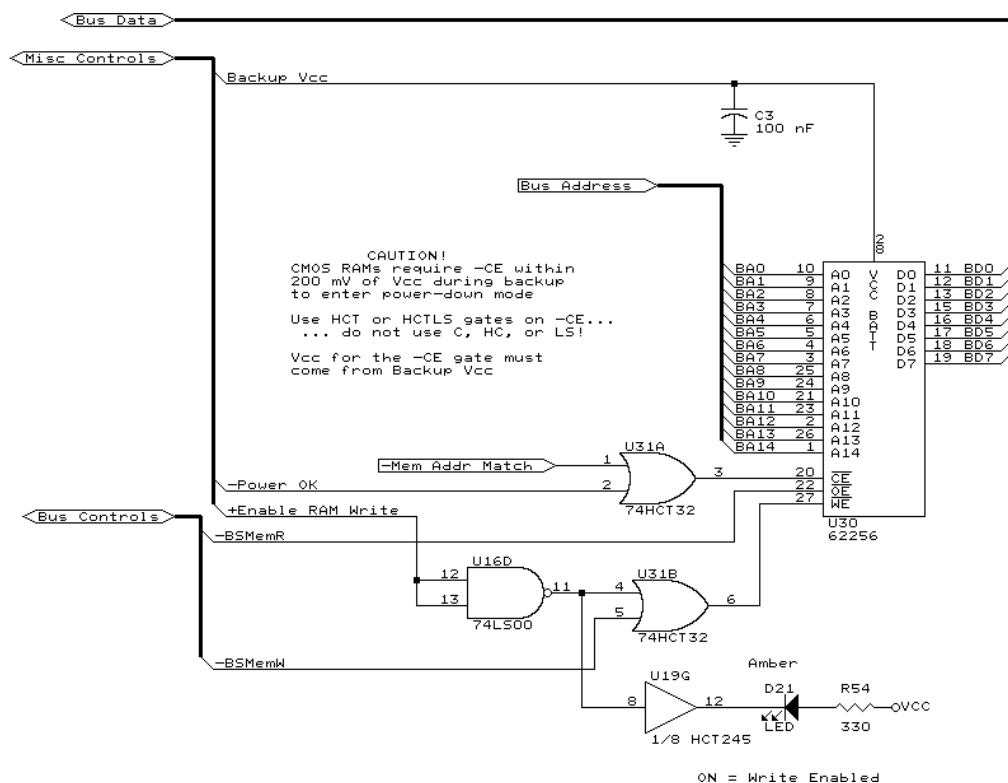


Figure 1

Although 8 KB and 32 KB RAMs, EPROMs, and EEPROMs all come in a 28-pin DIP package, they have some crucial differences. This table gives the connections for the oddball pins. My Firmware Development Board sprouted a cluster of jumpers around the memory socket to cope with all the choices.

Chip Type	Pin 1	Pin 20	Pin 26	Pin 27	Pin 28
8 KB RAM	n/c	Gated -CE	+CE	Gated -WR	Backup V_{cc}
32 KB RAM	A14	Gated -CE	A13	Gated -WR	Backup V_{cc}
8 KB EPROM	V_{pp}	-CE	n/c	-Pgm	V_{cc}
32 KB EPROM	V_{pp}	-CE	A13	A14	V_{cc}
8 KB EEPROM	-Busy	-CE	n/c	Gated -WR	V_{cc}
32 KB EEPROM	A14	-CE	A13	Gated -WR	V_{cc}

Note: External circuitry must *not* drive the 8 KB EEPROM's -Busy output on pin 1.

Figure 2 shows the result of a simple experiment measuring supply current as a function of **-CE** voltage. The vertical axis uses a logarithmic scale to compress the current range, making it easy to see when standby mode kicks in at about 4.5 V. I ran the RAM at +5 V, but the curve has a similar shape at 3 V.

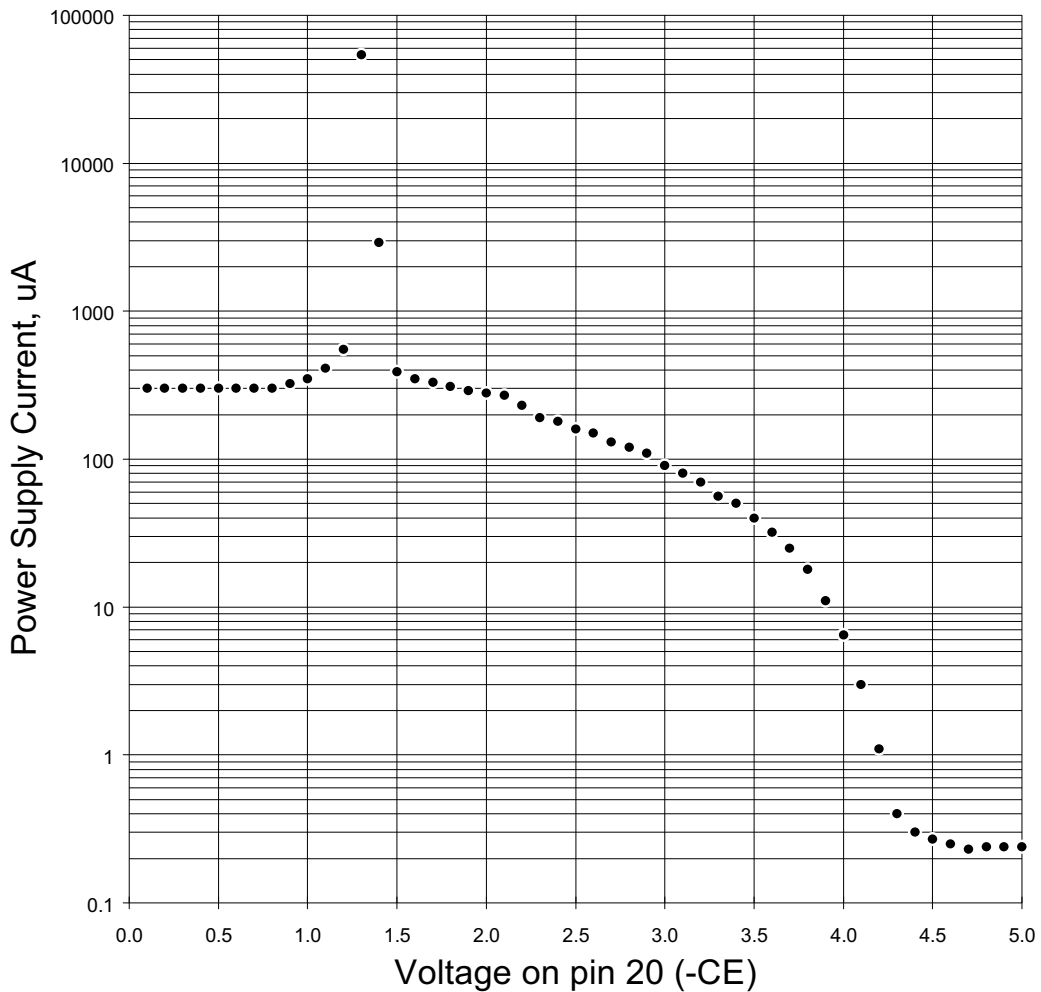
Note that the spike near 1.3 V exceeds 54 mA! It occurs when the **-CE** input teases the chip's internal CMOS logic into the range where both p- and n-channel FETs conduct current. That spike shows why you put lots of bypass capacitors on logic supply lines and where much of the digital noise on your circuit board comes from.

You must drive **-CE** with a CMOS gate to ensure that it reaches the right level. Ordinary TTL gates cannot pull the input high enough, draw too much current for battery operation, and don't run from a 3 volt supply, anyway. The output from a CMOS gate swings nearly to the power supply voltage and tracks that supply as it switches to battery backup.

As with all semiconductors, the RAM's current draw shows an exponential relation with chip temperature and may vary by nearly three orders of magnitude over the full operating temperature range. My graph represents room temperature, of course, and I could double the supply current by parking a desk lamp over the RAM chip. Pay close attention to the spec sheets when sizing the battery for extended temperature operation... those extreme values can be all too real.

Figure 2

This graph shows how the $-\text{CE}$ input voltage affects the current drawn by a static RAM at $V_{\text{CC}} = 5$ volts, but a similar curve applies for 3 V battery backup operation. The $-\text{CE}$ input must be within a few hundred millivolts of V_{CC} to put the RAM into standby mode. The 54 mA spike near 1.3 volts occurs when the chip's internal logic passes through the range where both its p- and n-channel FETs conduct current.



Backup Warning

Although we've all seen and used the canonical diode-and-battery backup power circuit, there are good reasons for more complexity. I picked the venerable MAX691, because it packs power monitoring, battery control, RAM protection, and a watchdog timer in a single IC. Other parts may be better for your particular application, but the MAX691 remains a general purpose workhorse.

Schematic 2 shows the minimal external circuitry we need. Most of the gates drive indicator LEDs that you might not use in a production system. I favor lots of LEDs that indicate firmware and hardware status for what is, after all, a demo system.

Battery backup for the RAM becomes straightforward, as the MAX691 switches the voltage on pin 2 to the higher of the power supply at pin 3 or the battery at pin 1. Although I used a 3 V lithium coin cell rated at 250 mAh, any power source that provides enough voltage for the RAM will work.

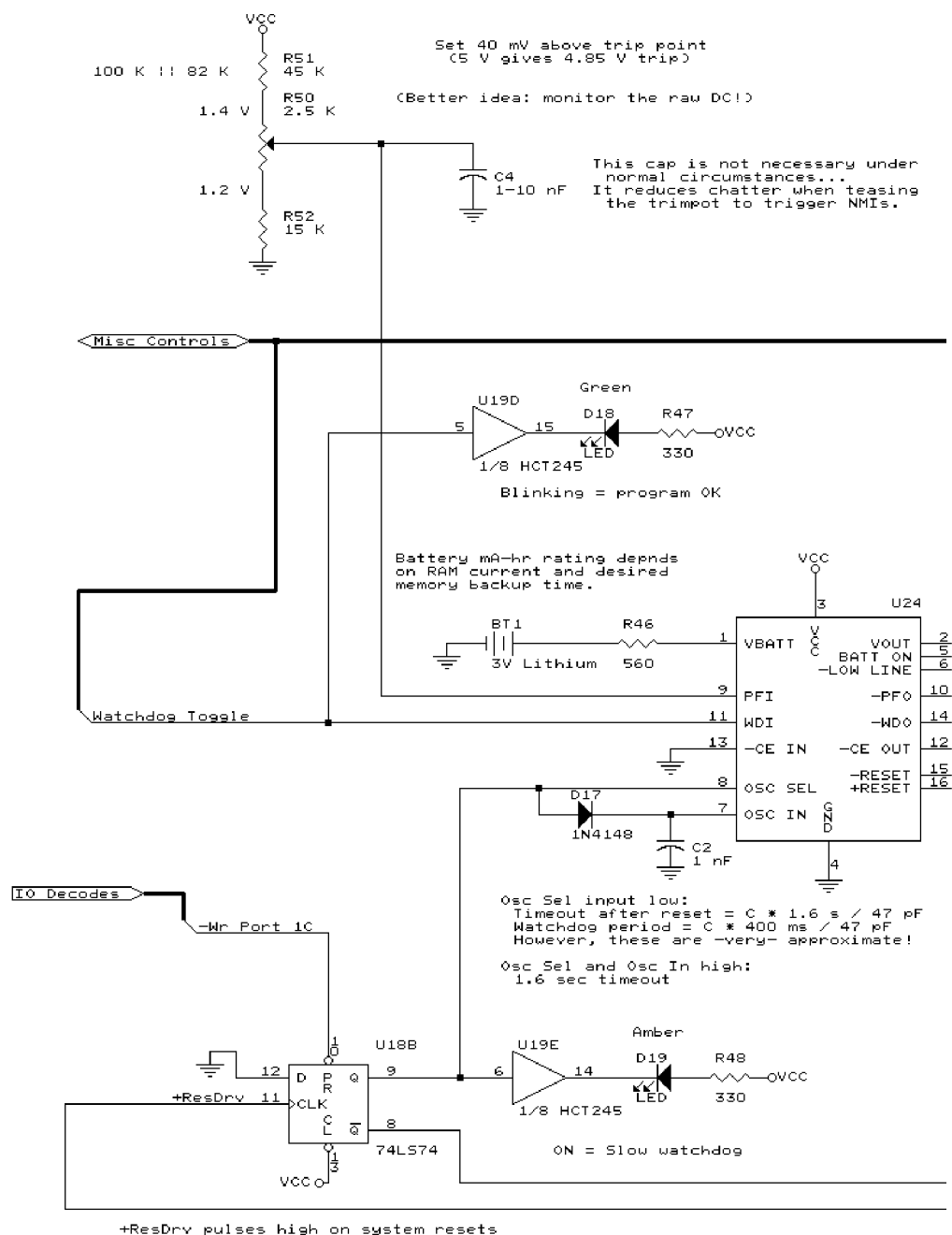
A NEC Static RAM Application Note I reviewed for this project mentioned several UL requirements for lithium cell backup circuits. Even if your product specs don't *require* UL approval, the guidelines make sense. Bear in mind that I have not read the UL regulations themselves, so don't depend on *my* suggestions to get your design approved!

Standard, nonrechargeable lithium cells react explosively to recharging, so you must prevent current from flowing into the cell. Typically you would use a Schottky barrier diode in series with the battery, because the forward drop of an ordinary silicon junction diode can be far too high. The UL requirement limits the charging current to 1% of the cell's capacity, prorated by the possible charging time over the battery's service life. This can be a surprisingly small number that requires careful diode spec checks.

For example, if the PC's power supply will be active 8 hours per day with a cell capacity of 250 mAh, the reverse charging current may not exceed

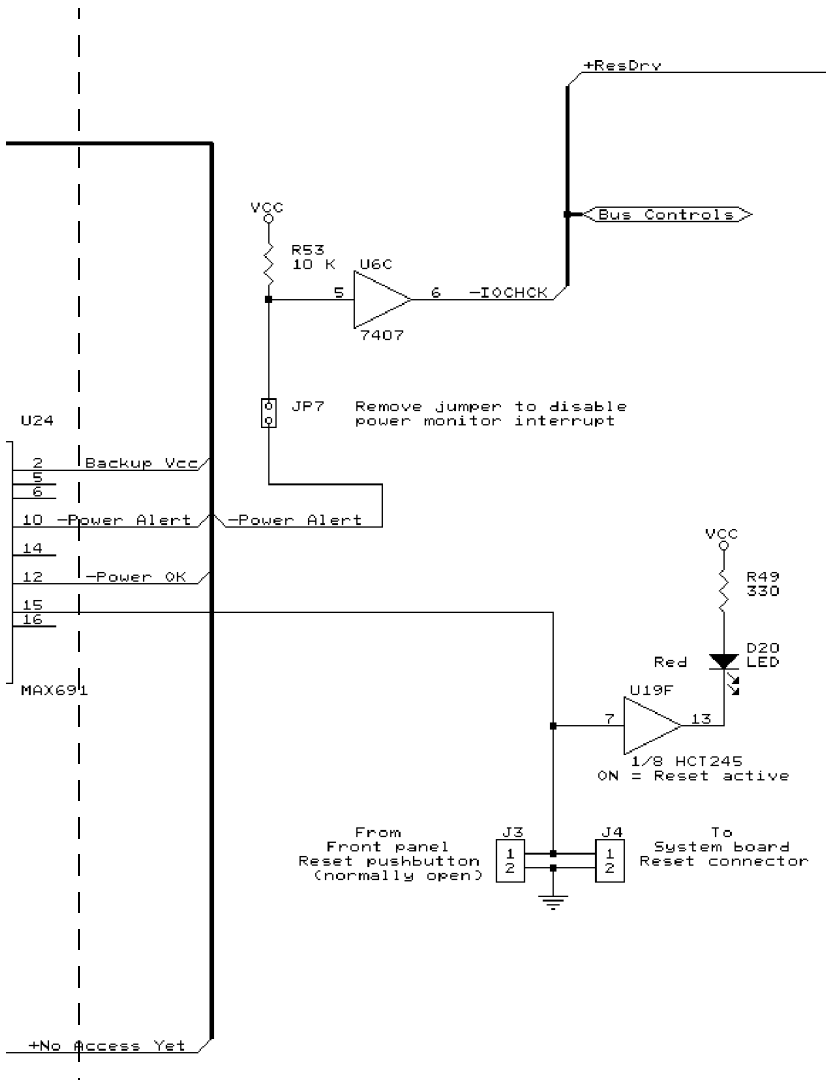
$$(0.01 \times 250 \text{ mAh}) \div (8 \text{ hours/day} \times 365 \text{ days/year} \times 10 \text{ years})$$

which works out to about 85 nA. The worst case occurs in continuously powered systems that always force charging current into the cell. In fact, those calculations may demand a much bigger battery than the RAM's standby current would lead you to expect, just to increase the allowable reverse charging current to a reasonable value. Refer to the actual UL specs for the details.



Schematic 2

The MAX691 monitors the power supply, warns of impending power failure, controls the backup battery and -CE switching, and includes a variable speed watchdog timer. The LS74 flipflop ensures that the watchdog times out after about 30 seconds following a hardware reset; any access to port 031C reduces the timeout to 1.6 seconds. Much of the remaining circuitry drives indicator LEDs that reveal what's going on. As with any high speed, mixed analog and digital circuit, applying proper bypassing to the MAX691 will prevent many headaches.



The MAX691 limits charging current to 10 nA typical, 100 nA maximum, and 1 μ A over the full temperature range. I decided to skip the series diode, as the Firmware Development Board will never sport a UL rating...

The UL requirements also specify a current limiting resistor in case the diode becomes damaged or shorted. The fault current in that situation must not exceed 5 mA regardless of battery capacity. The resistor value equals the maximum possible supply voltage minus the cell voltage divided by 5 mA:

$$(5.5 \text{ V} - 3 \text{ V}) \div 5 \text{ mA} = 500 \Omega$$

or the next higher standard value of 560 Ω . I included this resistor to prevent problems, should the MAX691 succumb to a static zap. Obviously, that line of thinking lacks consistency. The MAX691A, a '691 successor, includes a current limiting resistor on the chip to simplify your life a bit.

The MAX691 data sheet specifies a bypass capacitor on pin 2 to stabilize the internal voltage comparator and supply switch. The MAX691's internal transistor switch can source only 50 mA, requiring the cap to supply transient currents beyond that limit. If your circuitry requires more average current, the data sheet shows a current-boost circuit that doesn't affect the backup battery life when the power goes off.

The bypass cap must store enough energy to stabilize the voltage during the huge current spike shown in Figure 2. You should also bypass the static RAM and any other digital circuits at their sockets, as usual.

With clean power assured, the next step becomes controlling the CPU during the switchover. After all, it does no good to preserve data scrambled by a power-starved processor, does it?

Data Defense

Because the Original IBM PC power supply included a **Power Good** output signal, they're pretty much a standard feature on all supplies intended for use with current PCs. When **Power Good** was low, the CPU remained reset. After all the power supply voltages stabilized within their specified limits, **Power Good** went high and the CPU started up. When you flipped the Big Red Switch OFF, **Power Good** dropped *before* the supply voltages fell out of spec. In effect, the system always saw clean power while it was running normally and the power supply made sure it wasn't running at all when the power wasn't clean.

The system board activates the ISA bus **ResDrv** (RESet DRiVers) signal when it sees a hardware reset. In principle, **Power Good** going low should activate **ResDrv** and ensure all of the PC's circuitry stays reset.

However, to quote Solari, "[T]he above information ... is a combination of ... the IEEE P996 specification and various IBM technical reference manuals. It is sometimes unclear which platforms adhere to these specifications."

I've seen some PC power supplies, particularly for homebrew embedded systems, without a **Power Good** signal. Evidently, the designers depend on the system board's (possibly nonexistent) reset timing circuitry. In fact, one group I worked with simply tied the target system's **Power Good** input to a capacitor, ignoring the fact that the resulting, faked **Power Good** signal remained active long after the power went bad. I argued in vain for a power monitor chip, but, with the board already laid out, they preferred to kludge the cap instead of fixing the problem. *Ugh.*

The MAX691 monitors its supply voltage on pin 3 and triggers several actions when that input falls below specific levels. While these may not be strictly necessary in a PC with a solid supply, as long as we have the chip, we may as well put it to good use. If, for whatever reason, you are *not* using a standard PC supply, this circuit can help ensure that the RAM's contents remain valid, regardless of what happens to the rest of the system.

Recall that we must put the RAM chip into standby mode when the power fails. The MAX691's **-CE Out** signal tracks **-CE In** until the supply voltage falls below 4.65 V, whereupon the MAX691 unilaterally forces **-CE Out** high. This both disables the RAM and puts it into standby mode.

Unfortunately, while the MAX691 has a 50 ns nominal delay from **-CE In** to **-CE Out**, the maximum spec hits 200 ns. That may be OK for this relatively slow ISA bus application, but I felt I should show how to adapt it to faster systems. The MAX691A has a far more useful 10 ns nominal delay. All current versions of Maxim's other power monitors run at useful speeds, too, as you might expect.

The solution simply controls a faster logic gate with the **-CE Out** signal from the MAX691. As shown in Schematic 2, grounding the **-CE In** pin forces **-CE Out** low in normal operation. When the power fails, the MAX691 raises **-CE Out**. Although it's not exactly a DC signal, a few dozen nanoseconds one way or the other simply don't matter for a power failure warning. I called the signal **-Power OK** to indicate its new function.

With **-Power OK** low, the HCT32 gate in Schematic 1 delays the RAM chip select by only about 20 ns. When **-Power OK** goes high, it forces **-CE** high and

the RAM enters its low power standby mode. The CMOS gate drives the chip's **-CE** pin nearly to the supply voltage, ensuring that it meets the spec.

Obviously, you must power the external gate with the backup battery through the MAX691 **VOUT** pin! You should use an HCT gate, rather than C or HC, to ensure that its inputs respond to TTL switching levels. Pure CMOS gates, with **V_{IH}** specs well above the normal TTL **V_{OH}** level, won't work correctly when driven by TTL output voltages.

Processor Protection

With the data in RAM now safe from forgetfulness, Wouldn't It Be Nice If (a phrase, often found in design proposals, abbreviated as WIBNI) the CPU knew what was going on, too. After all, simply disabling the RAM may cause invalid data if the CPU halts in the midst of a multibyte update. Although the power may be failing, a millisecond or two can give you just enough time to put things in order.

The MAX691 provides an early warning of impending doom by monitoring the voltage on its **Power Fail Input** pin: when the voltage at **PFI** drops below 1.3 V, the **-Power Fail Output** pin goes low. The resistive divider and trimpot R50 shown in Schematic 2 set the trip point so that **-PFO** goes active well before the MAX691 disables RAM access. You can set the voltage without using a trimpot, but the pot lets you activate **-PFO** without actually blipping the supply.

Although you could wire **-PFO** through an inverting driver to one of the system's interrupt lines, all will be lost should interrupts be masked off when the power fails. The solution lies in the **-IOCHCK** (IO CHannel Check) ISA bus line, which activates the CPU's **NMI** (Non-Maskable Interrupt) pin. That interrupt cannot be ignored in normal operation, ensuring that the situation gets the CPU's attention.

Once the **NMI** handler gains control, it can take whatever steps you decide will ensure a safe and orderly system shutdown. With only a few milliseconds of power left, however, saving data to disk, sending a message out the serial port, or doing anything on a human time scale simply won't work. Think fast and think final!

The MAX691 activates its **-Reset** output when the supply voltage drops below 4.65 V. In a good PC with a standard supply, the spec for the +5 V power at the board connectors is 4.875 V minimum, which means that **Power Good** should fall long before the MAX691 triggers a reset.

The MAX691 also has a **+Reset** output for 8031-style microcontrollers with a high active **+Reset** input. Two additional power monitor outputs, **Battery On**

and **-Low Line**, may come in handy for some systems. Check the data sheet for further hints and tips.

To recap, the sequence of events during a power failure starts with **-PFO** activating the CPU's **NMI** input. Your **NMI** handler prepares for the coming shutdown, then enters a loop until either the MAX691 or the PC's **Power Good** circuitry detects an invalid voltage and activates the system **Reset** line. The MAX691 disables the RAM at the same time it activates **-Reset**.

When power comes back on, **Power Good** and the MAX691 together decide when the voltages fall within tolerance, then release the system **Reset** line. The CPU starts up, the BIOS takes control, and the system boots normally. When the MAX691 releases **-Reset**, it also enables the RAM and makes it ready for the first firmware access, with the write enable bit cleared to prevent changes.

Schematic 2 shows connections to both **ResDrv** and the system board **Reset** connector. The two are *not* identical: **ResDrv** is an ISA bus signal and **Reset** normally connects to the Reset switch on the front panel. You must not drive **ResDrv** and you do not have direct access to the signal that resets the CPU.

I kludged a small adapter for the **Reset** connection: the switch on the front panel plugs into the adapter, which then plugs into the system board. A pair of wires joins the adapter to a header on the Firmware Development Board. If you connect the fool thing backwards, the FDB's ground holds **Reset** low, making that an easy goof to find... your target system won't start up!

Firmware Supervision

The MAX691 has one additional feature that I believe should be in every embedded system: a watchdog timer. As you surely know already, a watchdog timer resets the system CPU after a predetermined interval following a transition on its input pin. The firmware (or hardware, in some systems) must wiggle that bit often enough to prevent the timer from timing out.

Presumably, correctly functioning firmware will periodically wiggle the watchdog timer's input, but locked-up or stalled code probably won't. When the watchdog times out, the ensuing system reset clears the slate and starts the firmware over again. Whatever the CPU is controlling must withstand a brief glitch while the system recovers its wits. If your system can't stand such an interruption, a watchdog won't work for you. However, you *must* provide some other way to detect lockups, because they *will* occur.

Trust me on this one.

The BIOS in a stock PC gets control when the CPU **Reset** signal goes inactive and remains in control until the disk boot starts your program. A typical system may require 20 seconds for this process and, should it use SCSI disk drives, can take far longer. Even a 20 s seizure probably lasts far longer than you're willing to wait during normal operation, when your firmware should be busy controlling whatever you have hitched up to the system. Obviously, we need a variable rate watchdog.

Some systems start up without a watchdog active or permit disabling the watchdog under firmware control, but I don't like those choices. A firmware fault or hardware glitch can (nay, *will*) disable the watchdog just before the CPU takes a permanent walk in the woods. A variable rate watchdog ensures that the reset *must* occur eventually, even if it may take a little more or less time than you'd like.

Schematic 2 shows how I adapted the MAX691's watchdog. The ISA bus **ResDrv** signal clears U18B, a LS74 flipflop. That bit holds the MAX691's **Osc Sel** input low, forcing the watchdog to run at a frequency set by capacitor C2. The 1 nF cap I used produces a watchdog timeout of about 30 seconds, long enough to load and start a (short) program from diskette before the first timeout.

The first time a program writes to port 031C on the Firmware Development Board, the hardware sets U18B, which raises the MAX691's **Osc Sel** and **Osc In** pins. With those inputs high, the watchdog uses its internal oscillator and times out after 1.6 seconds. That's fast enough for normal operation when your code should have control of the system.

Although the MAX691 data sheet has equations giving the external capacitor value for a given timeout, I've found that they provide only rough starting points. You should probably perform some experimentation to find the right value for your application and verify the results. Remember that a slow watchdog beats a fast one in most situations, as you'll rarely find your code speeding up as you add more functions. A system that resets itself once in a while can be rather disconcerting.

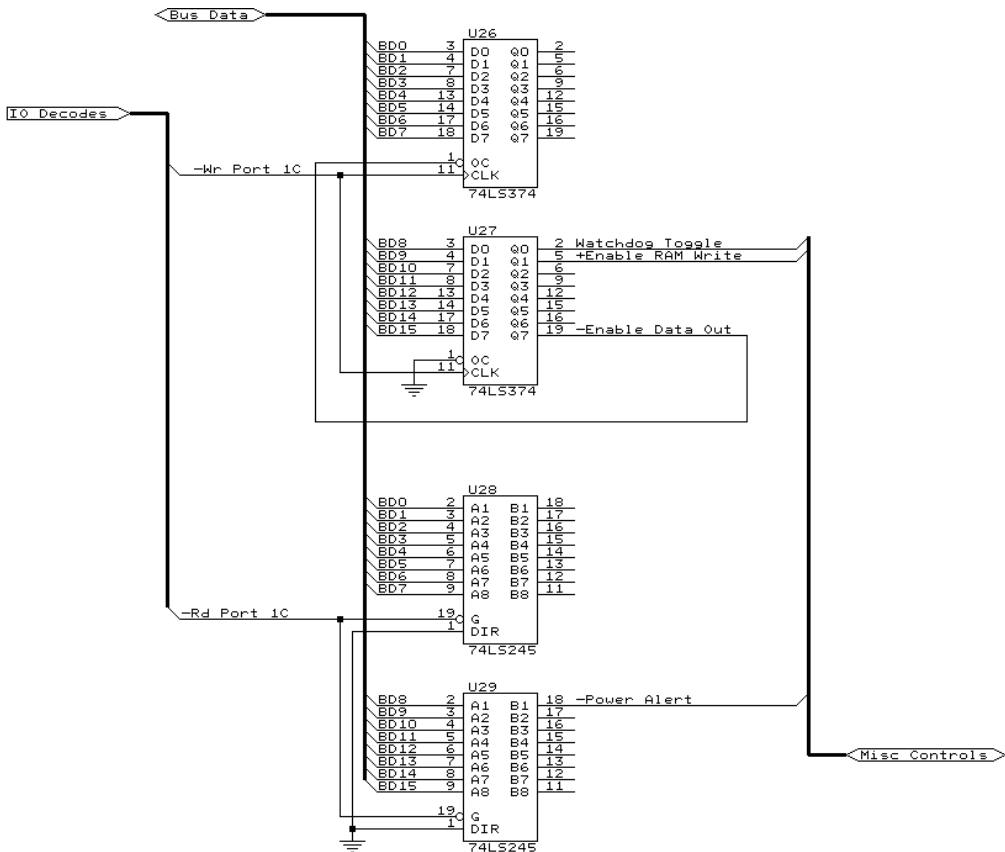
Incidentally, you might want to store a record of how and why you got reset for later analysis. Perhaps you could trigger an interrupt handler just slightly before the watchdog clobbers your system, save the current state, then wait for the end?

Schematic 3 shows the new I/O bits on port 031C, which uses hardware essentially identical to the LED digits and DIP switches on port 031E that we built in Chapter 3. Although only three of the new bits see action here, I've got plans for the remainder, never fear.

The Firmware Development Board now sports several more indicator LEDs. You can tell at a glance when RAM writes are enabled, **Reset** goes active, the

Schematic 3

These gates provide the input and output bits used by this chapter's circuitry. The unused bits will come in handy later on.



watchdog toggles, and how long a watchdog timeout will take. The LED drivers reside in U19, the same LS245 DIP that sends 82C54 timer interrupts to the ISA bus. Recall that we tied its output enable pin low when we set up those interrupts.

Getting Down to Code

The RAM circuitry bears enough resemblance to the (E)EPROM we covered in Chapter 6 that I could convert the **MemTest** program into **RAMTest** by just ripping out the EEPROM timing code and expanding the memory tests to include all

32 KB. With nothing much new here, I won't show the listings. Do, however, examine the complete source code and use it to check out your wiring.

Although a watchdog timer may be vital in a production system, it becomes a serious nuisance while you're developing and testing code like `RAMTest`. I disabled my board's watchdog by simply yanking the system board **Reset** connection. The red LED then indicates when the MAX691's **Reset** output goes active. If that LED ever goes on, you've goofed!

To verify the watchdog and power monitor code, run the companion program called `DogTest`. Connect the MAX691's **-Reset** output to the system's **Reset** connector and boot `DogTest` from diskette. If it gets control before the initial 30 s timeout expires, as it should, it will set up the interrupt vectors and begin toggling the watchdog output.

Should `DogTest` *not* get control, it'll be pretty obvious. Your PC will boot, begin loading `DogTest` from diskette, reset, and start all over again.

A watchdog timer doesn't care how often you toggle its input bit, as long as you do it often enough. If, however, there's an LED on that bit, it is a Very Good Idea to produce a regular heartbeat. An irregular heartbeat LED can be quite unsettling, even when it indicates perfectly good code in normal operation.

I use heartbeat LEDs as output devices: a regular blink signifies normal operation, while long and short blinks report errors. The firmware can be pretty straightforward: a timer interrupt handler takes care of timing, while the mainline code sets up the bit patterns. I've used this trick on many systems and you can probably adapt it to yours.

Listing 1 shows `DogTest`'s timer interrupt handler. The mainline code attaches this function to `INT_1Ch`, which the BIOS invokes after processing each 54.9 ms timer tick. I divided that rate down by three to shift out about 6 bits per second. The interrupt handler thus runs through all sixteen bits in the `watchBits` variable in about 2.6 s.

The interrupt handler sets `watchPending` when it finishes sending all 16 bits. If `watchPending` remains set after 16 more bits, the interrupt handler enters the tight loop at `WD_Lock`. Because the watchdog output bit no longer toggles, the MAX691 will eventually reset the system.

The mainline code thus has two responsibilities: it must load a bit pattern into `watchBits` at least once and it must clear `watchPending` at least every 32 bit times (ideally, every 16 bit times) to prevent a timeout. The maximum delay until a

Listing 1

Producing a regular heartbeat on the watchdog pin requires an interrupt handler attached to a timer tick. This code rotates a 16-bit variable and sends the high-order bit to the watchdog pin and LED. To avoid sending the bits faster than the eye can follow, it counts BIOS timer interrupts and sends out one bit when the count reaches WATCH_RATE. The mainline code must reset the WatchPending flag at least once every 32 bit periods to prevent this handler from forcing a watchdog reset.

```

HandlerWD() {
asm {
        PUSH    AX                save bystanders
        PUSH    DX
        PUSH    DS
        MOV     AX,CS             aim at our segment again
        MOV     DS,AX

*
* Count down the interrupts until we need a watchdog update
        DEC     <WatchDivide
        JNZ     WD_Ret
        MOV     <WatchDivide,#WATCH_RATE

*
* Decide if a new watchdog word is needed
* If it is, and the mainline code is jammed, we lock up and die
WD_Tick    DEC     >WatchCounter
        JNZ     WD_GO             nonzero says use old bits
        CMP     >WatchPending,#0  has mainline code reloaded the bits?
        JE      WD_Load           zero says yes, so we are golden
        MOV     DX,#LED_ADDR_A    nonzero says we have trouble
        MOV     AX,#~$8000        left decimal point flags the problem
        OUT     DX,AX
WD_Lock    JMP     <WD_Lock        stay here until watchdog timeout
*
WD_Load    MOV     AX,WatchBits    fetch new bits
        MOV     WatchShift,AX      ... for the shift reg
        MOV     >WatchCounter,#16 reload the counter
        INC     >WatchPending     set flag for mainline code

*
* Blip the watchdog output to ensure a transition every time
WD_GO      MOV     DX,#CTLS_ADDR_A set up for watchdog output
        MOV     AX,CtIsCopy       get existing bits
        AND     AX,#~WATCHDOG_A   send a low (LED ON)
        OUT     DX,AX
        Punt
        OR      AX,#WATCHDOG_A    send a high (LED OFF)
        OUT     DX,AX

*
* Rotate the watchdog bits and send the high one
* We flip the bit so 1 turns the LED ON like it should
        ROL     >WatchShift,1     get high-order bit in C
        JNC     WD_Z              clear says leave the output high
        AND     AX,#~WATCHDOG_A   set says make output low
WD_Z        OUT     DX,AX          send it out
        XOR     AX,#WATCHDOG_A    flip the bit back again
        MOV     CtIsCopy,AX       save for next time

*
WD_Ret     POP     DS             restore bystanders
        POP     DX
        POP     AX
        POP     BP
        IRET                    restore stacked flags
}
}

```

watchdog reset occurs will be the 5.2 s required to shift all 16 bits out (twice) plus 1.6 s after the last bit, or about 6.8 s overall.

You can clear `watchPending` in your main loop, as long as no code takes more than a few seconds. Watch out for things like user input prompts... resetting the system shortly after presenting a prompt on an LCD panel won't endear you to your users. Especially if it occurs just about when they noticed that prompt.

The most soothing bit pattern seems to be FF00, a reassuring pulse with 1.3 s ON and 1.3 s OFF. AAAA produces an exciting 3 Hz blink, while F140 send a "one long, two shorts" blink code that might indicate a particular failure or error condition. You can do a surprising amount with 16 bits if you think about it for a while. You can always go to 32 or more bits if you prefer: it's just a variable.

Note that setting `watchBits` to 0000 produces a perfectly valid, albeit dull, pattern that does *not* cause a watchdog timeout. The interrupt handler forces a transition between each pair of bits, pulsing the watchdog every 165 ms regardless of the heartbeat bit values. If you look closely at the LED in a dark room, you can actually see those 1.3 μ s pulses running at a 0.001% duty cycle. Try it!

`DogTest`'s main loop is quite simple: it checks and resets `watchPending` to keep the interrupt handler happy, copies the DIP switches into `watchBits` so you can

Figure 3

A system board memory parity check or the ISA bus -IOCHCK signal can trigger a Non-Maskable Interrupt. Your firmware can determine which input is active and mask it off by using these bits in I/O port 0x61. Some systems have additional NMI sources with different controls. Bit 7 in port 0x70 must also be zero to enable the CPU's NMI input.

Bit	Read Status	Write Function
7	1 = System board parity check	n/a
6	1 = IO channel check	n/a
5	1 = Timer 2 output bit	n/a
4	Toggles with each RAM refresh	n/a
3	0 = IO channel check enabled	0 = Enable IO parity check
2	0 = System board parity check enabled	0 = Enable parity check
1	1 = Speaker data enabled	1 = Enable speaker
0	1 = Gate Timer 2 output to speaker	1 = Gate Timer 2 to speaker

experiment with different bit patterns, and writes a counter value into the FDB's LED display so you can see something happening.

DogTest also accepts a command from the serial port connected to your host system: if you type **1** on the host's comm program, **DogTest** will stop clearing **watchPending** and force a watchdog reset. The interrupt handler turns on the decimal point of the LED's left digit just before it enters the final loop. Watch carefully to see the MAX691 activate the **Reset** LED (and reset the system) roughly 1.6 seconds later. It should reboot and start **DogTest** as it did before.

Unmasking the NMI

By definition, the CPU cannot ignore a Non-Maskable Interrupt. However, the IBM PC and its descendants include circuitry to prevent a signal from reaching the CPU's **NMI** pin. While this may seem contradictory, the system cannot start, let alone run correctly, with what's called a hot **NMI**.

For example, if an **NMI** occurs before the firmware validates RAM and loads the stack pointer, the system will certainly crash when the handler tries to return. The CPU will accept an **NMI** immediately after its **Reset** input goes inactive so, with **NMI** stuck active, the CPU cannot even run diagnostics to pinpoint the problem.

However, leaving **NMI** disabled all the time is an Exceedingly Bad Idea. IBM's PC AT designers picked a distressingly clever way to enable **NMI** without special programming. The MC146818A Real-Time Clock has 64 bytes of battery backed RAM addressed by the byte written to I/O port 0x70. The address circuitry inside the clock chip uses only six bits and ignores the high-order ones. The designers added an external latch that catches data bit 7 and drives a gate controlling **NMI**. Simply write address 0x80 instead of 0x00 to port 0x70 and mask the unmaskable.

Wish you'd thought of something like that for your last project?

In addition to the latch holding the mask bit, the system board has additional circuitry that sets the latch during each hardware reset. It remains set until the BIOS writes an RTC address between 00 and 7F, an event that happens only *after* the BIOS makes sure a hot **NMI** won't cause any problems.

A variety of sources can activate **NMI**, depending on exactly which system you have. The two standard sources, the system board memory parity check hardware and the ISA bus **-IOCHCK** signal, should (but may not) exist on all systems. Several bits in I/O port 0x61 control these signals, as shown in Figure 3.

DogTest's **NMI** handler, shown in Listing 2, resembles the interrupt handlers you've seen before, with one key exception. Because the **NMI** does not pass through the external 8259 interrupt controller chips, the handler must *not* send an **EOI** to either 8259 in response to the **NMI**.

The code examines the MAX691 **-PFO** bit through port 031C; a power failure is impending if it finds a zero. Otherwise, the code simply invokes the previous handler set up by the BIOS during the power-on sequence.

Listing 2

This routine decides if a Non-Maskable Interrupt was caused by the MAX691's Power Fail detector. If so, it write protects the RAM, lights a decimal point, and enters a spin loop waiting for Reset. If not, it passes control to the NMI handler set up by the BIOS.

```
HandlerNMI() {
asm {
*
        PUSH    AX           save bystanders
        PUSH    DX
        PUSH    DS
        MOV     AX,CS        aim at our segment again
        MOV     DS,AX
*
* Check to see if the power fail bit is active
*
        MOV     DX,#STAT_ADDR_A
        IN      AX,DX
        Punt
        TEST    AX,#PWR_GOOD_A
        JNZ     NMI_Chain    nonzero says not our problem
*
* We have a power failure, so write-protect the RAM and lock up
*
        MOV     DX,#CTLS_ADDR_A    turn off the write-enable bit
        MOV     AX,#~NV_WENABLE_A
        OUT     DX,AX
        Punt
*
        MOV     DX,#LED_ADDR_A    show that we are locking up
        MOV     AX,#~$0080        ... with right decimal point ON
        OUT     DX,AX
        Punt
*
NMI_Lock    JMP     <NMI_Lock    jam up here until next reset
*
*
* Chain to previous NMI handler
*
NMI_Chain    POP     DS           restore bystanders
        POP     DX
        POP     AX
        POP     BP
        JMP     CS:0:>Int020ff    indirect to old handler
}
}
```

The Micro-C assembler requires the rather strange syntax in the last `JMP` instruction to emit an indirect `JMP FAR` with a `CS` segment override. Borland's `TASM` uses this somewhat less opaque notation:

```
JMP [DWORD CS:Int02Off]
```

Because the CPU blocks further Non-Maskable Interrupts until it executes an `IRET` instruction, you could replace the tight loop at `NMI_LOCK` with a `HLT` instruction. I favor a loop over a `HLT`, because, in a pinch, I can easily add a few instructions that toggle an output bit and flag the spot on a scope. Take your pick.

Release Notes

The code for this chapter includes source and `HEX` files for `RAMTest` and `DogTest`. Remember: boot `DogTest` directly from diskette and make *sure* it gets control before the `MAX691` resets the system! Otherwise, you'll watch your system boot continuously until you turn the power off.

I've also tweaked the `LoadEXT.ASM` routines you saw in Chapter 6. You can now lob a BIOS extension from diskette into either `EEPROM` or battery backed `RAM`, while setting the checksum on the fly.

Compare the specs on the `MAX691` with the improved `MAX691A`. Obviously, you'd use the `A` version for new projects. Maxim now provides all their datasheets on a single CD-ROM, as well as their Web site, which makes it particularly easy to find specific chips. See the Sources appendix for the Web pointer. Oh, yes, you can order the CD-ROM online, too.

Chapter 8 explores more **NMI** code topics, including how to handle glitches on the **NMI** input that may cause problems on some systems.

OK, that's enough hardware for while! If you can't start doing embedded PC work with what we've got now, it's time to dust off those `COBOL` manuals.

8 Ticks, Pops, and Restarts

Traditional embedded systems start up at the flip of a switch. Even the PC's built-in ROM BIOS lights up immediately. However, as we saw in the previous chapter, loading code from diskette can take quite a while and may conflict with your watchdog timer. There's no quick-and-dirty cure for that, but converting your code into a BIOS extension can help.

In this chapter, I'll explore BIOS extensions in more detail with a set of routines that capture interrupts, support the Firmware Development Board's power failure detection hardware, and record information in nonvolatile storage. You can use either the EEPROM or the battery backed RAM circuitry, as long as the firmware can enable and disable the memory chip's **-WE** line.

The Key to the Code

Schematic 1 shows the new hardware you'll need: a pushbutton switch with a pullup resistor driving bit 9 of the input port at 031C. This may be barely worth warming your soldering iron, but, every now and then, we need an easy one.

Holding that button can become awkward at times, particularly when you're also maneuvering a scope probe or two, so I rewired the front panel keyboard lock switch in parallel with the button. Because we have yet to use the keyboard, I figured the keyboard lock might be superfluous.

Incidentally, should you ever come up against a "locked" PC clone, just whip out your Swiss Army knife's Phillips blade, unscrew the clone's case, yank (or slice) the lock switch wires, and fire that sucker up. I trust I'm not compromising the security of what was once the Free World by letting that trick out of the bag. The Original IBM PC AT had a lock that disabled the keyboard *and* secured the metal cover to prevent just such an assault.

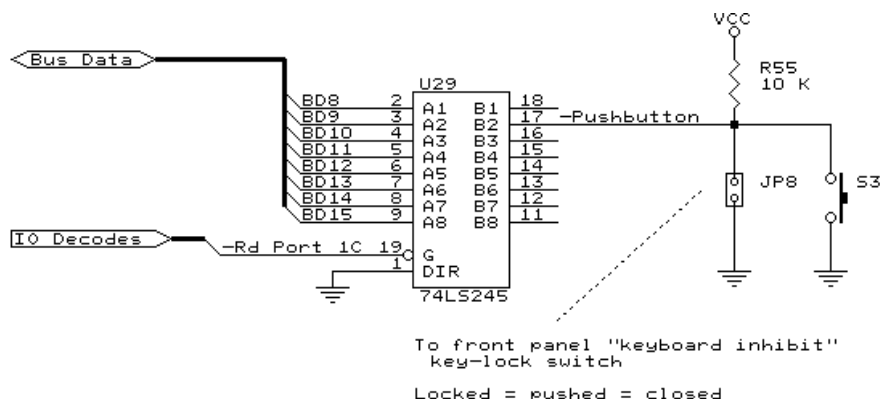
So much for the hardware. Now, on to the code!

Extension Essentials

The Original PC BIOS didn't permit any extensions, which led to some truly remarkable kludges as each vendor devised different and mutually incompatible ways to glue new functions into old PCs. The method we'll use dates back to a PC BIOS revision slightly before the XT. That means, for all intents and purposes, every IBM PC now handles BIOS extensions the same way.

Schematic 1

This simple hardware, a button and a resistor, allows you to skip over a malfunctioning BIOS extension during a boot and prevent a system crash. The supporting circuitry appeared in Chapter 7 as part of the battery backed RAM and watchdog controls.



The part of `FDBExt.ASM` shown in Listing 1 sets up the 55 AA signature, length, and checksum bytes required by the BIOS extension scan. Recall that the checksum byte in the source code must be zero, because our diskette boot loader computes the actual checksum as it copies the extension into the Firmware Development Board.

The code at `BootEntry` forms an escape hatch I suggest you build into all your extensions, at least while debugging them. With the external button pressed (or the key lock switch ON) while booting, `FDBExt` simply updates the LEDs and returns to the BIOS. This can save your bacon when your new extension crashes the BIOS boot sequence. Trust me. It can happen to you, too.

With the switch released or the lock switch OFF, the code shown in Listing 2 makes the whole software development process I'm using in this chapter work correctly. As Steve Ciarcia often puts it, "Let me explain..."

The Case of the Missing PSP

`FDBExt`, written in Borland's Turbo Assembler `TINY` memory model, produces an ordinary `COM` file. As far as `TASM` and `TLINK` can tell, the program will run under DOS with all the usual DOS assumptions and restrictions. Of course, if we actually *do* run it under DOS, it won't work very well at all.

Listing 1

This code fragment resides in the battery backed RAM at C800:0000 on the Firmware Development Board. The BIOS passes control to the instruction at offset 0003 (just after the length byte) during the power-on sequence. This code first checks the pushbutton switch input bit; if the button is pressed, the code immediately returns to the BIOS.

```
; This does not start at offset 0100, so be careful about data accesses!
; Actual execution is at absolute address C800:0003...

CODESEG
STARTUPCODE

DB      055h                ; signature
DB      0AAh
DB      2                   ; length in units of 512 bytes

JMP      SHORT BootEntry    ; force two-byte jump
DB      00h                ; loader sets this value

;--- constants that must be stored within the checksummed region
RevCode  DW      0001h      ; current revision level

;--- if pushbutton is down, exit without doing much
BootEntry:
MOV      DX,STAT_ADDR
IN       AX,DX
TEST     AX,PUSHBUTTON
JNZ      SHORT Continue    ; nonzero means not pushed

MOV      AX,NOT 0101h      ; show -- to track our path
MOV      DX,LED_ADDR
OUT      DX,AX

RET      RETF              ; return to normal BIOS boot

Continue:
```

As you know by now, COM files contain an exact binary image of the program's code and data. They date back to the days of CP/M and 8080 CPUs, when 64 KB of RAM was all even a big spender could have. The CP/M operating system reserved the first 256 bytes of RAM to control the 8080's reset and interrupt vectors and provide some entry points. Therefore, it loaded your COM programs at absolute RAM address 0100.

MS-DOS adopted much the same memory layout, except that 64 KB of RAM suddenly seemed not quite so much after all. A COM file fit neatly into one 64 KB segment atop the reserved 256 bytes, which, still filled with operating system stuff, became known as the Program Segment Prefix. DOS handles EXE files somewhat differently, their PSPs live in a different segment, and we'll get to them later, but,

Listing 2

With the switch open, the next step adjusts the segment registers. The RETF instruction loads the new values into CS and IP from the stack. Note that this code uses the FS and GS segment registers found in '386 and higher CPUs and will not run on earlier CPUs.

```
;--- adjust CS and DS to simulate the normal COM situation
; we need both code and data starting at offset 0100 rather than 0000
; C800:0000 is also C7F0:0100, so we just subtract 0010 from the segments
; Storing this CS in the vectors allows normal access after an interrupt

        MOV     AX,NOT 0100h           ; show single - here
        MOV     DX,LED_ADDR
        OUT     DX,AX

        MOV     AX,CS
        SUB     AX,0010h               ; adds 100 to offsets in segment
        PUSH    AX
        PUSH    OFFSET BootStart
        RETF                            ; set CS and IP to new values

BootStart:
        MOV     AX,0057Eh              ; show r0 on LEDs to mark entry
        CALL    ShowBits

        PUSH    DS                     ; preserve seg regs
        PUSH    ES
        PUSH    FS
        PUSH    GS

        MOV     AX,CS                  ; set up DS to match CS
        MOV     DS,AX
        MOV     ES,AX

        MOV     AX,0
        MOV     FS,AX                  ; FS points to 0000:xxxx

        MOV     AX,0040h
        MOV     GS,AX                  ; GS points to 0040:xxxx

        CALL    OpenRAM                ; enable writes

        INC     [ES:ResetCtr]          ; count this reset
```

for now, the key point remains that COM files start at 0100 for simple, historical, largely arbitrary reasons.

Although all of the code and data addresses within a COM file assume that it's loaded at offset 0100, the actual disk file does *not* include those first 256 bytes. The instruction at offset 0100 within the program's code segment resides at offset 0000 relative to the start of the *file*. DOS must set up the segment registers so that the offsets become correct within the segment where the program will be loaded.

The diskette boot loader in Chapter 1 simulates this process. It loads your program from diskette into RAM at address 1000:0100 and places nothing at all in the first

256 bytes of the segment at 1000:0000. Although there's no PSP, that trick let us use standard COM files without invoking a specialized linker. As long as our program doesn't expect anything in the PSP, having a blank (or random) one is OK.

The PC's BIOS, on the other hand, knows nothing of this. When it finds our Firmware Development Board extension, it passes control to the branch instruction with CS:IP set to C800:0003. Because our extension will set interrupt vectors as well as change data, it must somehow adjust the segment registers on the fly.

The solution involves simple subtraction, because a given physical address can be represented by many different segment and offset values. The CPU hardware shifts the segment register left by four bits, adds the offset, and takes the result as the physical address. At least that's the case in real mode, which is all we require now; protected mode programming is *entirely* different.

In our situation, the branch instruction at C800:0003 resides at physical address C8003. We can also call that address C7F0:0103, because $C7F0 + 00103 = C8003$. Thus, if we reload the segment registers with C7F0, rather than C800, and change IP from 0003 to 0103, all our offsets become correct and we can use COM files for BIOS extensions.

The easiest way to reload both CS and IP is by yank them from the stack with a RETF (Far Return) instruction. Listing 2 shows the trick in all its glory... not particularly impressive, hmmm? Just try to figure out what it does without some commentary, though. Notice that the PUSH instruction operates on the value of OFFSET BootStart as defined in the COM file, not BootStart's absolute offset within segment C800.

COM programs generally assume that all four of the CS, DS, ES, and SS registers have the same value, but CS and SS hold the essential values for our code. I load DS and ES from the adjusted CS value. SS cannot point into the nonvolatile memory, because that RAM has hardware write protection.

Fortunately, as long as we only PUSH, POP, CALL, and RET from the stack, whatever values the BIOS put in SS and SP will work fine. I have not investigated how deep the default BIOS stack may be. Should your program need lots of stack space for some peculiar reason, you should certainly create a stack somewhere else.

Although I didn't find this written down anywhere, the BIOS in my system requires that you restore at least DS and ES before the extension returns. As usual, the final RETF restores CS from the stack. I save and restore all the segment registers, even though the requirements surely depend on which BIOS you're using.

`FDBExt` also marks a departure from the code you've seen so far: notice that I'm now using the `FS` and `GS` segment registers that appear only in '386 and higher CPUs. As a result, this code will *not* run on 8088 or 80286 systems. I don't include any tests for the CPU type, as I assume we're all responsible folks around here. Don't try this on your old clunker PC just to see what happens... it won't work!

Yes, we could rewrite `FDBExt` to work on any 80x86 CPU, but it's long past time to start using hardware that's been around since 1985. OK?

Capturing Interrupts

The remainder of `FDBExt`'s initialization code captures the BIOS timer and Non-Maskable Interrupt vectors. You've seen similar routines in Chapters 5 and 7, so refer back to those source files if you need a refresher.

Listing 3 shows the timer interrupt handler. The Firmware Development Board's RAM has its write protection hardware active, forcing each handler to enable writes before updating its variables. The `CS` segment stored in the interrupt vector equals the `DS` value set up in Listing 2, allowing the `INC` instruction to reach `TickCtr` using `CS` without saving, loading, using, and restoring `DS`.

Because the RAM write enable bit shares the same port as the watchdog timer bit, I made the `OpenRAM` and `CloseRAM` routines toggle the watchdog on each BIOS

Listing 3

The BIOS extension captures the BIOS timer interrupt to count the ticks since the most recent reset. The value of `CS` stored in the interrupt vector allows access to the extension's variables in nonvolatile memory. Because this code fragment was assembled in '386 mode, the `INC` instruction increments a 32-bit counter in one shot and the final `JMP` uses the `SMALL` keyword to specify that `OldTimer` holds a `seg:off` address.

```
PROC      TickHandler
PUSH      AX
PUSH      DX

CALL      OpenRAM           ; enable writes
INC       [CS:TickCtr]      ; TickCtr is 32 bits wide!
CALL      CloseRAM         ; disable writes

POP       DX
POP       AX

JMP       SMALL [CS:OldTimer.DWORD]

ENDP      TickHandler
```

timer tick. Measuring the bit's active time shows that the interrupt handler requires about 30 μ s on a 33 MHz '386.

As I mentioned in Chapter 7, it's generally a Bad Idea to toggle a watchdog from a timer interrupt, because the main routine can crash while the timer tick continues running. However, this trick can keep the watchdog at bay while loading a big program from diskette. The mainline code can always capture the timer tick and implement my favored method after it starts running.

Assuming it gets that far, of course! If something goes wrong during the disk boot or program setup process, we've just defeated the entire purpose of the watchdog timer. Think about it, then decide just how paranoid you should be. Which is worse: a failure that you don't detect or a system that doesn't start up correctly?

Enabling '386 assembly mode has some interesting side effects. Even though the `TickCtr` variable occupies four bytes, the assembler updates it with one 32-bit `INC` instruction. The `JMP` at the end of the routine chains to the previous interrupt handler as usual, but you must specify `SMALL` to indicate a 16-bit `seg:off` value, rather than a 32-bit `LARGE` offset in the current segment.

I like that sound... even in real mode!

Failing Power

Our BIOS extension responds to power failures by write protecting the RAM and spinning in a safe shutdown loop. While writing this code, though, I uncovered a nasty bug: **NMI** glitches. While they shouldn't pose a problem in most systems, it's worth thinking about them if you're using a power monitor.

The Firmware Development Board includes a trimpot that adjusts the voltage on the MAX691's **PFI** pin. The correct setting activates **-PFO** when the supply voltage falls near the system's lower tolerance limit; say -5% on a $\pm 10\%$ system. The remaining 5% gives you enough time to shut the system down before the voltage goes completely out of tolerance.

In small microcontroller systems, the MAX691 may be the only source of Non-Maskable Interrupts. In our situation, though, many parts of a PC can contribute to the **NMI** signal. Our handler must examine the board's power failure status and either chain to the previous **NMI** handler if it finds **-PFO** high (inactive) or shut down the system when it sees a low **-PFO** input.

Here's the problem: if the supply voltage falls slowly enough, a small supply glitch that would normally be well within tolerance can trigger the power failure

comparator and generate a Non-Maskable Interrupt. By the time the CPU responds to the **NMI** and checks the **-PFO** status bit, however, the glitch has long since Gone Away, even though the voltage continues to fall.

You can simulate this by very slowly adjusting the **PFI** trimpot. Tease it until the system shuts down, then leave the trimpot unchanged. The system will probably shut down sporadically every time you reboot it.

With none of the **NMI** sources active, the interrupt handler chain eventually passes control to the default BIOS handler. Guess what? On my system, the default handler disables further **NMI**s caused by the ISA **-IOCHCK** signal. Thus, when the power really *does* fail after the glitch, the **NMI** handler never gets control.

If the MAX691 presents the only **-IOCHCK** interrupts in your system, your interrupt handler can check the status bit in I/O port 61h to verify that the **NMI** actually came from the bus. Because the system board latches that status bit when **-IOCHCK** goes active, the status will not go away even when the glitch vanishes.

However, if you have several I/O boards that can produce **-IOCHCK** interrupts, the situation becomes somewhat messier. Your handler must test the board's status and chain to other handlers if it finds no problem. An **NMI** glitch from your hardware will set the **-IOCHCK** latch before vanishing. Your handler will find nothing wrong with the PC's power, then incorrectly pass control to other handlers that will also find nothing wrong on their boards.

It should be obvious that this a situation that's best avoided.

In Schematic 2 in the previous chapter, C4, a 1 nF capacitor on the **PFI** trimpot's wiper, filters the glitches with a 10 μ s time constant that made the trimpot teaseproof. You should, of course, evaluate this trick in your system to verify that it does not delay the interrupt too much during a real power failure.

An alternative approach, described in the MAX691A data sheet, applies hysteresis around the power failure comparator. Because **-PFO** switches low as **PFI** drops, a high value resistor between those two pins will yank **PFI** down and prevent the end of the glitch from restoring **-PFO**. Assuming, of course, that the comparator's propagation time doesn't glitch it the other way!

You can also add a digital latch that preserves the **-PFO** glitch, much as the system board latches **-IOCHCK**. Remember to include hardware that clears the latch on each hardware reset to prevent a hot **NMI**.

In any event, one of the conditional assembly options shown in Listing 4 inserts a timing loop that starts on the first **NMI**. It displays the loop count on the LEDs until the next **NMI**, at which time it locks up the system. You can use `ExtTest.BIN` to see how this problem looks on your system.

The CPU automatically disables all interrupts within the **NMI** handler, preventing watchdog timer updates during the final lockup loop. The MAX691 times out and resets the system shortly after the second **NMI** occurs.

Listing 4

This NMI handler normally shuts down the system in response to a power failure. If the supply voltage falls very slowly or if you tease the trimpot controlling PFI, you can get a glitch on NMI that vanishes by the time this routine gets control. The code shown here includes an optional section that displays the elapsed time from the first NMI to the next, then locks up the system. If these glitches pose a problem in your system, the code can also lock up in response to any NMI caused by the ISA -IOCHCK input.

```

PROC      NMIHandler

PUSH      AX
PUSH      DX

CALL      OpenRAM          ; enable writes
INC       [CS:NMIctr]      ; record this NMI
CALL      CloseRAM         ; disable writes
IF        COUNT_NMI        ; show delay?
MOV       CX,0             ; set up the counter

@@wait:   MOV       DX,LED_ADDR      ; show the loop counter in binary
          MOV       AX,CX
          NOT       AX
          OUT       DX,AX

          IF        USE_IOCHCK

          IN        AL,SYS_CTL5      ; look at IOCHCK flag
          TEST      AL,IOCHCK
          JNZ       @@Kapat         ; lock up when it goes high

          ELSE

          MOV       DX,STAT_ADDR     ; check power status
          IN        AX,DX
          TEST      AX,PWR_GOOD
          JZ        @@Kapat         ; lock up when status goes low

          ENDIF

@@Kapat:

```

Listing continues on next page

Listing continued from previous page

```
        INC        CX
        JMP        @@wait

        ELSE

        MOV        AX,1510h            ; show ni (more or less)
        MOV        DX,LED_ADDR
        NOT        AX
        OUT        DX,AX

        IF         USE_IOCHCK

        IN         AL,SYS_CTL5        ; look at IOCHCK flag
        TEST       AL,IOCHCK
        JNZ        SHORT @@Lockup     ; lock up when it goes high

        ELSE

        MOV        DX,STAT_ADDR       ; check power status
        IN         AX,DX
        TEST       AX,PWR_GOOD
        JZ         SHORT @@Lockup     ; zero = power NOT good...

        ENDIF

        ENDIF

        POP        DX
        POP        AX

        JMP        SMALL [CS:01dNMI.DWORD]

@@Lockup:  MOV        AX,08080h        ; both decimal points
          CALL       ShowBits

@@Stall:   JMP        @@Stall

          ENDP       NMIHandler
```

Resets and the Worst Hack

The Original IBM PC AT's design engineers faced a serious problem. They needed a way to get their shiny new 80286 CPU back into real mode, even though the chip had no way to shut off its Protected Mode Enable bit. The 80826 emerged from hardware reset in real mode, but once the program entered protected mode, the 80286 architecture provided no way back. Their solution stands as a monument to engineering ingenuity.

The AT included an 8042 microcontroller managing a variety of tasks implemented with discrete logic in the Original PC. The designers simply added a command to the 8042's repertoire that toggled the 80286 CPU's **Reset** line.

Blam... back to real mode!

However, the BIOS normally clears the system RAM and runs power-on diagnostics immediately after a hardware reset, which was not quite what they wanted. The engineers reserved a byte at address 0F in the Real-Time Clock's battery backed CMOS RAM to indicate the reason for the shutdown.

Before the BIOS gets too far along in its reset sequence, it asks the keyboard controller what caused the reset to occur. If the controller reports that it executed a **Reset** command, as opposed to participating in a power-on or manual reset, the BIOS reads the shutdown reason code from the clock's RAM. If that byte indicates a transition from protected to real mode, the BIOS branches directly back to the mode switch routine.

The only reason *you* think it's a kludge is that *you* didn't design it. It's really a clean, general, and useful way around an otherwise insurmountable hardware limitation. Remember the Consulting Engineer's First Principle: you don't get paid if the system doesn't work.

Intel got the message loud and clear. Starting with the 80386, all their CPUs enter and exit protected mode at the flip of a bit (well, all right, you need a little setup and takedown code on either side of the bit-flipping instruction, too). By that time, however, a considerable body of software used the Officially Approved '286 method. You can even buy hyperthyroid keyboard controllers with special fast-path hardware logic to recognize and speed up the **Reset** command. I kid you not.

The shutdown reason code in the clock's RAM can select one of several different routines after a reset. Most are ill-suited for civilian use, but one may come in handy in certain desperate situations. I'll show how to use it and you figure out when the trick might be appropriate. Fair enough?

If the shutdown reason code is 0A, the BIOS vectors through the pointer stored at address 0040:0067 to the restart code. That code is normally within the BIOS, but you can redirect the BIOS to your own routine. Because the DRAM refresh hardware continues to run during the shutdown, you regain control immediately after a hardware reset with nearly everything intact. Of course, all the CPU registers except **CS:IP** lose their values, so you must consider a few, ah, minor details that I'll leave as an exercise.

Listing 5a shows **FDBExt**'s rudimentary restart routine, which simply increments a counter and sends a second **Reset** command to the keyboard controller. The BIOS clears the shutdown reason code before branching to the routine and the hardware treats the second reset as a complete, normal, power-on reset.

Listing 5a

This code in `FDBExt.ASM` gains control after the keyboard controller blips the CPU's Reset line. The BIOS checks the shutdown reason code at address 0F in the Real-Time Clock's CMOS RAM; if that value is 0A it vectors through the address stored at 0040:0067, which `ExtTest` aims at this routine.

```
PROC      StartupHandler
CALL      OpenRAM           ; enable writes
INC       [CS:StartupCtr]   ; record another startup
CALL      CloseRAM         ; disable writes
MOV       AX,00580h         ; show r. on the LEDs
CALL      ShowBits
MOV       AL,0FEh           ; tell kbd controller
OUT       KEY_CMD,AL        ; ... to blip the reset lind
@@Stall:  JMP               @@Stall
ENDP      StartupHandler
```

Although our restart handler lives in the Firmware Development Board's nonvolatile RAM, because it *must* exist whenever the CPU uses it, `FDBExt` cannot write its address into the vector at 0040:0067. At least on my systems here, that restart vector changes *after* `FDBExt` exits, although the shutdown reason code does not. That means `FDBExt` must put the address in a spot that `ExtTest` knows about, but somewhere that the BIOS won't wipe out during its startup processing.

Listing 5b shows the code from `ExtTest` that transfers the vector from our nonvolatile RAM to address 0040:0067 and sets the shutdown reason code in the clock's RAM. Later, in response to a keyboard command, `ExtTest` executes this instruction to reset the system using the keyboard controller's reset command:

```
outp(0x64,0xFE);
```

The BIOS then executes the code in Listing 5a, goes through a *second* reset with all the normal power-on tests, and reloads `ExtTest` from diskette. That's all there is to it. Easy, once you know the secret, isn't it?

Now, if anybody asks you about the Worst Hack in PC-dom, you can say you've been there and done that. Tell me if you put it to good use and what you did with the honorary T shirt.

Listing 5b

This code from `ExtTest.C` loads the vector and sets the shutdown reason code into the Real-Time Clock's CMOS RAM. Note that the BIOS restart vector cannot be not in the interrupt table because neither a hardware nor software interrupt invokes it.

```
printf("Changing startup vector from %04x:%04x to %04x:%04x...\n",
      peekw(0x0040,0x0069),peekw(0x0040,0x0067),
      peekw(EXTSEG,STARTUP_SEG),peekw(EXTSEG,STARTUP_OFF));
pokew(0x0040,0x0067,peekw(EXTSEG,STARTUP_OFF));
pokew(0x0040,0x0069,peekw(EXTSEG,STARTUP_SEG));
putstr(" setting shutdown code...\n");
disable();
outp(RTC_ADDR,0x0f);          /* aim at shutdown reason code    */
outp(RTC_DATA,0x0A);          /* ... vector through 0040:0067 */
enable();
putstr(" done\n");
```

Release Notes

The files for this chapter include the modified `LoadExt` boot sector loader that stuffs a BIOS extension into the FDB's nonvolatile storage. You also get `FDBExt` and `ExtTest` to show you how the whole process works. The `ReadMe.txt` file and comments in the source code explain how to load and run the code.

You may find that the interactions between the **NMI** hardware, the BIOS power-on resets, and the BIOS extensions on your target system don't behave quite as I've described. If the sample programs deliver strange results, add some trace outputs and monitor the code's progress through the various stages. A few LEDs that go ON in the right order (or don't go ON at all!) can quickly reveal how your system works.

As always, patience and careful sleuthing will give you far more understanding than applying an In-Circuit Emulator...

9 IDs and LCDs

Some years ago, the PC press pushed hardware serial numbers that could key software to a single, unique PC. The predicted benefits included killing software piracy, increasing profits, reducing prices, and making things right with the world.

Another prediction gone awry.

I suppose I've been a software pirate, at least by some definition. A while back, I had two PCs: a dying Model 80 from which I transferred programs to a new box. If these PCs had software that expected unique hardware serial numbers, I'd be out of luck, even though I had only one set of hands on one keyboard at any one time... and the Model 80 was in such bad shape it could no longer run the programs.

It got worse. The *first* new system didn't quite work, so I made *two* attempts at transferring the files and programs (well, two sets of attempts, to be precise). It's never been clear what the software transfer fee might be for that situation, but I'm sure paying it twice wouldn't have improved my disposition at all.

Nevertheless, in this chapter I'll describe how to give your Firmware Development Board a unique ID based on the Dallas DS2400 Silicon Serial Number chip. If you combine the (trivial) hardware with the BIOS extensions from Chapter 8, you can produce a system that won't even boot with a missing or incorrect serial number, let alone load and run a program from disk.

Just don't try to charge me for your code, OK?

We'll also add a small character LCD panel that displays status messages, without requiring a terminal or video monitor on your PC. If naught else, it can point out a wrong serial number or show everyone the unchanging value of your embedded PC's clock frequency. That last idea makes as least as much sense as the hardware behind your desktop PC's numeric display.

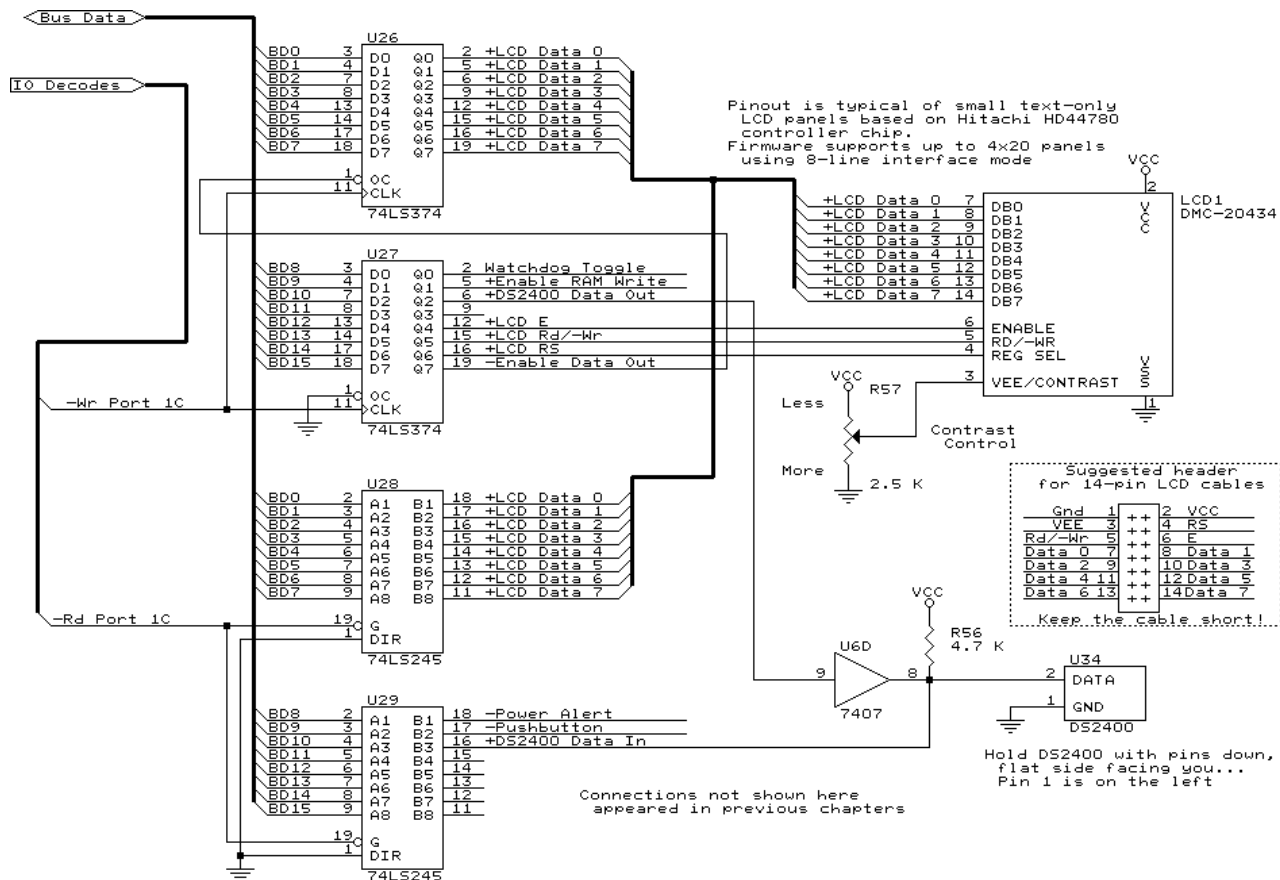
The two devices require completely different interfaces: the DS2400 uses precisely timed bit-serial communications and the LCD runs with a byte-wide parallel port. The Firmware Development Board's 82C54 timer provides timings independent of the CPU speed for the DS2400, while a dollop of firmware timing fixes the LCD.

Simple Circuitry

The DS2400 and character LCD interfaces require a few additional gates and buffers, as you can see in Schematic 1. Once again, I've omitted the ISA bus

Schematic 1

The DS2400 uses a single open collector output bit and one input bit, while the LCD panel requires a full byte of I/O data and several control lines. Firmware produces all of the required pulse timings through the ports, as these devices do not connect directly to the PC's ISA. The support circuitry for the buffer and driver ICs appeared in earlier chapters.



connections and support chips described in previous chapters. Check the Schematics appendix for the complete diagrams.

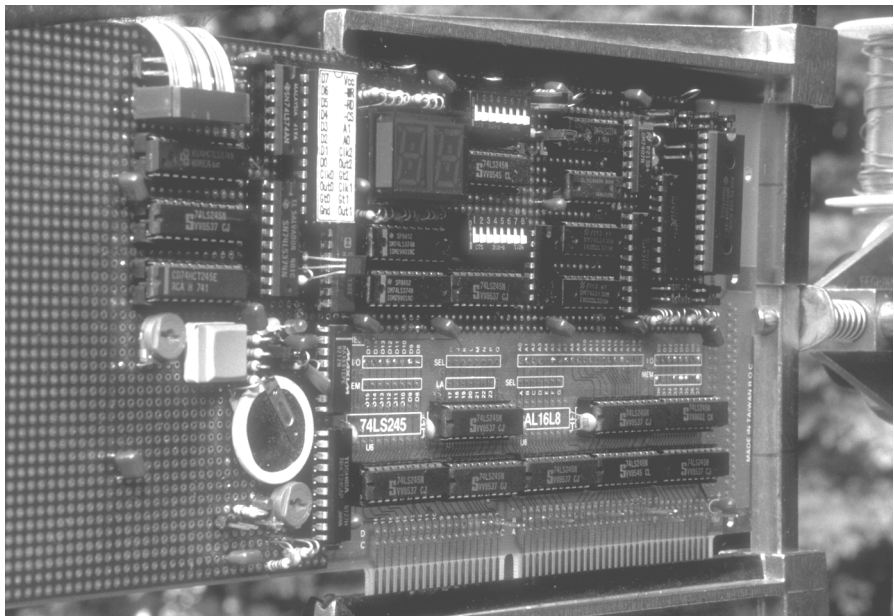
Photo 1 shows the Firmware Development Board after it gained this chapter's hardware. The DS2400 resembles a plastic transistor in a TO-92 case just below the 82C54. The character LCD panel, connected to the ribbon cable snaking from the top, hangs behind the board in this photo. Chapter 7 presented the circuitry for the lithium cell, MAX691, and the clump of discrete parts providing backup power for the static RAM chip.

The DS2400 claims to use a one wire interface. The chip must have a ground connection, of course, but a single signal wire provides bidirectional data and timing. Both the 7407 open-collector driver and the DS2400 can pull the signal wire to the logic low state, while the 4.7 k Ω pullup resistor supplies the logic high state when the driver transistors switch off.

As you might expect from a CMOS part, the DS2400 does not use TTL voltage levels. Its minimum V_{IH} rating of 3.0 V lies well above TTL's minimum 2.4 V

Photo 1

The DS2400 resembles a plastic transistor just below the 82C54. The LCD panel connects to the ribbon cable going off the top of the board. The coin-shaped lithium battery provides backup power for the static RAM chip in the upper-right corner.



V_{OH} . Keep that in mind if you're trying something truly bizarre... something that doesn't involve an open collector or open drain driver with a pullup resistor.

Although the LCD interface uses more wires and even sports an analog contrast control trimpot, it remains fairly simple. In fact, you can probably connect the panel's V_{EE} terminal to ground to get acceptable contrast without the trimpot. If your LCD panel requires a negative bias voltage, connect the pot between the +5 V and -5 V (or -12 V, in extreme cases) supplies. You should increase the pot to about 10 k Ω to reduce its power dissipation. Better yet, always follow your LCD's data sheet recommendations.

A perennial issue surrounds interfacing these little LCD panels directly to a microcontroller bus. The Hitachi HD44780 LCD controller bus interface matches the Motorola 6800 microprocessor and, thus, requires a bit of hocus pocus for an Intel-style ISA bus with entirely different control signals and timings. Rather than confront that problem, I elected to use simple port I/O and be done with it. If you want just an LCD panel on the ISA bus without the rest of the Firmware Development Board, the references in the Bibliography appendix should give you a good head start on your interface design. Pay attention to the timings!

Be careful while you wire up the LCD's LS245 and LS374 data lines. It's quite easy to solder one group "backwards" and create some truly baffling bugs. The LCD test firmware includes a counting sequence that should help pin down that problem, but exercising slightly more care than I did while soldering *my* board should avoid it entirely. Note that the LS374 driving the LCD's data lines must have its **Output Enable** line controlled by the high-order bit from the other LS374, as we must shut it off while our code reads data from the LCD.

With the hardware out of the way, on to the code...

Bidirectional Bit Banging

A laser trimmer personalizes each Dallas Semiconductor DS2400 Silicon Serial Number chip during production so that, unlike ordinary ICs, every one is unique. The data includes an 8-bit type number, a 48-bit serial number, and an 8-bit cyclic redundancy check (CRC) value to verify the data.

Most ICs transmit and receive synchronous data using at least two wires, one of which supplies a clock signal defining when the other has valid data. The DS2400 takes a minimalist approach to its interface by transmitting all information in pulse width modulated blips on a single wire. Using PWM reduces the hardware, while also requiring moderately complex firmware. Fortunately, firmware becomes

cheaper than hardware, after you've figured out how to make it work, and it costs nothing to reproduce. Sounds like a fair tradeoff.

Incidentally, the DS2400 mates nicely with one of the bidirectional I/O pins on an 8051 microcontroller. I used a 7407 open collector driver on the Firmware Development Board, but an 8051 system can get along with just a wire to the pin. You can't get much cheaper than that! (Well, you can, if you embed the serial number in the CPU die itself... but that's another topic entirely.)

Notice that the DS2400 has no power connection. A minuscule internal capacitor charges up while the signal pin is high and powers the CMOS circuitry while the pin is low. The timing specs ensure that the capacitor does not discharge during a transmission, which means you must *carefully* observe the minimum and maximum pulse width limits. Even if the CRC tells you when you get bad data, you should design the interface correctly, rather than depend on happenstance.

Figure 1 outlines a complete transaction: the PC resets the DS2400, sends a command word, and then clocks the type identifier, serial number, and CRC bits from the chip. The initial reset pulse discharges the internal capacitor and clears the DS2400's circuitry to start each transaction from the same, well-defined state.

I've found that the DS2400 sometimes requires *two* reset pulses the first time you access it after the power goes on. From what I can determine, the DS2400 circuitry jams into an invalid state when the power supply voltage rises slowly and requires a complete reset *before* it will respond properly to a normal reset pulse. In effect, you must completely discharge the internal capacitor and give the chip a solid reset. After that, it works just as the data sheet says it does.

In general, the PC will read the serial number only once during its power-on reset sequence. Leaving the signal line high at the end of the transaction eliminates the

Figure 1

Reading a DS2400 requires a reset pulse followed by 72 time slots for the bidirectional data flow. Each half of the reset pulse must be at least 480 μ s long and the data time slots are between 60 and 120 μ s each with a mandatory 1 μ s delay after each bit. A complete transaction thus occupies less than 10 ms.

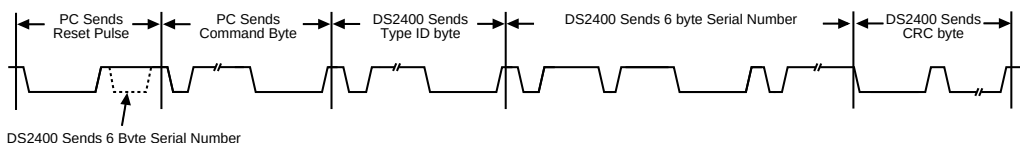


Figure 2

The DS2400 communications protocol requires close cooperation between the PC and the DS2400. The PC determines when each time slot begins and controls the duration of transmitted data, while the DS2400 sets the duration of the pulses it sends in response to the PC's prompting. All times shown in this figure are in microseconds.

Figure 2a

The second half of the reset pulse allows the DS2400 to respond with a "presence detect" pulse: the DS2400 pulls the data line low shortly after the PC raises it. Although it's not completely unambiguous, this behavior indicates that *something* is on the wire.

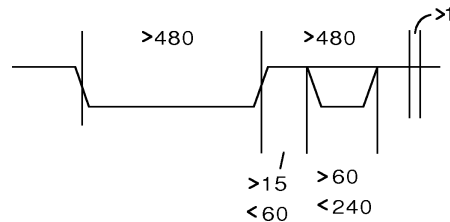


Figure 2b and 2c

The PC pulls the data line low at the start of each transmit time slot. If it is transmitting a 1 bit, it must release the line within 15 μs . The line must remain low for the entire duration of the 60-120 μs time slot to transmit a 0 bit.

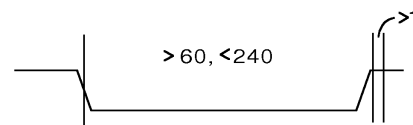
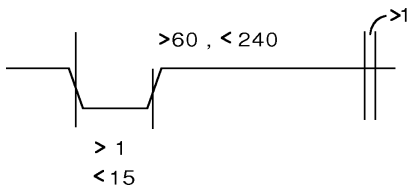
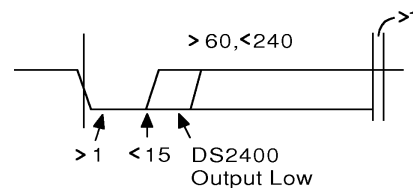
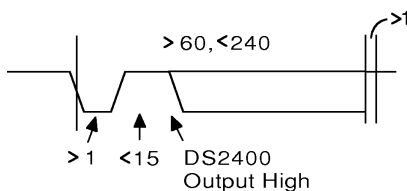


Figure 2d and 2e

The PC also pulls the data line low at the start of each receive time slot, but must release it within a few microseconds to allow the DS2400 to control the pulse width. The DS2400 will release the line immediately to transmit a 1 bit or hold it down for at least 15 μs to transmit a 0 bit. In either case, the PC must sample the DS2400 within 15 μs after the start of the time slot.



few milliwatts dissipated in the pullup resistor, which is surely orders of magnitude more than the DS2400 idle power that's left unspecified in its data sheet.

Figure 2 details the five waveforms used by the DS2400 interface. The reset pulse sports the easiest requirement with only two minimum times: no less than 480 μ s for both the high and low parts. The transmit and receive pulses, however, last only a few microseconds and the PC must accurately measure the received pulses to decide what data it's getting.

When a project's timing specs are denominated in microseconds and you're charged with writing the pulse measurement firmware, get ready for some serious thinking. The first issue must be whether the task is feasible at all. After settling that issue, you can decide the best way to implement it.

Sometimes you really *will* be asked to do the impossible. I recall discussing a project proposal with an engineer who'd been charged with sampling an analog voltage at something like 20 megasamples/sec, then storing 16-bit samples direct to disk through a custom ISA bus board. If you've been paying attention so far, you know that can't *possibly* work (why?). My advice to him: present the ISA bus limits to his managers, then hide behind a bush if the project continued in defiance of reality.

Fortunately, the specs changed when he explained how the ISA bus worked and what those limitations meant to the project. Sometimes you don't get off that lightly, but it's certainly worth the effort.

Hardware Timing

When you use the DS2400 in an 8051 microcontroller system, you can determine pulse widths by simply counting instruction cycles in a timing loop. Not so with embedded PCs! In fact, x86 CPUs encourage programming as an experimental science: figure out how long a loop should last, write it, measure the results, then tweak the loop count to make the answer come out right. With any luck, a few iterations will converge on the desired value.

Your luck will go the other way, though, when you change PC board vendors, move the program from RAM to EPROM, or just twiddle a few BIOS setup values, as the different timings can clobber your precisely tuned loop. Most of the routine tricks in 8051 code just don't apply here. In fact, delay loops fall in the "better to avoid if at all possible" category, although I'll show you a reasonable use for one in the LCD interface code.

I included an 82C54 timer chip on the Firmware Development Board for just such applications: it can measure time intervals with 139 ns resolution, up to a total

Listing 1

The PC and DS2400 communicate by means of pulse width modulated signals. This routine produces low and high pulses without depending on the CPU clock or timing loops. It also enforces a minimum 1 μ s idle time between each pair of pulses. Because read and write signals are identical, this code samples the data line shortly after the rising edge; the caller decides whether the bit is useful or not.

```
int PulseDS2400(LowTime,HighTime)
WORD LowTime;
WORD HighTime;
{
    WORD TestValue;
    int RetValue;

    TestValue;                      /* keep the compiler happy... */
    HighTime;
    LowTime;

asm {
    CLI                            ; need precise timing for first pulse

    MOV    DX,#SYNC_ADDR_A        ; show sync on printer port
    IN     AL,DX
    OR     AL,#$02
    OUT    DX,AL

    MOV     DX,#I8254_BASE_A+6      ; set mode
    MOV     AL,#DS_MODE
    OUT     DX,AL

    MOV     DX,#I8254_BASE_A+(DS_CTR*2) ; set low time
    MOV     AX,6[BP]
    OUT     DX,AL
    XCHG    AH,AL
    OUT     DX,AL

    MOV     DX,#CTLS_ADDR_A        ; set DS2400 data bit low
    MOV     AX,#0
    OUT     DX,AX

    CALL    WaitTimer

    MOV     DX,#I8254_BASE_A+(DS_CTR*2) ; set read delay interval
    MOV     AX,#TIME_RD
    OUT     DX,AL
    XCHG    AH,AL
    OUT     DX,AL

    MOV     DX,#CTLS_ADDR_A        ; set DS2400 data bit high = floating
    MOV     AX,#DS2400OUT_A
    OUT     DX,AX

    CALL    WaitTimer

    MOV     DX,#STAT_ADDR_A        ; read output from DS2400
    IN      AX,DX
    AND     AX,#DS2400IN_A        ; isolate the data bit
    MOV     -2[BP],AX
}
```

Listing continues on next page

Listing continued from previous page

```

MOV     DX,#I8254_BASE_A+(DS_CTR*2)      ; set rest of high time
MOV     AX,4[BP]
SUB     AX,#TIME_RD
ADD     AX,#7                             ; ... plus mandatory recovery time
OUT     DX,AL
XCHG    AH,AL
OUT     DX,AL

STI                                     ; re-enable interrupts

CALL    WaitTimer

JMP     PulseDone

;--- wait for timer interval to expire

WaitTimer EQU *
MOV     DX,#I8254_BASE_A+6                ; latch & read back counter 0 status
MOV     AL,#DS_LATCH
OUT     DX,AL

MOV     DX,#I8254_BASE_A+(DS_CTR*2)      ; read counter status
IN      AL,DX

TEST    AL,#DS_PIN                       ; wait for pin to go high again
JZ      WaitTimer
RET

PulseDone EQU *

MOV     DX,#SYNC_ADDR_A                  ; clear sync pulse
IN      AL,DX
XOR     AL,#$02
OUT     DX,AL
}

return !!RetVal;                          /* return Boolean value */
}

```

duration of 9.1 ms (64 K counts $\times$ 139 ns). Because it uses the 14.3 MHz ISA bus oscillator signal, its time intervals do not depend on the CPU clock frequency or system board hardware and you'll get the same results every time.

Because the read and write pulses resemble each other so closely, I combined their code into the routine shown in Listing 1. `PulseDS2400()`, written using 8086 assembler within a Micro-C wrapper, accesses its input arguments and local variables on the stack.

Although the 82C54 timers can produce hardware interrupts, I use simple software polling to detect the end of each pulse. Remember that hardware interrupt handlers must start by saving the CPU registers and setting up the segment registers. When you are dealing with microsecond pulses, that can take far too long. In fact, you must disable external interrupts before starting the precision timing code.

(Yes, there are sneaky ways to streamline hardware interrupt handlers under very special, carefully controlled, exotic conditions. The key is knowing exactly when the handler will be active, then preloading the registers for it. For an example of this, check my *Firmware Furnace* column in *Circuit Cellar INK* Issue 3. The DS2400 didn't call for such gymnastics, so I could get away with polling. *Whew!*)

The first timing loop meters out the low part of the pulse. If the mainline code calls for a 0 bit, this loop will last for 90 μ s. When the mainline code reads data from the DS2400, the pulse ends after 5 μ s. `PulseDS2400()` also sets a sync bit in the parallel printer port to mark its entry into this code; if that bit never goes low again, you know you have hardware problems on the Firmware Development Board. Plug the LED-and-switch box (from Chapter 1) into your parallel port and watch the firmware in action for some reassurance or diagnostic tracing.

The 82C54 **output** pin goes low during the pulse and returns high at the end. The delay loops load the timer count registers to start the interval, then detect the ending by polling the output status bit. Polling involves writing a latch command to the 82C54 and reading the status register, making it slightly slower than reading a port input bit. Any current PC runs fast enough to add only a few microseconds of delay at the usual ISA bus speeds. If execution time became critical, you could route the timer output bit back through an input port and poll it directly.

After the low pulse ends, the code writes a 1 to the DS2400 output bit, thus allowing the signal line to float high, and sets a 2 μ s delay. When that interval expires, the code samples the DS2400 and records the input value. During a read operation, the DS2400 controls the signal line and the sampled value represents the incoming bit. For a bit written from the PC, the sampled value contains no information and will be ignored by the caller.

Following the sample, the code loads the timer with the rest of the required high time, plus the mandatory 1 μ s idle time, and waits for it to expire. Because the DS2400 output is high, any additional delays have no effect and the code can enable interrupts. The DS2400 spec sets the minimum idle time, but does *not* specify a maximum. Even a protracted delay at this point will not cause a data error, because the capacitor inside the DS2400 remains charged from the input line.

The sample code includes routines that use `PulseDS2400()` to write the 8-bit command word (always 0F for the DS2400) and read back all 64 bits. It's quite straightforward, except for one little, teeny, tiny, essential detail: you must transmit and receive bytes starting with low-order bit. Believe it or not, that critical fact appears nowhere in the DS2400 data sheets on my shelf!

Checking Your ID

After you read the contents of a DS2400, you should verify the data by computing the data's CRC value. You may, if you like, assume that nothing can possibly go wrnog (*oops!*) with such a simple interface, but I recommend checking the CRC just to be sure. The CRC byte's protection includes the Type ID byte, which is always 01 for DS2400, and all six bytes of the serial number.

The DS2400 data sheet presents the CRC algorithm in 8051 assembly language. While just transliterating the code into *x86* assembler certainly looks tempting, remember that the two CPUs set their ALU status flags differently for what seem to be the same instructions. A simple line-for-line conversion will produce the wrong answer. Listing 2 shows how the code I wrote for the CRC calculation takes advantage of the *x86* CPU's 16-bit registers.

Listing 2

The DS2400 includes an 8-bit CRC computed over its Type ID and serial number data. If the byte you compute using this algorithm matches the one read from the DS2400, the odds are pretty good that no errors occurred during the transfer. This code is based on the 8-bit algorithm shown in the DS2400 data sheet, while using 16-bit registers to shuffle the bits. Remember that the 8051 and 80x86 CPUs set their ALU flags differently.

```
int CRCDS2400(pData)
BYTE *pData;
{
    int CRC;

    pData = pData;
    CRC = 0;
    /* keep compiler happy */
    /* clear both bytes */

    asm {
        MOV     AH,#0           ; set up initial CRC
        MOV     SI,4[BP]        ; set up data pointer
        MOV     CX,#7           ; set up byte counter
    OuterLoop EQU *
        PUSH    CX              ; save byte counter
        MOV     CX,#8           ; set up bit counter
        MOV     BL,[SI]         ; fetch data byte
    CRCLoop EQU *
        MOV     AL,BL           ; set up data byte
        XOR     AL,AH           ; combine low bit with old CRC
        ROR     AX,1            ; ... put result into high bit of CRC
        JNC     Zero            ; ... and into carry flag, too
        XOR     AH,#$0C         ; stir in constant (pre-shifted!)
    Zero
        EQU     *
        SHR     BL,1            ; set up next data bit
        LOOP    CRCLoop         ; repeat for all bits
        INC     SI              ; aim at next byte
        POP     CX              ; fetch byte counter
        LOOP    OuterLoop       ; and repeat for each byte
        MOV     -2[BP],AH       ; set up return value
    }

    return CRC;
}
```


Just when you think you've got everything right, though, there's always one more gotcha. I recall a discussion with a group who simply could not get the right CRC value. They used the Official Dallas Algorithm on an 8051 CPU (by Dallas, no less), eliminating any possible translation error. Nevertheless, the CRC bytes they read from their DS2400 chips simply did not match their computed values, no matter how often they checked their code and hardware.

After considerable headscratching and many calls to Dallas' tech support, they found the problem. It seems there was a little glitch in the manufacturing code driving the laser trimmer on the DS2400 production line. It burned the unique serial number into each chip, then computed and burned *the wrong CRC value!*

Talk about hard-to-find bugs!

Needless to say, that was fixed PDQ and all current DS2400 chips work fine. However, should you have a stash of old DS2400s lying around, check them *very* carefully just to be sure... they'll make excellent Show and Tell items.

I've said it before and I'll say it again: hell hath no fury like that of an unjustified assumption.

The **SerNum.C** program reads back the DS2400 data, calculates the expected checksum, and displays everything. To judge from the DS2400s I have on hand, Dallas produced about 400,000 parts before I bought my stash, at least if they're assigning the serial numbers in roughly sequential order.

If your circuit doesn't work the first time, **SerNum** can also write the command byte in a tight loop and scoping the DS2400 data pin should quickly reveal the problem. The interface circuit is simple enough that a solder splash ought to be the extent of your troubles. But, you never can tell...

LCDs Redux

The Firmware Development Board sports a pair of seven-segment LED digits that we've used to report error conditions and display status information. The messages tend to be cryptic, but suffice for our simple purposes. If you must dress your system up for company, though, you probably want a few lines of legible text output. The small LCD panels you see on microcontroller projects can be equally handy on embedded PCs. Making them work is a simple matter of firmware.

I reviewed the operating principles of small LCD character panels in *Circuit Cellar INK* Issue 8. Refer back to that column for details of how they work and what all

the interface lines mean. For now, I'll concentrate on the specifics of the Firmware Development Board's hardware, rather than cover that ground once again.

The common denominator of all these LCD panels lies in the Hitachi HD44780 LCD controller that provides their (admittedly limited) intelligence. Most of the small character LCD panels you'll find use this controller or one similar to it, but you should get the data sheets for your panel just in case you catch an oddball.

The LCDTest.C program can handle any display based on the HD44780 controller, up to the chip's 80-character maximum size. I preset the code for 4 rows of 20 characters, using a pair of constants that define the number of visible rows and columns. You can tweak those two lines, recompile, and have your new display up and running, pretty much regardless of its size or shape.

As shown in Schematic 1, the LCD panel interface uses port I/O, rather than a direct PC ISA bus connection. Although a bit-banged port interface runs much slower than a direct bus hookup, the firmware can still update the entire display faster than those liquid crystals can respond. That's fast enough.

Listing 3 shows the few lines of code that write a byte to the display. The LCD's **Enable** line latches data and commands into the controller, so the code must write

Listing 3

This routine writes a single byte to the character LCD panel. The Mode parameter determines whether the byte is a character for the display buffer or a command to the LCD controller. After writing data to the port, the code calls BlipEnable to pulse the LCD's Enable input high and low to strobe the byte into the LCD controller.

```
LCDSendByte(Data,Mode)
WORD Data;
WORD Mode;
{
WORD PortValue;

    if (!(Mode & SENDFORCE)) {          /* want to force without wait?      */
        LCDWaitBusy();
    }

    PortValue = (Mode & SENDDATA) ? LCD_RS : 0;    /* cmd/data, -wr, enabled */
    PortValue |= Data;                          /* combine data             */
    outpw(CTLS_ADDR,PortValue);

    BlipEnable(PortValue);

    PortValue |= LCD_RW;                    /* disable -wr line        */
    outpw(CTLS_ADDR,PortValue);
}
```

the 16-bit port three times. First it sets up the data and control lines with **Enable** low, then sets **Enable** high, and, finally, lowers **Enable** to finish the operation.

Because we cannot read the outgoing data back into the CPU, the code stores it in the `PortValue` variable between writes. This is a classic hardware *vs.* software tradeoff: implementing the hardware with TTL gates means that additional functions require more chips and occupy more board space. Had I built the Firmware Development Board using PALs or gate arrays, the additional hardware would be essentially free. Think of this as practice for real life.

Maintaining a copy of the most recent output data is a simple trick that suffices for now. I'll describe the real world problems that arise from this sort of situation later on in this chapter.

Listing 4 presents `LCDReadByte()`, which reads status and data from the LCD controller. Two other routines call this function to check the controller's **Busy** status flag and to read the display buffer during vertical scrolling. Although both functions can be simulated, with some difficulty, in firmware, I used Steve Ciarcia's favorite programming language: Solder. Sometimes, if there's just room for just one more chip, you must choose which one provides the most benefit. Ah, tradeoffs.

Despite my tirade about software delay loops, you'll find one buried in the heart of `LCDReadByte()`, just after the code raises the **Enable** line. Some LCD panels I've used require more time than expected to drive data back to the Firmware Development Board. The value I settled on creates a 150 μ s pause on an 80 MHz '486DX2, about four times the delay my worst panel expects. If you get erratic vertical scrolling on a faster CPU, you know what the problem is and how to fix it.

This may be related to the ribbon cable layout that comes naturally to these panels, although the glitch doesn't appear on my scope (not that glitches ever do, of course). You might want to experiment with different cable layouts to see if bracketing the **Enable** wire or the data lines with pairs of quiet ground lines helps. Trust me on this: if you haven't used firmware to fix hardware before, this is a good time to get started...

Why not use one of the 82C54 timers on the Firmware Development Board? Well, if we had the services of a realtime operating system at our disposal, I'd have no qualms about invoking a programmed delay. Unfortunately, we don't, and I have plans for this timer later on that preclude dedicating it to a character LCD. In real life, you might also find the overhead involved in setting up a 150 μ s delay through an operating system might take longer than the delay itself. Measure first, then decide what to do.

Listing 4

This routine reads a byte from the LCD controller; the Mode parameter determines whether it comes from the display buffer or the controller logic. The loop in the asm{} section compensates for delays caused by cable capacitance and must be tuned for each system... there is no need for a precise measurement, just a minimum delay.

```
int LCDReadByte(Mode)
WORD Mode;
{
    WORD Data;
    WORD PortValue;

    outp(SYNC_ADDR,inp(SYNC_ADDR) | 0x02);          /* scope sync          */

    PortValue = (SENDDATA == Mode) ? LCD_RS : 0;    /* read data or addr?    */
    PortValue |= LCD_RW + LCD_NODRV;               /* LCD read and disable drivers */
    outpw(CTLS_ADDR,PortValue);

    PortValue |= LCD_E;                             /* strobe data from LCD  */
    outpw(CTLS_ADDR,PortValue);

asm {
    MOV    CX,#RD_DELAY                ; delay to allow data settling time
?lp1 LOOP ?lp1
}

    Data = inpw(STAT_ADDR) & 0x00FF; /* fetch and isolate LCD data */
    PortValue &= ~LCD_E;              /* remove LCD strobe          */
    outpw(CTLS_ADDR,PortValue);

    outp(SYNC_ADDR,inp(SYNC_ADDR) & ~0x02);      /* sync off                  */

    return Data;
}
```

The remaining LCD code adds cursor positioning, string output, simple scrolling and some control character processing. It's straightforward code that you should review to see how it's done. The `LCDTest.C` program provides several test loops with scope trigger outputs on the parallel port, as well as a routine that copies incoming serial characters to the LCD, so you can check out new displays by hand.

Extra credit project: you can add ANSI cursor positioning support fairly easily. Peek ahead to Chapter 16 for details and some sample code. You can then send the same text to a small character LCD or a big bitmapped panel with similar results.

Shared Bits

The `PortValue` variable lets `LCDSendByte()` toggle the LCD **Enable** line while holding all the other bits steady. Unfortunately, `LCDSendByte()` has no way to preserve the other bits previously written to that port, because it can't read back

the current port contents. That has no effect on the demonstration program, but in real life you may have a killer problem.

For example, the watchdog timer routines I discussed in Chapter 7 send a new bit to the watchdog about six times a second. That interrupt handler cannot read back the existing bits, either, so if your code writes a byte to the LCD controller at the same time as the watchdog interrupt routine, your bits get clobbered. When the LCD write completes, it returns the favor by wiping the watchdog bit.

This problem can be easy to solve in a single program, at least after you recognize it. Simply put, all your routines must refer to a single global `PortValue` variable that holds the current output port state at all times. If an interrupt handler can change the port, as is usually the case, you must protect your mainline code by enabling and disabling interrupts around the sections where you update `PortValue`.

Unless you disable interrupts, you can experience some devilishly subtle bugs. For example, when your mainline code loads `PortValue` into a register to change the bits, as most C compilers will, your interrupt handler can fetch the *old* `PortValue` from the variable, change a few bits, and write it back. The mainline code will then destroy the handler's bits when it writes its *new* version of `PortValue` from the register into the variable, atop whatever the handler just stored.

It's worth working through a few examples on paper to convince yourself that you're in a lot of trouble. Look for those instants in time when `PortValue` differs from the actual port bits: *that's* when the interrupt handler will strike. Make sure only one routine can alter `PortValue` at any time, always keep the variable in sync with the hardware port, and disable interrupts around the critical regions.

You'll see these problems again in subsequent chapters, but I should at least point out the land mine right now. The solutions to this problem are both well known and well documented in the multitasking literature. You've surely read about semaphores, critical regions, and threads of execution by now. If not, well, those same issues will crop up when you write multithreaded Windows programs.

Release Notes

The code includes two Micro-C programs: `SerNum.C` for the DS2400 circuits and `LCDTest.C` for character LCD panels. You can combine the essential routines into a single program, but pay attention to the shared bits problem. When you stir in the watchdog code from Chapter 7, it gets still more complex.

In the next chapter, we'll see how to put C code into a BIOS extension, so you can write a program that can tell you you're not authorized to use your own PC...

10 Booting C from ROM

A foolish consistency is the hobgoblin of little minds...

Ralph Waldo Emerson

That may be an aphorism suitable for any occasion. In this case, back in Chapter 6, I opined that writing BIOS extensions in C probably wasn't practical, citing the Firmware Development Board's limited memory address space. Emerson would smile, as I'll now show you how to do just that.

In point of fact, if you need just a smidge of code you may as well use C and be done with it. After all, it doesn't matter if you have 31 KB of C or 3 KB of tightly written, carefully tuned assembler... that chip has 32 KB of space available. Now that you know how to build battery backed RAM, write code that can send diagnostics through serial ports, and understand the general mechanics of BIOS extensions written in assembly language, we can pull it all together.

In this chapter, we'll explore the gory details of turning a Micro-C program into a BIOS extension. The complexities along the way may seem daunting, but the end result will help you get your own firmware working.

At least, for small values of firmware...

Basic BIOS Booting

As we've seen, every PC goes through much the same ritual immediately after a hardware reset. It first checks the hardware, finds and initializes any BIOS extensions, and finally boots a program from disk or diskette. Each BIOS extension must hook an interrupt vector if it wants to be part of the action after the BIOS regains control and continues booting.

The exact system state isn't predictable when your extension gets control, because any previous extension can add features or change the BIOS setup in nearly any way. In a given system, of course, the same thing happens (or *should* happen) during each boot, but you cannot assume that all systems respond the same way. For example, you'd think that the BIOS would set up the serial ports before invoking the extensions, but, at least on one of my systems, that's not the case.

After calling all the extensions, the BIOS finishes its own initialization and issues an `INT 19h`. Under normal circumstances, that interrupt vector points back into the BIOS code to a bootstrap loader routine responsible for booting from either

diskette or a hard disk. Any BIOS extension, however, can hook `INT 19h` and regain control just before BIOS accesses the disk. That extension may check a serial number, verify a password, or skip the disk boot entirely by booting from ROM.

If the original `INT 19h` BIOS routine eventually gets control in a system with no bootable disks, it will issue `INT 18h` after failing to read the disks. In the Original IBM PC, that interrupt fired up the built-in Cassette BASIC interpreter (remember Cassette BASIC?), but other manufacturers don't have rights to that IBM proprietary code. Most of them simply display a message and await a three finger salute on the Ctrl-Alt-Del keys. Your BIOS extension can hook `INT 18h` to regain control after the disk boot fails, allowing you to start one routine from disk or run another from the extension ROM without booting from disk.

According to the references, the system will be completely ready for action when the BIOS invokes the `INT 19h` and, if needed, `INT 18h` interrupts. Your extension can, therefore, take control of a perfectly functional PC without handling any of the grubby setup work. The sample code for this chapter has a “your code goes here” note at the appropriate spot so you can complexicate it as required.

For what it's worth, Cassette BASIC lives on. My ancient Model 80 (nigh on to a decade old now) proudly displayed its Cassette BASIC screen when I disconnected its disk drive controller to track down the problem I mentioned in Chapter 9. The IBM Tech Reference manual admits that Cassette BASIC might not be too useful, perhaps because Model 80 systems lack a cassette port...

But, well, BASIC is still there!

Modeling Memory

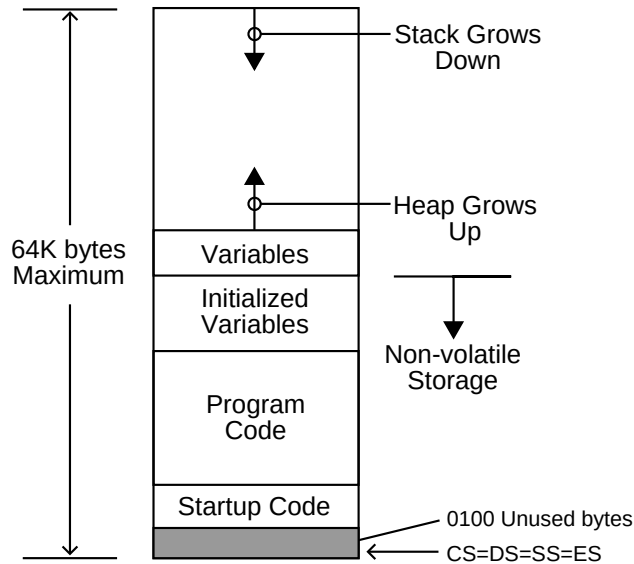
Thus far, we have used Micro-C's `Tiny` memory model for our embedded programs. Unfortunately, `Tiny` model won't suffice for BIOS extension code. The reason, as with most things PC-ish, involves the segmented memory inherent in `x86` real mode programming.

`Tiny` memory model puts all of the C program's code and data into a single 64 KB segment, as shown in Figure 1. When the program begins, all of the CPU's segment registers contain the paragraph address of that segment. An assembly language function or inline code within a C function can refer to memory outside that segment by reloading the segment registers, but it must restore them before returning control to the C code.

The program's startup code begins at offset 0100, followed by the compiled C code and library functions. All of the initialized data, including strings and “constant”

Figure 1

Micro-C's Tiny memory model puts all of the program's addressable storage in a single segment that may be up to 64 KB long. The program code and initialized variables must either reside in nonvolatile storage or be copied from diskette by a loader program. The variables, heap, and stack must be in RAM, for obvious reasons. All of the CPU's segment registers point to the start of the segment.



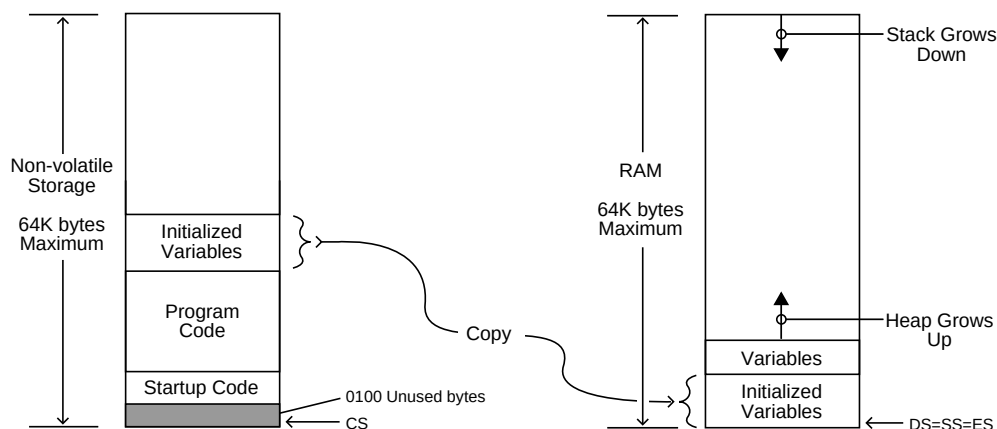
variables, come next. Uninitialized variables follow, with the heap beginning just after the last variable. The stack grows downward from the end of the segment, allowing the heap and stack together to use whatever space remains beyond the program and data.

Recall that the first 256 bytes of the segment remain as a legacy of the DOS `COM` program structure. While this format does let us produce and manipulate the programs with ordinary DOS utilities and compilers, it wastes those 256 bytes. If you have written the program loader to set up the registers properly, as we have, then you can tweak Micro-C's startup code to begin at any offset you'd like, because it need not produce DOS compatible files.

A BIOS extension with a combined code and data segment runs into trouble, because the FDB's battery backed RAM circuitry includes write protection. Storing variables in write protected RAM would be bad enough, but running a CPU with a

Figure 2

Micro-C's Small memory model allows up to 64 KB of program code and initialized variables, which, as with Tiny model, must be in nonvolatile storage addressed by the CS register. The startup code or loader copies the initialized variables into a separate RAM segment addressed by the DS, SS, and ES registers. After that, the `main()` program may use them during execution without any special attention.



write protected stack simply won't work. The assembler code in Chapter 8 controlled the RAM -WE circuitry around each data access, a completely impractical process for a C program. And, of course, the C code must have enough stack space in ordinary RAM for each function's automatically allocated variables.

The most straightforward solution uses Micro-C's *Small* memory model, which puts up to 64 KB of program code and the starting values of any initialized variables in one segment. A separate data segment, also up to 64 KB, holds the actual initialized variables, uninitialized variables, the heap, and the stack. The code segment can be write protected, while the variables reside in ordinary read-write RAM. Figure 2 shows the *Small* model segment layout.

How this works in our situation should be evident. The code segment can live in the FDB's battery backed RAM, while the data segment resides in system RAM below the 640 KB line. The C startup code, in addition to its other duties, must reserve the program's data segment, copy the initialized variables into it, and load the segment registers before calling the `main()` C routine.

It's a simple matter of firmware...

Incidentally, because the startup code copies the initialized variables from nonvolatile storage into RAM, the C program can treat them as ordinary variables with nonzero starting values. Micro-C's *Tiny* model locks the variables into read-only, nonvolatile storage unless you boot from diskette into RAM, as we have been doing all along. In case you hadn't guessed, variables placed in read-only, nonvolatile storage behave a whole lot like constants!

Micro-C includes a *ROM* memory model that directly supports this type of *Small* model programming by copying the initialized variables from ROM into RAM. All we must do is tweak that code into acting like a BIOS extension.

There are other ways to solve the problem, of course. I picked *Small* model because it worked out quite neatly and exploited an interesting Micro-C feature, but another approach may be more suited for your projects. As always, take what you read here and make your own improvements.

For example, you could duplicate the diskette boot loader's function in the BIOS extension. On each reset, your extension would copy the entire C program, initialized variables and all, from the FDB's battery backed RAM into system RAM, then execute it using *Tiny* memory model just as before. The RAM can (and should) remain write protected during the whole operation. If your code fits into 32 KB or you build a paged RAM interface (*ugh*), this can be a perfectly viable way to make it work.

You might also modify the Firmware Development Board's address decoding circuitry to protect the lower 16 KB of battery backed RAM and allow writes into the upper 16 KB, but it certainly seems a shame to leave the entire 640 KB of system RAM unused, doesn't it? Micro-C's startup code includes support for such split memory segments, which are much more common in minimal 8088 systems than full-fledged embedded PCs. However, if you replace the FDB's discrete-logic address decoding with a PAL, you can take advantage of a small, self contained block of RAM that's not directly available to other PC programs.

Sequenced Startup

The C startup code must accept control from the BIOS, adapt Micro-C's runtime conventions to the BIOS extension entry requirements, run the C program, and then return control to the BIOS after the extension finishes its setup. Coupled with the segment shuffling required by *ROM* model code, the C startup code holds some interesting tricks.

The extension must, of course, begin with the signature required by the BIOS scan, so Listing 1a resembles the code presented in Chapter 8. The diskette boot loader

computes the checksum on the contents of the disk file and the values in the battery backed RAM beyond the end of the program. As before, the checksum byte in the extension header must be zero in both the source code and disk file. After the loader returns, the extension is ready to run on the next boot.

The extension bails out without further action if it finds the FDB's pushbutton switch closed, thus allowing a normal boot without loading the extension. Recall

Listing 1a

The ordinary Micro-C startup routine for the ROM memory model assumes it has complete control of the system, so you must make several changes to adapt it to use as a BIOS extension. Pressing the pushbutton on the Firmware Development Board disables this extension and allows a normal boot sequence.

```
*-----
* BIOS Extension header
*
*          ORG          $0100          standard COM offset
*
*          DB           $55            0000 signature
*          DB           $AA            0001 ... second byte
*          DB           6              0002 code + init data size
*          JMP          <BootEntry      0003 jump around checksum
*          DB           $00            0005 ... which goes here
*
*-----
* Now proceed with boot code
*
*--- if pushbutton is down, exit without doing much
*
BootEntry   EQU          *
            MOV          DX,#STAT_ADDR
            IN           AX,DX
            TEST         AX,#PUSHBUTTON
            JNZ          <Continue      nonzero means not pushed
*
            MOV          AX,#~$0101    show -- to track our path
            MOV          DX,#LED_ADDR
            OUT           DX,AX
*
            RETF          ; return to normal BIOS boot
*
*--- pushbutton is up, so it is OK to continue
*   adjust CS so our offsets are correct
*
Continue    EQU          *
            MOV          AX,#~$0100    show single - here
            MOV          DX,#LED_ADDR
            OUT           DX,AX
*
            MOV          AX,CS
            SUB          AX,#$0010     add 0100 to offsets!
            PUSH         AX            simulate a FAR CALL
            MOV          AX,#LdRegs
            PUSH         AX
            RETF          ; ... to load the new values
LdRegs      EQU          *
```

Listing 1b

Because the Micro-C extension code resides in write protected, battery backed RAM, the data segment must use system RAM to let the program's variables work normally. This section of the startup routine reserves 32 KB just below the infamous 640 KB barrier by adjusting the BIOS Ram Size word at 0040:0013. It stores a pointer to that segment in the battery backed RAM to allow the INT 19h handler to find its data.

```
*--- locate the data segment at the top 32K of contiguous RAM
*   the BIOS RAM size value at 40:13 has units of 1K bytes
*
*       MOV     AX,$057E           show r0 on LEDs to mark entry
*       CALL    ShowBits
*
*       PUSH    DS                 save BIOS seg regs
*       PUSH    ES                 ... on the BIOS stack
*
*       MOV     AX,$0040           fetch memory size
*       MOV     ES,AX
*       MOV     BX,$0013
*       SUB     ES:[BX],#>$0020    reserve 32K (1K byte units)
*       MOV     AX,ES:[BX]         convert RAM size to seg reg
*       MOV     CL,#6              (10 bits left, 4 bits right)
*       SHL     AX,CL
*       MOV     ES,AX              target seg for MOVSB and STOSB
*
*--- copy the initialized data to the data segment
*
*       MOV     AX,$0530           ; r1
*       CALL    ShowBits
*
*       MOV     AX,CS              aim at this segment
*       MOV     DS,AX
*       MOV     CX,#?Isize         pick up length
*       MOV     SI,#?Ivars         start of init data in code seg
*       MOV     DI,#0              copied to start of data seg
*       JCXZ    ?nocopy            skip if zero (or 64 K) bytes long
*
*       REP     MOVSB              from DS:SI to ES:DI
*
*?nocopy      EQU      *
*
*--- clear the uninitialized data while we have the pointers at hand
*   length is the difference between ?heap and ?temp
*
*       MOV     AX,$056D           r2
*       CALL    ShowBits
*
*       MOV     CX,#?heap+1-?temp  length of uninit data + heap top
*       MOV     AL,#0              get a zero (or a test value!)
*
*       REP     STOSB
*
*--- aim DS at the new data segment
*
*       MOV     AX,ES
*       MOV     DS,AX
*
*
```

Listing continues on next page

Listing continued from previous page

```
*--- save the BIOS SS:SP registers
*   ... begin using our stack in the new data segment
*
*           MOV     ?BIOS_SP,SP
*           MOV     ?BIOS_SS,SS
*
*           MOV     SS,AX           load stack pointer
*           MOV     SP,#STK_TOP
*
*--- now save pointer to our data in the battery-backed RAM
*
*           MOV     AX,#NV_WENABLE   allow writing into RAM
*           MOV     DX,#CTLS_ADDR
*           OUT     DX,AX
*
*           MOV     AX,#NV_SEGMENT   aim at RAM segment
*           MOV     ES,AX
*           MOV     BX,#STACKTOPOFF   ... stack offset
*           MOV     ES:[BX],SP        ... write offset
*           MOV     ES:2[BX],DS       ... and data segment
*
*           MOV     AX,#0            prohibit writing into RAM
*           MOV     DX,#CTLS_ADDR
*           OUT     DX,AX
*
*           MOV     AX,DS
*           MOV     ES,AX            restore ES
```

that we rewired the keyboard lock switch in parallel with that pushbutton to provide the bypass function without opening the case (assuming, of course, that your PC both has a case *and* that it remains closed). If you omit the button test, you have no way to disable a malfunctioning BIOS extension other than popping the RAM chip out of the board. Trust me, it won't look professional.

The next step adjusts the CS and IP registers to match the C compiler's ROM-model assumptions. The FDB's RAM begins at C800:0000, so the BIOS sets the initial CS:IP to C800:0003. The simulated FAR CALL reloads CS with C7F0, then adds 0100 to IP, making the startup code's offsets correct. Refer back to Chapter 8 for more details on this trickery.

The code also must reserve some RAM for the data segment. During power-on, the BIOS self test routine records the system RAM size in kilobytes in the word at 0040:0013. Because this value excludes memory above the 640 KB line, even a 128 MB target system reports a RAM size of 0280 hex or 640 decimal. If your system sports an old CGA board and some specialized hardware, its system RAM can extend up to address B800:0000 for 736 KB of contiguous memory. As I mentioned in Chapter 6, that trick won't work with monochrome or VGA/SVGA boards, or in embedded systems that run without the DOS support required for memory manager programs found in desktop PCs.

The most convenient data segment location, at least for our purposes, lies at the top of system memory. The code in Listing 1b subtracts 32 KB from the nominal RAM size. That means any subsequent BIOS extensions or the embedded program booted from diskette will find only 608 KB of RAM. Admittedly, 32 KB far exceeds what we need for this chapter, but you can see the general principle at work.

The next chunk of code copies the initialized variables from their write protected location in the FDB RAM to system RAM. The Micro-C compiler defines a work variable, `?temp`, as the first uninitialized variable. Because the initialized variables begin at offset 0000 in the data segment, the length of the block of initialized variables is equal to the address of `?temp`. This length will be zero when there are no initialized variables, so we must check the CX register before starting the `REP MOVSB` operation to avoid copying 64 KB of variables that don't exist.

Up to this point, the startup code used whatever stack the BIOS passed to it in `SS:SP`. The C program must run with its own stack to guarantee enough space for nested function calls and interrupts, and that stack should be in a known location. The code stores the BIOS's `SS:SP` in the `?BIOS_SS` and `?BIOS_SP` variables, then reloads `SS:SP` with a pointer to the end of the reserved RAM segment. Before returning, the code restores `SS:SP` to keep the BIOS happy.

Figure 2 shows that the stack grows downward in the data segment, so the initial stack pointer aims at the top of the segment. Because this extension uses a 32 KB chunk of RAM, its initial SP should be 7FFE to allow an unused word at the end of the stack. If your code requires a 64 KB data segment, you *must* use FFFE, because the initial SP must be at least two bytes below the end of the segment (four bytes, in 32-bit protected mode) to avoid the dreaded Stack Fault error trap when the CPU pops the final value from the stack.

The next step may seem peculiar, but, because our BIOS extension might not be either the first or last extension, we cannot hardcode our data segment's location into the startup code. For example, if an earlier extension claimed 10 KB, our 32 KB segment would begin at 9580:0000 rather than 9800:0000 (work it out!). But, obviously, we can't store a pointer *to* the data segment *in* the data segment, as subsequent code wouldn't know where to look for it.

IBM's BIOS architects reserved the interrupt vectors between `INT 60h` and `INT 67h` for user functions. In principle, our startup code could store the pointer to our segment in the `INT 60h` vector. When the extension begins execution, it could fetch the `INT 60h` vector to find its data. In fact, I used this technique in an earlier version of this code.

Unfortunately, while the books say those interrupts are reserved for user functions, the BIOS in some systems clears their contents after initializing the BIOS extensions and before invoking `INT 19h`. That means any data our BIOS extension stores in that spot simply vanishes. Perhaps those BIOS designers felt that no user code could possibly begin running before that point... they certainly never had us in mind, did they?

The only other spot where we can reliably store information is in the battery backed RAM holding our code on the Firmware Development Board. The last few lines in Listing 1b disable the RAM write protection, store the initial `SS:SP`, and enable the write protection. Because the data segment holds both the variables and the stack, that single pointer provides the initial values for both `DS` and `SS`.

The downside of this technique is that something may clobber the RAM during the few instructions that must execute with writing enabled. Also, you must use RAM, rather than ROM or EPROM, to allow on-the-fly updates as the extension executes. Weigh the risk of overwriting the RAM against the benefits of flexible data allocation to see if this technique makes sense in your application.

The remainder of the startup routine is almost anticlimactic, as you can see from Listing 1c. It initializes the heap by writing a zero at `?heap`, then calls `main()`.

Listing 1c

After preparing the variables and segment registers, the startup code initializes the Micro-C heap and calls the `main()` function. Unlike most embedded C programs, `main()` must return to allow the BIOS to continue its boot sequence. This listing shows how the code restores the segment registers and returns to the BIOS.

```
*--- now, for the main event...
*
*          MOV      AX,#$0579      r3
*          CALL     ShowBits
*
*          MOV      <?heap,#0      set up empty heap
*          CALL     main            execute main program
*
*--- restore regs and return
*
*          MOV      SS,?BIOS_SS     aim back at BIOS stack
*          MOV      SP,?BIOS_SP
*
*          POP      ES              from BIOS stack
*          POP      DS
*
*          MOV      AX,#$0505      show rr to indicate the end
*          CALL     ShowBits
*
*          RETF                    ; and continue with boot
*---
*** Remainder of Micro-C startup code is unchanged
```

When `main()` returns, the startup code restores the BIOS `SS:SP` values, pops the saved registers from the BIOS stack, and returns to the BIOS through the obligatory `FAR RET`.

If your system has a video board, monitor, and keyboard, the BIOS will make them ready before it invokes your BIOS extension. However, it may or may not prepare optional equipment such as the serial and parallel printer ports. On a stock PC, your BIOS extension can display its status on the screen and read ordinary keyboard input. I'm taking a minimalist approach in the sample code, but don't let that discourage you too much.

Many embedded BIOS extensions will control hardware similar to the Firmware Development Board's LCD panel, watchdog, or serial number. When your extension gains control, you can set your hardware up. Don't go overboard; do just the bare minimum and be done with it. You can report error messages on the standard PC screen, or, preferably, on some LED digits, a heartbeat LED, an LCD panel, or any other hardware that you control.

Listing 2 shows a rudimentary `main()` function that saves the `INT 19h` Bootstrap Loader vector and installs a pointer to a customized `INT 19h` handler. Rather than create a separate storage location for the old vector, the code simply tucks it into the `INT 61h` vector. This is a time-honored PC technique that allows you to invoke the old handler through a simple `INT 61h`, rather than a convoluted indirect `FAR CALL` through a pointer in the data segment. If you must restore the registers before calling the old handler, you'll appreciate why this is a Good Thing.

Listing 2

The `main()` function shown here moves the `INT 19h` vector to `INT 61h` so we can use it later, then aims `INT 19h` at our own interrupt handler. It sends a tracking output to the parallel port, but cannot send a serial message because the BIOS has yet to identify the hardware ports. Because this function is called during the BIOS extension scan, it must exit to allow the PC to continue booting.

```
main() {
    GetVector(0x19,&Int19Seg,&Int19Off);          /* move old handler */
    SetVector(0x61,Int19Seg,Int19Off);           /* ... to new interrupt */
    outpw(CTLS_ADDR,NV_WENABLE);                 /* save in NV RAM, also */
    pokew(NV_SEGMENT,OLDINT19OFF,Int19Off);
    pokew(NV_SEGMENT,OLDINT19SEG,Int19Seg);
    outpw(CTLS_ADDR,0);

    SetVector(0x19,GetCodeSeg(),INT_S_ENTRY(Int19Handler));

    outp(SYNC_ADDR,0x01);

    return;
}
```


It also stores the `INT_19h` vector in the Firmware Development Board's battery backed RAM to protect it from BIOSes that clear the user interrupts. If you *know* your target system's BIOS clears those vectors or you expect your code to run on many different machines, simply don't store your vectors in system RAM. The code we'll see later checks the `INT_61h` vector against the value in the RAM, re-installs it if the BIOS changed it, and tells you what your BIOS did.

Note that, unlike all the other Micro-C programs we've used so far, this `main()` function *must* exit to allow the remaining BIOS initialization to continue. Because system setup will not be complete while your code executes, the C program should do no more than absolutely necessary. In particular, the BIOS probably hasn't loaded the serial port addresses at 0040:0000, which means `main()` can't send a cheerful "Hello, world!" message through the BIOS serial functions.

Getting the Boot

By intercepting `INT_19h`, the C code regains control just before the BIOS tries to boot from the disk or diskette. I covered such interrupt handlers in Chapter 7, but there are enough differences in the `Small` and `ROM` models to warrant another look.

Listing 3

This macro wrapper uses the Firmware Development Board's battery-backed RAM to locate the handler's data segment and stack. The `stk` argument determines where the handler's stack begins within the stack segment, thus allowing several simultaneously active handlers to use the same macro. The data locations in RAM must match those used by the program's startup code!

```
#define STACKTOPOFF    (32512+0)
#define DATASEGMENT  (32512+2)
#define OLDINT19OFF    (32512+4)
#define OLDINT19SEG    (32512+6)
#define OLDINT18OFF    (32512+8)
#define OLDINT18SEG    (32512+10)

#define INT_S_PROLOG 64          /* size of prologue code      */
#define INT_S_ENTRY(fn) (&fn-INT_S_PROLOG) /* start of prologue */

#define _SPACE_
#define INT_SMALL(fn,stk) asm {
*--- save bystanders on incoming stack
fn PUSH    AX
   PUSH    BX
   PUSH    CX
   PUSH    DX
   PUSH    SI
   PUSH    DI
   PUSH    ES
   PUSH    DS
```

Listing continues on next page

Listing continued from previous page

```

*-- fetch ptr from battery-backed RAM to find DS/ES/SS and stack  \
MOV     BX,#NV_SEGMENT_A                                         \|
MOV     ES,BX                                                     \|
MOV     BX,#STACKTOPOFF                                          \|
MOV     CX,ES:[BX]                                                \|
SUB     CX,#stk                                                    \|
MOV     AX,SS                                                      \|
MOV     BX,ES:2[BX]                                               \|
MOV     SS,BX                                                     \|
XCHG    SP,CX                                                      \|
PUSH    CX                                                         \|
PUSH    AX                                                         \|
MOV     DS,BX                                                      \|
MOV     ES,BX                                                      \|
*-- save our compiler temp on our stack                          \|
MOV     AX,?temp                                                  \|
PUSH    AX                                                         \|
*-- invoke the Micro-C handler                                  \|
CALL    fn+INT_S_PROLOG                                           \|
*-- restore compiler temp from our stack                         \|
POP     AX                                                         \|
MOV     _SPACE_?temp,AX                                           \|
*-- restore pointer to incoming stack                           \|
POP     AX                                                         \|
POP     CX                                                         \|
MOV     SS,AX                                                      \|
MOV     SP,CX                                                      \|
POP     DS                                                         \|
POP     ES                                                         \|
POP     DI                                                         \|
POP     SI                                                         \|
POP     DX                                                         \|
POP     CX                                                         \|
POP     BX                                                         \|
POP     AX                                                         \|
*-- bypass Micro-C return                                       \|
IRET                                                                \|
}
#undef _SPACE_

```

Listing 3 shows the macro wrapper for a ROM-model Micro-C interrupt handler that knows about our battery backed RAM. It saves a few of the caller's registers on the interrupted routine's stack, recovers the handler's `SS:SP` from the RAM and sets up that stack, saves the rest of the incoming registers on the new stack, then invokes the interrupt handler. When the handler returns, the wrapper undoes all that before finishing the process with an `IRET`.

If your code can generate nested interrupts, such as hardware interrupts during a software interrupt, you must place each handler's stack in a different part of the data segment. The second macro argument simplifies this process: the wrapper code subtracts `stk` from the stack pointer saved in RAM. You can easily allocate separate chunks relative to the original stack top with this trick.

I could use the same stack for both INT 19h and INT 18h, because my INT 18h handler executes after the INT 19h handler returns. In general, however, you must be more cautious with your stacks. In particular, make sure you allocate a different stack area for each hardware interrupt handler and reserve enough space for each stack so that they cannot possibly overwrite each other.

Your INT 19h handler can take over the PC or continue booting as you see fit. The handler in Listing 4 hooks INT 18h, displays a message and waits for a serial character from the host PC before continuing. Depending on what you type, it will either invoke the original INT 19h handler or issue an INT 18h directly. If there are no diskettes in the system, the BIOS INT 19h routine passes control to a customized INT 18h handler.

That INT 18h routine displays a message, then enters an endless loop updating a counter on the FDB's LED display. Obviously, you can be a lot more clever than

Listing 4

The BIOS INT 19h handler normally boots a program from disk, but this handler gives you a choice. Depending on the serial input character it will either boot using the BIOS handler (which the startup code moved to INT 61h) or pass control directly to the INT 18h boot failure handler. You can insert an entire application program either here or in the INT 18h handler, depending on how much hardware setup you expect the BIOS to do.

```
INT_SMALL(Int19,0) Int19Handler() {
int Option;

    serinit(9600,1);
    enable();                               /* allow timer ticks      */
    putch(FORMFEED);
    putstr("Embedded PC's ISA Bus Chapter 10 -- Ed Nisley\n");
    putstr("Micro-C BIOS Extension Demo\n\n");
    putstr("Interrupt 19 (pre-boot) handler\n");
    outp(SYNC_ADDR,0x02);

/*--- show that we have got proper memory addressing      */
    StkSeg = peekw(NV_SEGMENT,DATASEGMENT);
    StkOff = peekw(NV_SEGMENT,STACKTOPOFF);
    printf(" Pointer in NV RAM to top of stack is %04x:%04x\n",
        StkSeg,StkOff);

/*--- capture Int 18 vector to route control to our handler */
    GetVector(0x18,&Int18Seg,&Int18Off);
    printf(" Capturing Int 18 vector, was %04x:%04x\n",
        Int18Seg,Int18Off);
    SetVector(0x18,GetCodeSeg(),INT_S_ENTRY(Int18Handler));
    outpw(CTLS_ADDR,NV_WENABLE);
    pokew(NV_SEGMENT,OLDINT18OFF,Int18Off);
    pokew(NV_SEGMENT,OLDINT18SEG,Int18Seg);
    outpw(CTLS_ADDR,0);
}
```

Listing continues on next page

Listing continued from previous page

```
/*--- see if Int 61 survived the BIOS boot process */
GetVector(0x61,&Int61Seg,&Int61Off);
if (Int61Seg != peekw(NV_SEGMENT,OLDINT19SEG)) {
   _putstr("*** BIOS overwrote our info in the Int 61 vector!\n");
    printf(" we wrote %04x:%04x\n",
        peekw(NV_SEGMENT,OLDINT19SEG),
        peekw(NV_SEGMENT,OLDINT19OFF));
    printf(" is now %04x:%04x\n",Int61Seg,Int61Off);
    SetVector(0x61,peekw(NV_SEGMENT,OLDINT19SEG),
        peekw(NV_SEGMENT,OLDINT19OFF));
   _putstr(" Restored to proper value!\n");
}

/*--- now run the command loop */
while (1) {
   _putstr(" Press Enter to reboot or Esc to invoke Int 18h: ");
    Option = getch();
    putch('\n');
    switch (Option) {
        case '\n':
            printf(" invoking old Int 19 through Int 61 -> %04x:%04x...\n",
                Int19Seg,Int19Off);
asm {
    INT $61
}
            break;
        case ESC:
            printf(" invoking our Int 18 handler...\n");
asm {
    INT $18
}
            break;
    }
}
```

that. Remember, however, that this is the last chance you get to affect the system. I don't know what the BIOS will do if the INT 18h handler returns, as Cassette BASIC offered no way back to the BIOS boot code.

To summarize, the C startup code and `main()` function cooperate to form a BIOS extension that gets called during the boot sequence. The `main()` code may hook the INT 19h and INT 18h interrupts to regain control before and after the disk boot. Those routines are software interrupt handlers, rather than BIOS extensions, and should use the C interrupt macro wrapper to save and restore the registers.

Your code can hook other software or hardware interrupt vectors to replace, modify, or extend standard BIOS services. For example, you could redirect calls to the BIOS INT 10h video routines to your character LCD panel routines... that panel is a mite cramped, but your embedded applications could use the LCD or a normal video display with no changes. Get the picture?

Release Notes

Because BIOS extensions require a specialized C startup file that isn't useful for normal C code, I created a separate library file called `BIOSExt.LIB` that includes the startup code module `8086RLXR.ASM`. You can modify `BIOSExt.LIB` to include specialized library files or exclude Micro-C library files as needed.

The macro wrapper shown in Listing 3 is in the `firmdev.h` file in this chapter's subdirectory. Unlike the `Tiny` model wrapper, it requires enough setup and specialized tweaks that I didn't want to put it in the `Micro-C` subdirectory.

See the source code and `ReadMe.txt` files for more information.

The Micro-C `CC86` Command Coordinator cannot specify a library other than the standard ones, so I've been using my `MCComp.BAT` file to create BIOS extensions. You can simply replace the `8086RLPR.ASM` file with (renamed) `8086RLXR.ASM` and use `ROM` model if you find that more convenient.

You must have the FDB's battery backed RAM and write protection circuitry installed to use this code, because it updates the RAM contents during the BIOS boot sequence.

I *still* think 32 KB doesn't provide enough room for a complete PC program, but now you have a framework for small, C language BIOS extensions and diskless programs. Just don't get carried away and smash into that 32 KB limit at, oh, say, 90% of the way through your next project.

11 Beyond Small

Does your application suffer from Creeping Featuritis? The symptoms include a **ReadMe** file bigger than the program, multiple layers of nested functions, and a tendency to display odd behavior should you dare use any new features. You'll know it when you see it...

One of the (few) nice things about 8031 microcontroller projects is how the CPU's limited address space puts an upper limit on complexity. Apart from obvious perversions, like paging 2 MB of code into an 8031 (yes, it's been done), you can only do so much with a single-chip microcontroller. This chapter may herald the end of such innocence, as embedded PCs, those computers in controllers' clothing, have no such limitations.

Thus far, we've used Dunfield Development Systems' Micro-C, a compiler admirably suited to embedded projects. The startup code occupies only a few lines, it has simple and well controlled memory requirements, and, best of all, it works! You also get a diskette of video, serial, joystick, and other PC-flavored functions that help jumpstart your embedded PC projects.

But a recurring question runs, "So, what about our Borland and Microsoft C compilers?" Suddenly, life becomes more complicated.

Unlike Micro-C, which was designed for small controller projects, the Borland and Microsoft C compilers grew up with the PC. By necessity, they include the assortment of memory models required for the 80x86's segmented, real-mode architecture. Whether those complex convolutions are a Good Thing or not remains open to debate, but, if you must crawl into the PC's hidden spaces, these compilers help you get there.

Those compilers have a catch: perforce, mainline C/C++ compilers for the PC produce code that must run under DOS, Windows, or worse. Loading their **EXE** output files isn't easy, their C startup code goes on for pages, and operating system functions litter their startup and runtime library files. All that makes producing a standalone program for a DOS-less PC a task for the stoutest of heart.

Although several authors have tackled the subject of directly embedding DOS programs, I believe that remains an unrewarding and endless task. Each new version of the compiler ripples changes through your existing applications, making you spend time and effort keeping up with the Borlands and Microsofts of the world. Basically, all you *really* want is to sprinkle magic dust on your **EXE** file and turn it into an embedded PC file.

In this chapter, we'll cover some of the issues involved in producing DOS-less projects using a DOS compiler, just so you know what you're missing with Micro-C. I'll also introduce **Locate**, a commercial product from Paradigm Systems, that helps solves this problem. There are similar products available from other companies, so check the Sources appendix, peruse the ads in your favorite magazines, and troll the Web for the current state of the art.

Paradigm allowed me to include a customized version of their **TDREM** program on the diskette accompanying this book. If you've built a Firmware Development Board with battery backed RAM or (E)EPROM, you can debug your Borland C code using Turbo Debugger. The **ReadMe.txt** file for this chapter includes the ugly details and relevant versions.

Along the Mainline

After years of working on large-system software, a friend recently began writing PC code. As he put it, "C is C, but what the heck is all this about **NEAR** and **FAR** pointers? When do I need **LARGE** memory model? What's going on here anyway? Who's responsible for this outrage?"

As long as you don't need more than 64 KB of memory, PC programming can be easy, because the familiar **COM** file format with its single segment does nearly everything you need. As you've seen, a **COM** file holds a binary image of the program's code and initialized data. Running the program requires nothing more than copying the disk file into RAM, setting the segment registers, and passing control to the first instruction at offset 0100 in the code segment.

But, when you must create complex programs or manipulate megabytes of data in real mode, the Intel $\times 86$ Segmented Memory Monster rears its ugly head. Because the simple **COM** file, left over from days when 64 KB was a *lot* of RAM, has no way to specify multiple program or data segments, Microsoft's DOS designers came up with the **EXE** file. It's not just a different file format, but a completely different way to treat executable program files.

Here's the gotcha: programs with multiple code or data segments *must* come in **EXE**-format files. Even something as simple as a **SMALL** model program with code in one segment and data in another requires the full **EXE** file structure. Worse yet, because of the additional segment information, **EXE** files for anything other than **TINY** model programs cannot be converted directly into **COM** files.

Figure 1 shows the fields in a DOS **EXE** file header. The Code and Data Segments field near the bottom contains the actual instructions and variables created from your source code, similar to the stuff that's normally in the **COM** file. The remaining

fields define the overhead information required to load those segments into RAM, adjust them for the actual memory addresses, set up the CPU registers, and start executing the program.

The compiler and linker produce the `EXE` file, but they cannot assign the final segment addresses. That step occurs when DOS allocates memory and sets the load address just before reading the file from disk. The Relocation Table entries point to

Figure 1

Programs stored as `EXE` files have a header with additional information used to load and run the program. The exact format depends on the DOS version, but older files will generally work unchanged on newer systems. In this context, a paragraph is 16 bytes of storage that starts on a 16-byte boundary.

Offset	Description
0000	Signature: MZ
0002	File length modulo 512 (remainder)
0004	File size in 512-byte units (rounded up)
0006	Number of relocation table entries
0008	Header size in paragraphs
000A	Minimum RAM needed in paragraphs
000C	Maximum RAM needed in paragraphs
000E	Location of stack segment in paragraph
0010	Initial SP value
0012	Checksum (not used!)
0014	Initial IP value
0016	Location of code in paragraphs
0018	Offset of relocation table in bytes
001A	Overlay number
001C	Optional reserved space
...	Relocation table
...	Optional reserved space
...	Code and data segments
...	Stack segment

the values within the code and data that require some adjustment before execution. In effect, the DOS `EXE` file loader performs the final link step that puts a relocatable program at an absolute address.

For example, the C statement

```
++Variable;
```

might compile into the assembly language instruction:

```
INC Variable
```

The `DS` register, implicit in that `INC` instruction's operation, must hold `Variable`'s segment address. But the `DS` value depends on precisely where `Variable` wound up when DOS loaded the file. If your program accesses more than 64 KB of data, each variable has a corresponding `DS` and the compiler must insert code that loads the proper value into `DS` before accessing each one. All that fussing with segment registers helps make `LARGE` model programs run slower than you might expect.

Unlike DOS programs, many of our embedded programs are definitely *not* relocatable. For example, if our BIOS extensions in Chapter 10 didn't reside at precisely `C800:0000`, they simply wouldn't work. The `EXE` file format's flexibility becomes a liability in our situation.

Therefore, we must find a way to assign absolute segment addresses and convert the `EXE` file into an EPROM image or a binary disk file. That's part of what `Locate` does for a living, but there's more to the story.

It's worth noting that some protected mode operating systems can load and run code without making any relocation changes. The trick lies in their use of what's called `FLAT` memory model, which takes advantage of the memory management hardware in '386 and higher CPUs. In effect, a `FLAT` model program uses 32-bit addresses in a single segment that can span up to 4 GB. We can't use this in real mode, but imagine what a `COM` program would look like without its 64 KB addressing limitation.

Jumping the Mainline Track

By now, you should be familiar with the C startup code that gets control before the first line of your `main()` function. Micro-C's startup code requires only a few lines of assembler and we have modified it to do some interesting things. With a little tweaking, we even turned a more-or-less ordinary Micro-C program into a workable BIOS extension.

The Borland or Microsoft C startup code is considerably more elaborate than anything we've seen so far. In fact, the Borland C++ 3.1 startup file spans 746 lines and handles everything from clearing variables and initializing the file system to parsing the DOS environment strings. Quite a bit of that effort does us no good at all (no DOS means no DOS environment strings) and some of it can be actively harmful in an embedded environment.

And then there's Borland C++ 5.0, even bigger and more complex... which is probably obsolete by the time you read this. Get the point?

The startup code need do nothing more than the program requires. Because we won't use the DOS file system, all that code can simply Go Away. Similarly, with no DOS environment or command line information, eliminating that part of the startup routine makes perfect sense.

You cannot discard other sections quite so blithely. The startup code calls many DOS functions to adjust the program's stack and heap, return excess storage to the operating system, save and install interrupt vectors, and even display error messages. You must replace each of these functions with code that performs the same operations without invoking DOS: running a C program without a heap, for example, just isn't possible. Even if *you* don't call DOS services to allocate a heap, *someone* must do the job before your program reaches its first `malloc()`.

On top of this, the startup code handles such minutiae as defining the proper segment order for the linker, calling C/C++ entry and exit functions, and so forth. Some of these routines tie tightly into the way the compiler generates code, allowing well intentioned (but misinformed) changes to wreak subtle havoc on your program. The requirements of C++, which I won't attempt to cover, add yet another layer of complexity beyond what you expect for a C program.

The good news: **Locate** includes a sample C/C++ startup code file that you can use unchanged. Comparing the startup files from Borland and Paradigm tells me I'm glad I didn't have to figure out those changes on my own. Try it yourself and see.

The last vestige of DOS dependence lurks in the runtime library. Many C library functions use DOS calls, but, unless you have the library source code and plenty of time, you cannot tell which routines pose a problem. Because the library invokes DOS functions through software interrupts, the linker cannot identify nonexistent functions. Your code simply crashes when an `INT 21h` branches into the weeds.

The **Locate** package includes a program that creates a new set of libraries by deleting all the DOS-dependent routines from the standard Borland C runtime library files. If the linker complains about "missing functions" and "unresolved

externals” in your program when you use those libraries, well, even if you must write some additional code, that should be considerably better than debugging a mysterious failure and *then* writing the missing code.

Now we have all the pieces in place: absolute address assignment, a tailored startup file, and a purified runtime library. The rest of the task should be a matter of turning the crank...

Embedding a Sieve

Listing 1 shows Paradigm's version of the venerable **Sieve** benchmark program. I've added a line that displays the loop counter on the Firmware Development Board's LEDs and increased the test limit to 30,000 integers. I did not actually verify that the program produces the right answers... in this situation, it really doesn't matter.

Listing 1

The Paradigm Locate disk includes the good old Sieve benchmark program. I've modified it to display the iteration number in raw binary on the Firmware Development Board's LED display so you can watch it run. The key point: this code doesn't look any different than any other C program, because the startup code and Locate handle all the hocus pocus under the covers.

```
#define TRUE 1
#define FALSE 0
#define SIZE 30000

#include <dos.h>
char flags[ SIZE + 1 ];

void main(void)
{
    unsigned int i, k, count, loops;
    loops = 0;

    /* Run the Sieve forever */
    for ( ; ; ) {

        /* Perform the initialization */
        count = 0 ;
        for (i = 0; i <= SIZE; i++)
            flags[i] = TRUE ;

        /* Run the Sieve */
        for (i = 2; i <= SIZE; i++) {
            if (flags[i]) {
                /* Cancel out all multiples of this prime */
                for (k = i + i; k <= SIZE; k += i)
                    flags[k] = FALSE ;
                count++ ;
            }
        }
        outport(0x031e,~loops++);          // display count on FDB LED segments
    }
}
```

The code looks just like any other C program you've ever seen, because all the magic that embeds it into a DOS-less PC occurs elsewhere. The `#include <dos.h>` line might seem worrisome, but that file simply defines the `outport` macro. Fear not, your program won't invoke DOS just for that.

Compiling and linking the file proceeds normally, with one exception: you must link the object file with Paradigm's special startup code and runtime libraries. The linker produces, as usual, an ordinary `Sieve.EXE` file, which the `MakeFile` renames to `Sieve.ROM`. That step prevents you from accidentally running the program, should you type `Sieve` at the DOS command prompt.

In our case, the `outport` macro becomes an inline routine and `Sieve` doesn't use any library functions at all. Incidentally, because the `Locate` installation program creates the DOS-less libraries without changing the original Borland libraries, you can continue to build ordinary DOS and Windows programs as usual.

`Locate` translates `Sieve.ROM` into `Sieve.BIN` by assigning absolute memory addresses to all the segments. `Locate` can produce several different output formats, each with an array of suboptions. Among the choices: Intel `HEX`, extended `HEX`, Tektronix `HEX`, a special Absolute `EXE`, Intel's OMF86 files, and raw binary.

For some embedded systems projects, you may burn a `HEX` file into an EPROM or load a binary file into an emulator. For our work, we can use a slightly modified version of the diskette boot loader for the new files. The only change involves setting the program's initial `CS:IP` value so that our loader properly passes control to the program.

The loader we've been using so far follows the DOS `COM` file convention by passing control to the program at offset 0100 in the segment indicated by `CS`. All `COM` files expect that starting value, because they have no way to specify any other register contents. Although `EXE` file headers include the program's initial `IP` value (see Figure 1), the pure binary file produced by `Locate` no longer has that header information and, thus, gives us no clue as to where to start it up.

The C startup code expects to receive control at label `_startup`, which could be anywhere in the code segment. Paradigm's startup code positions that label and its corresponding instruction as the first byte of the file. By linking that file first, it appears at the beginning of the code segment. Then, we just tell `Locate` to put the code segment at the start of the output file. Because `_startup` marks the first instruction in the startup code, it becomes the first byte in the final binary image.

However, with `_startup` at that position, `IP` must be 0000 rather than 0100. Our familiar `COM` file loader from Chapter 1 can't handle `Locate`'s converted `EXE` files!

Fortunately, we control the disk boot code. The files for this chapter include a slightly modified `BootSect` program that loads the file at address 10000, rather than 11000, then jumps to the first byte with `CS:IP` set to 1000:0000. Even though the Paradigm startup code handles all the other segment register initializations, I left the COM-style setup in place to give the registers a known value on entry to the startup code.

You might think twiddling the startup code to make the offset of the first instruction equal 0100, perhaps by moving other instructions ahead of it, would let you use the old loader. Remember that the COM loader preset `CS:IP` to address the first byte of the file at `CS:0100`, not `CS:0000`. The loader will transfer control to the first byte, even if it's not the program's entry point. Both the startup code and its loader must agree on where the first instruction resides.

You could, I suppose, add a few lines to the startup routine that catches the boot loader entry at the first byte, then reloads `CS` and `IP`. Rather than do that, I figure

Listing 2

The information in this configuration file tells `Locate` how to convert `Sieve.ROM` from EXE to binary. The `MAP` statements define the system address space, while the `CLASS` statements set the segment starting addresses. The `DUP` statement creates a copy of the initialized variables in the code segment, so the startup routine can copy them to the actual data segment in RAM.

```
hexfile binary offset=10000h size=8           // boot loader bin-file
listfile segments                             // output listing file

map 0x00000 to 0x005ff as reserved             // Int vectors BIOS data
map 0x0c600 to 0x0ffff as reserved             // Unused
map 0x10000 to 0x1ffff as rdonly               // 64 K application boot load area
map 0x20000 to 0x9ffff as rdwr                 // Available for app
map 0xA0000 to 0xfffff as reserved             // Video, BIOS ROMS, etc

dup      DATA ROMDATA                        // Copy initialized data

class    CODE = 0x1000                        // Boot loader area
class    DATA = 0x2000                       // Data area

order    DATA                                \ // RAM organization
         BSS BSEND
         STACK

order    CODE                                \ // ROM organization
         FAR_DATA ENDFAR_DATA
         INITDATA EXITDATA
         ROMDATA ENDROMDATA

output   CODE                                \ // Output classes
         FAR_DATA ENDFAR_DATA
         INITDATA EXITDATA
         ROMDATA ENDROMDATA
```

you'll use either Micro-C's `COM` files or Paradigm's converted `EXE` files, but not both at once. Just use the loader that matches your situation and be done with it. Make *absolutely* sure you mark the diskettes while you're experimenting, though, because half of the possible combinations simply won't work.

With that as background, Listing 2 shows the segment address assignments defined in the `Sieve.CFG` file. `Locate` has far more options than I have room to describe here, but you should get a feel for the program's flexibility.

The `HEXFILE` statement specifies an 8 KB binary output file containing the instructions and data starting at absolute address 10000, the load point set by our diskette boot loader. You may use additional `HEXFILE` statements to create multiple output files during the same pass. `Locate` will warn you if the file includes no information at all, which can catch stupid addressing mistakes.

The `MAP` statements define how the program will use the CPU's address space. `Locate` verifies that all of the segments seem appropriate for their addresses. For example, you should not put a code segment into a read/write area that allows code overwrites. Remember that `Locate` and `MAP` cannot actually write protect your system's RAM. They simply document your intentions and verify that no other segments encroach on the code during the file conversion process.

The `MAP` statement provides more benefit with the `TDREM` debugger I'll discuss shortly. Because we load our code from disk into RAM, the `rdonly` (read-only) sections will be just as volatile as any other RAM. However, the debugger can detect changes within those areas and report problems.

The `DUP` statement creates a copy of the data segment in the `ROMDATA` part of the code segment. Paradigm's C startup code automatically copies `ROMDATA` from the code segment, where the loader put it from diskette, into the data segment. That's all it takes to get initialized variables working.

The two `CLASS` statements specify the starting addresses for code and data segments. The name `CLASS` comes about because the compiler and linker can combine several different program segments into a single physical memory segment, thus treating the original segments as a class. This statement sets the address of the first segment in the resulting class. If your application has multiple classes, you must include a `CLASS` statement for each one.

The `ORDER` statements specify how to combine individual segments into an overall class that a `CLASS` statement will nail at a specific address. This corresponds (roughly) with the way the linker handles groups of segments, but requires manual intervention to get it right. The `Locate` documentation goes into more detail on this

issue. Suffice it to say that Paradigm's sample files give you a good idea of what to do in several different situations.

Despite the seeming complexity, once you get a configuration file set up, you probably won't tweak it again until you change your program's segment structure or the target system's memory map. In fact, you can probably use the same `CFG` file for most of your embedded PC projects with little change.

Actually running `Locate` requires nothing more than another `MakeFile` line, which should take even less thought than figuring out your compiler or linker switches. While I was inside the `MakeFile`, I added a few commands that copy the resulting `BIN` file to the boot diskette. Two keystrokes in my host's DOS window now rebuild the whole application and set up the diskette for the target machine.

Although I compiled `Sieve.c` with the `Small` memory model, `Locate` and the modified runtime libraries support all models except `Tiny`. Because our custom diskette boot loader can only handle binary files up to 64 KB, you must modify the loader for truly large programs. You can certainly use my source code to get started.

Borland's `Tiny` memory model can generate `COM` files without running `Locate`, just as we did with Micro-C. Remember to eliminate all DOS functions from the startup code, link your files with a DOS-less library, and boot the result with our original (`IP = 0100`) `COM` file diskette loader.

`Locate` supports Microsoft C++ as well as the Borland compilers. I don't have any examples because, back when I developed these programs, their C compiler didn't run under OS/2 without lots of twiddling that I wasn't interested in doing. Now that I'm using Windows 95, I suppose I could switch, but it does seem like a lot of effort to get essentially the same result.

That leaves us with one question: "What if the program doesn't work?" Finding program errors is always the most difficult part of development, particularly with burn-and-crash debugging, but we now have a full-featured debugger at our disposal.

Debugging by Wire

Creeping Featuritis afflicts all PC programs. Simple command-line compilers accrete features, sprout program development utilities, and gather debuggers on their disks. More recently, these conglomerations gelled into Integrated Development Environments delivered on a CD-ROM or two. If you look hard, you can actually find the command-line compiler hidden in a subdirectory and, perhaps, described in a footnote.

Creeping Featuritis means more than never buying another diskette or figuring out what to do with a stack of obsolete CD-ROMs. Somewhere along the way, we lost sight of the fact that an IDE spawning a debugger to load a nontrivial program compiled with every debugging option turned ON can soak up all the RAM in a system. In some cases, the debugging infrastructure doesn't leave enough room for the program we intended to debug in the first place. Thus began the arms race to exploit expanded memory, extended memory, any memory except the precious 640 KB down there in DOS territory.

Borland's Turbo Debugger designers, in either a stroke of genius or an admission of defeat, included a *remote debugging* mode that runs the debugger's user interface on an entirely separate machine from the target program. A serial cable or network connection eliminates the requirement for exotic RAM mapping, video display sharing, and mouse handoffs.

Whenever Turbo Debugger refers to a memory location, an I/O port, or a CPU register, it sends a message from the host system across the cable to the target system. A debugging kernel in the target handles the request and returns the results over the same cable to the host. Because the kernel doesn't interact with the target's keyboard, video display, or mouse, it can be both small and simple. Most of the conflicts between the target program and the debugger simply Go Away when the two programs run on different systems.

Windows has, to some extent, relieved the DOS memory shortage on '386 and higher CPUs by putting the PC's entire multimegabyte RAM address space to good use. Unfortunately, because high capacity debuggers exact a significant toll in system overhead and, of course, our target program don't run under Windows, you will still find a remote debugger your best choice.

The folks at Paradigm (and others, as well) figured out that remote debugging could be the solution for debugging embedded x86 systems. With a bit of programming magic, they can fool Turbo Debugger into thinking it has an entire PC on the other end of the wire, even if the target system lacks disks, DOS, or even a BIOS to support the embedded program.

Adapting TDREM to the Firmware Development Board was straightforward, as the Paradigm code can handle most hardware configurations with just a few EQU or #define twiddles. I picked the simplest target interface: polled communication through the COM1 serial port.

Because TDREM should get control before booting the target program from disk, I turned it into a BIOS extension using code similar to that shown in Chapters 8 and 10. The extension code hooks INT 19h and returns to the BIOS. When the BIOS

finishes initializing everything, it invokes `INT 18h`, which starts the `TDREM` remote interface. My code shows its progress on the Firmware Development Board LEDs, concluding with a cheery `td` immediately before diving into Paradigm's Turbo Debugger support routines.

Installing the extension uses the same BIOS extension loader program we used earlier. It copies `TDREM` from diskette into the FDB's battery backed RAM and computes the checksum required for a valid BIOS extension. Remember to hold down the FDB's pushbutton while booting the loader to install the extension.

After that, running the debugger requires just booting the target system. My prologue code checks the FDB's pushbutton before starting the extension. With the button pressed, control returns to the BIOS without capturing `INT 19h` and the standard diskette boot sequence occurs. The LEDs show a pair of dashes to indicate that the extension did not install itself.

With the pushbutton up, the extension installs itself, the FDB LEDs show `td`, and you can start Turbo Debugger on your host system. For the DOS version of TD, use the command line:

```
TD -RS3
```

on the host to specify a 38.4 kb/s data rate. I found that `TDREM`'s default 115 kb/s rate doesn't work on my target system, probably because the serial bytes arrive too fast for the software polling loop. Obviously, an interrupt-driven interface would solve that problem, but fitting it into the target system involved more complexity than I thought justified for this part of the project.

The `TDREM` kernel always tells the debugger that it has an outdated copy of the program because, in fact, it doesn't *have* a copy. Turbo Debugger then asks whether you want to send the "newer" version... you should always answer Yes.

Now you can choose either Micro-C or a mainline C compiler, depending on your project's goals and finances. Either compiler will work for a broad range of projects, so there shouldn't be much holding you back. Go for it!

The Second-Worst Hack

While we're on the topic of debuggers, here's an interesting application of a deliberate error. Back in Chapter 8, I described the Worst Hack in PC-dom: using the keyboard controller to reset an 80286 CPU and bail out of protected mode. While doing some research for another project, I came across a tidbit that probably qualifies as the Second-Worst Hack in PC-dom.

Suppose you're writing a protected-mode operating system (can you spell OS/2?) for '286 systems and realize that the Worst Hack's speed (or lack thereof) will definitely clobber overall system performance. What to do?

Easy: crash the system!

Starting with the 80286, Intel CPUs detect severe errors and shut down, rather than continuing with unpredictable results. For example, when `SP = 0001` a `PUSH` instruction shuts down the CPU, instead of wrapping `SP` to `FFFF` and clobbering RAM outside the stack area (as the 8086 and 8088 did). The CPU sends a specific status signal that tells the rest of the system what happened, then waits for a hardware reset or an **NMI**. It does not execute any further instructions, process external interrupts, or do anything else besides wait quietly.

Rather than leave the system stalled forever, the Original IBM PC AT system board (and, thus, all subsequent PCs) included a shutdown detection circuit that causes a hardware reset. The CPU pops out of reset in real mode, the BIOS checks the Real-Time Clock's battery backed RAM for the shutdown reason code, and vectors to the appropriate routine. Apart from the fact that the reset comes from a different circuit, it works just as you saw in Chapter 8.

Because the reset happens at hardware rather than software (or even firmware) speeds, the whole sequence takes a tiny fraction of a millisecond. That's enough faster than the Worst Hack to make it worthwhile. It's even faster than those hyperthyroid keyboard controllers with hardwired command bypasses, too.

How do you force a CPU shutdown? The CPU invokes a Double Fault interrupt handler that gets control when two protection violations occur on a single instruction. If the Double Fault handler *also* causes a violation, the CPU shuts down. It seems the OS/2 designers set up a deliberate *triple* fault to bail out of '286 protected mode faster than the standard keyboard controller command.

Although I don't know the exact method they used, you could mark a segment as Not Present, then invalidate both the Segment Not Present and Double Fault interrupt gates. You disable external interrupts and the **NMI** input, aim a segment register at the missing segment, and fetch a byte. The fetch triggers a Segment Not Present interrupt through `INT 0Bh`, where the invalid gate causes a Double Fault interrupt (`INT 08h`), and the invalid Double Fault gate slam dunks a triple fault.

Thud!

Release Notes

The **Sieve** binary files require the modified **BootSect** diskette loaders from this chapter. You'll find the source and binary files for all four diskette sizes in the subdirectory: as always, make sure you pick the appropriate one for your system.

Because this chapter's version of Paradigm's **TDREM** remote driver for Turbo Debugger runs as a BIOS extension, copy it to a diskette along with the appropriate **LoadExt** extension loader from Chapter 8. Boot that diskette in your target system while holding the pushbutton down to stuff **TDREM** into the Firmware Development Board's battery backed RAM. The next boot, with the button up, will display **td** on the LEDs when the target is ready for Turbo Debugger.

TDREM will use polled operation through COM1 at 38.4 kb/s. Remember to start Turbo Debugger on your host PC using the **-RS3** switch to get the proper serial data rate. All the usual features should be available, except breaking into a running program. That requires a fully interrupt-driven configuration that I'll leave as an exercise for you. Don't forget that watchdog timer...

I assume you already have the Borland or Microsoft compilers running on your system. Follow the directions accompanying Paradigm's programs to properly install **Locate** and modify your runtime libraries. Remember that few, if any, embedded PC programs are upwardly compatible with new compilers; consult the **ReadMe.txt** file for the versions you'll find on the diskette with this book.

Note that the software for this book does *not* include Paradigm's **Locate** program or their other utilities and libraries. You may use the executable programs found on the diskette, but you must have the complete Paradigm **Locate** package to modify and recompile them.

If your finances won't stretch around a commercial **Locate** program, you should spend some time searching the Internet for shareware equivalents. Several magazine Web sites have large collections of links: *Circuit Cellar INK* at www.circellar.com, *C/C++ User's Journal* at www.cuj.com, and *Embedded Systems Programming* at www.embedded.com. Dunfield Development Systems at www.dunfield.com will conduct you to several useful programs for low-level embedded work. You'll find other pointers in the Sources appendix.

12 Lots'a Dots

You probably have a tempting embedded PC application for those big LCD panels you see in the ads, but the phrase “external controller required” translated into “Well, maybe, one of these days...” Figuring out how to use a bitmapped graphic LCD made the project hard enough that you never got around to it, right?

I'd been thinking about them for a while, too, for a series of firmware projects that can take advantage of a big display that *isn't* a CRT, *doesn't* require a serial port, and *must* function with a VGA board also in the PC. As you'll see in the next few chapters, a handful of logic gates, plus a dollop of firmware, can convert an LCD panel into a bitmapped graphic and alphanumeric display. Best of all, you won't need a specialized LSI widget between you and the dots.

Overall, the Graphic LCD Interface is about as complex as the rest of the Firmware Development Board taken together. It summarizes most of the techniques we've used so far and illustrates how you can combine firmware and hardware to implement a complete design.

I'll start by describing how graphic LCD panels work, present some condensed specs for several typical panels, and go over the interface's hardware block diagram. You'll be in a better position to understand why the Graphic LCD Interface works the way it does when you see what all the signals do after they get to the panel. If only our real-world projects had such bounded properties, we'd have fewer failures!

In the next two chapters, you'll see the interface hardware schematics and the test code that checks it out on your system. Because of the variety of panels floating around out there, the circuitry sports an unusually large number of options and jumpers. I'll also describe some embellishments you may want for a few oddball panels that lie just outside the standard hardware's capabilities.

With the hardware up and running, we will explore some bitmapped graphics, using Conway's Game of Life to generate the dots. Each type of panel has a different memory layout, making the task more complex than it seems and giving us a good reason to use modular firmware. Fortunately, graphic output devices are easy to debug when your errors appear right in front of your eyes.

Finally, I'll show how to generate text characters on a bitmapped panel, a trick that should come in handy for status displays on your embedded projects. The LCD firmware uses standard ANSI cursor control codes, allowing the same output data stream that reports status through the serial port to a PC comm program on your

host system to drive the LCD panel. If your target system has a 640×400 panel, you'll see 50 lines with 80 characters each. That should be enough, right?

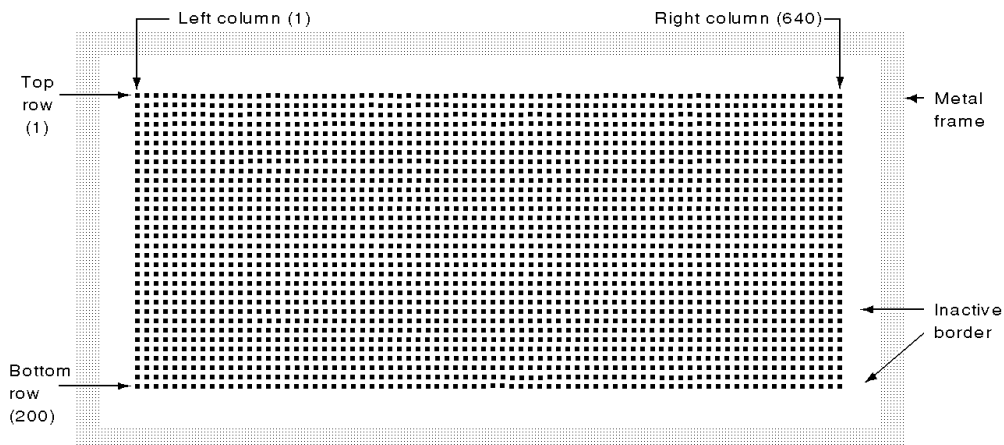
Dots in Ranks and Rows

Graphic LCD panels are fairly easy to understand, but the nomenclature certainly doesn't provide any assistance. Each manufacturer uses different names for the control signals and I've even seen one signal with two names in a single data sheet. I'll use signal names that I find descriptive, but you should plan on a little data sheet spelunking to match them up with your panel.

A graphic LCD panel consists of a rectangular array of dots similar to Figure 1. Generally, the grid has its larger number of dots running horizontally, so we'll call an array with 640 dots along each of 200 rows a "640×200" panel. It's reasonably easy to produce a 200×640 portrait display, should you need one for your next project, using firmware to rotate the character bitmaps a quarter-turn. As the saying used to go, "We control the vertical".

Figure 1

A graphic LCD panel is essentially a thin double-pane window: you actually see the reflector or backlight atop the circuit board underneath the panel. Transparent electrodes laid out in horizontal rows on one pane and vertical columns on the other delineate the dots at their intersections. The liquid crystal syrup between the panes reacts to the applied electrical field at each dot by rotating the transmitted light's polarization; fixed polarizers outside the panes cause the dots to appear dark or light as the dots change their polarization. The metal frame clamps the whole affair to the underlying circuit board and ensures good connections to the hundreds or thousands of column and row drivers.

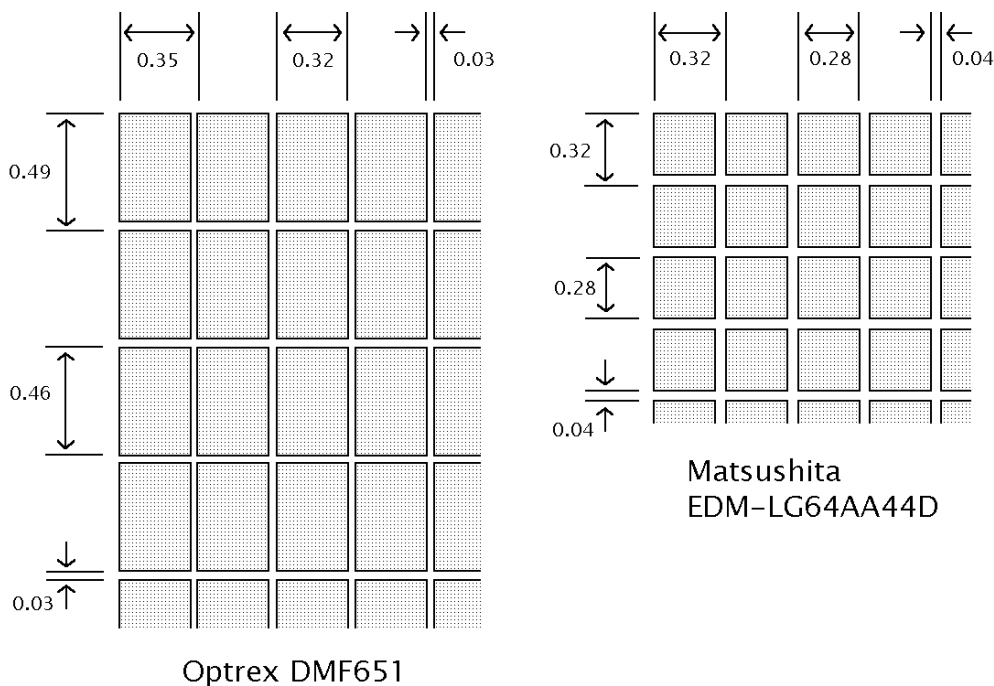


Although older panels had rectangular dots, most recent panels present square dots in equally spaced rows and columns. The small gap between each pair of dots is inactive, so, unlike pixels on a CRT, each dot has crisp, distinct edges. Figure 2 shows the dot dimensions for Optrex DMF651 640×200 and Matsushita EDM LG64AA44D 640×400 panels.

The panels we'll use for this project have a binary interface: each dot must be either completely ON or OFF. The newer VGA and SVGA panels can present continuous grayscale or color images, but driving those displays is a whole 'nother subject. For now, each dot on our panels represents a single bit: a DMF651 displays 128,000 bits. Yes, that's a nice round decimal number with lots of zeros, neither the binary 128 kilobits nor its decimal equivalent of 131,072 bits.

Figure 2

The width of the transparent row and column electrodes determines the shape of the dots at their intersections. Because the electrodes must be isolated from each other, every pixel has an inactive border surrounding it. Recent panels, typically those with 400 or more rows, have square dots. This simplifies life for graphic programmers: displaying *round circles* and *square squares* take no special effort.



A binary 1 bit may make its corresponding dot either transparent or opaque, depending on the liquid crystal chemistry, the driving electronics, and the arrangement of the external polarizers and reflectors. Because we have complete control over the data, dark dots on a light background or vice versa require no circuit changes. However, applying a **NOT** operation to every bit cannot change the inactive border around each dot. In general, you should set up the data so that a 0 bit produces a dot that matches its border and a 1 bit is distinctly different.

Contrary to popular opinion and despite what your eyes tell you, those bits are not all active at the same time. You cannot just write 128,000 bits into the panel and go about your business: the panel itself does *not* have a frame memory. You must send the same bits to the panel at least 60 times per second to produce a stable, flicker-free image. Surprised?

I recall a discussion with someone who tried driving a big, graphic LCD directly from an 8031 microcontroller's output bits. Once we went over the code's timing, he realized what was wrong: his hardware refreshed the display at about 3 Hz! An 8031 just can't supply four bits every 500 ns, making some additional support hardware a distinct necessity. Whether a big, bitmapped panel made sense in an 8031 application was another question that we didn't explore at the time.

Anyhow, an "external controller" is just that "additional support hardware" squashed on a single chip. It manages all the control signals, refreshes the panel data, and provides a convenient CPU interface. You'll find two catches: any given controller can handle only a subset of the panels out there and they all come in those minuscule, awkward, surface-mount packages. What fun is that?

After you decide on the refresh rate, simple arithmetic provides the panel data rate:

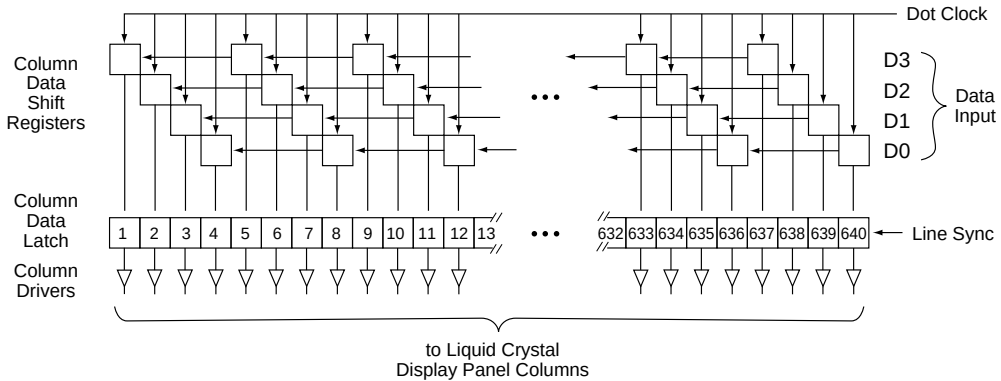
$$(128,000 \text{ dots/frame}) \times (60 \text{ frames/second}) = 7.6 \times 10^6 \text{ dots/sec} = 1 \text{ dot per } 130 \text{ ns}$$

Unlike a CRT that lights up only one pixel at a time, an LCD panel accepts several bits in parallel. For example, the DMF651 clocks in four bits (that's one *nybble*, pronounced with the *y* from *nymph*, not the *y* from *nylon*) every 520 ns or so. Transferring the 640 bits appearing in each row requires only 160 clock cycles and displaying the entire 200 line frame takes exactly 32,000 **Dot Clocks**.

Rather than presenting each nybble as it arrives, the DMF651 accumulates them in a 4-bit-wide, 160-element-long shift register. When that register contains all the bits for the next row, a single **Line Sync** pulse transfers them to a 640-bit Column Data Latch leading to the panel's column drivers. Figure 3 shows the connections for the beginning and end of this circuit.

Figure 3

A shift register accumulates 160 groups of four bits to drive all 640 dots in a single row simultaneously. The Dot Clock sets the basic timing for the entire panel, because all the other signals are defined in terms of its period and transitions. The Line Sync pulse transfers data from the shift register to a 640-bit parallel latch that holds the output stable while the next 640 bits arrive.



Yes, I know it's sadly inconsistent to talk about LCD *rows* and **Line Sync** signals. Unfortunately, my notes, sketches, and code started out that way and, despite my best efforts, it's much too late to change now. Take this as an object lesson about both the importance and difficulty of planning ahead. Check those LCD panel specs for more of the same, too.

Because the panel displays the dots in each row until all of the bits for the next row arrive, each row (and, thus, each dot) has a duty cycle of 1/200 rather than 1/32000. As you might expect, making a dot appear ON when it's active only 0.5% of the time presents a considerable challenge; don't even think about a 0.003% duty cycle. Hats off to the engineers who make LCD panels work at all, let alone as well as we've come to expect.

LCD panels don't implement row selection with a binary counter and decoder the way you might expect, either. The panels include another shift register, illustrated in Figure 4, behind the row drivers. Successive **Line Sync** pulses clock this register and pass the **Frame Sync** pulse through each flipflop in the chain. Unlike the 640-bit Column Data Latch shift register, the Row Select register has no output latch, because the **Frame Sync** pulse activates only one row driver at a time. We will meet some exceptions to that rule later in this chapter.

A **Frame Sync** pulse accompanying a **Line Sync** pulse marks the first display line. Because **Line Sync** pulses occur at the end of each line, the **Frame Sync** pulse occurs *after* the first row of data arrives. Figure 5 shows the relationship between the various pulses. It should go without saying that not all panels operate quite the same way, but you can see the general ideas.

Figure 4

The outputs of a 200-bit shift register drive the LCD panel rows. The Frame Sync pulse passes from one latch to the next on each Line Sync pulse, activating only one row at a time. To achieve a steady display, you must refresh each row often enough to prevent visible flicker, which implies a rate that typically exceeds 60 Hz. Each row occupies 1/200 of the total frame time, or about 80 μ s in each 16 ms. The jolt applied to the liquid crystal material turns the dot ON within that 80 μ s and it gradually fades OFF during the 15,920 μ s until the next pulse.

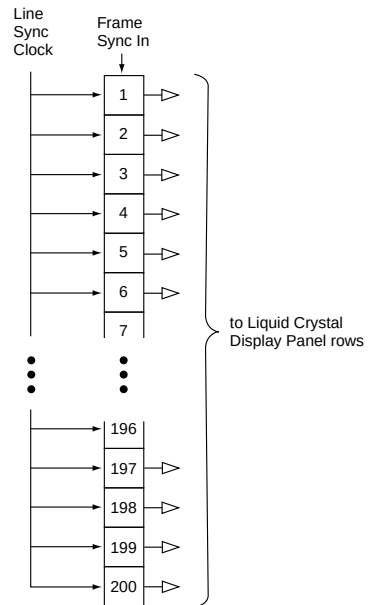
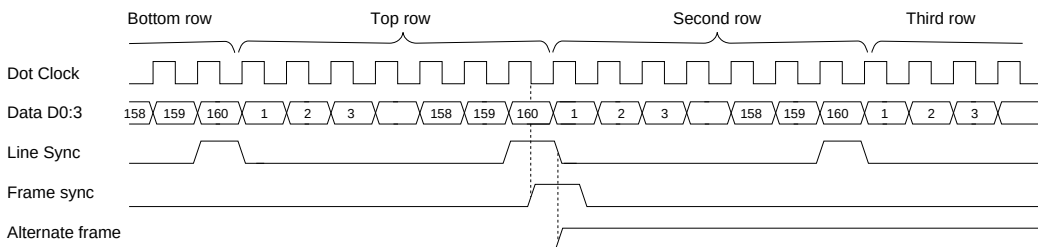


Figure 5

Each falling edge of the Dot Clock signal transfers data into the column data shift registers. If the Line Sync signal is high when Dot Clock goes low, the shift register contents transfer to the column driver latches. The Line Sync signal also clocks the Frame Sync pulse through the row driver shift register shown in Figure 4. Note that Frame Sync is active at the *end* of the top row, not between the top and bottom rows as you might expect from your experience with CRTs.



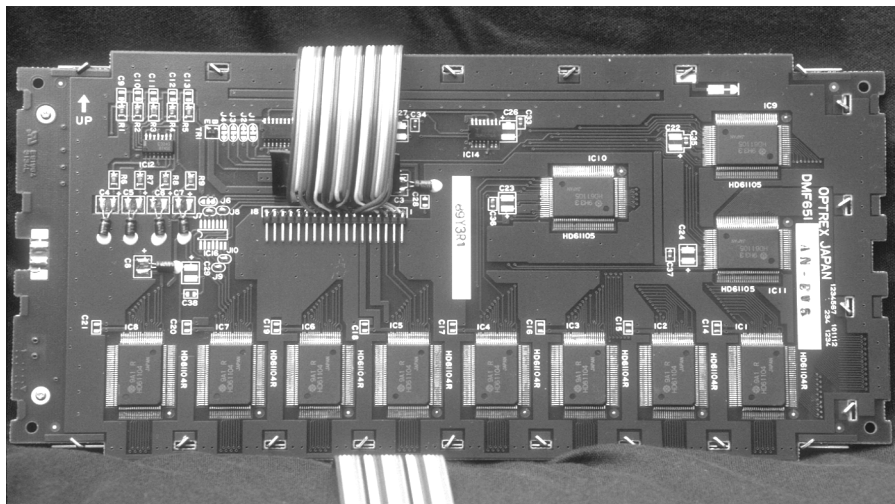
The DMF651 requires an additional signal that alternates from frame to frame. To produce this signal, which I call **Alternate Frame**, you just toggle a flipflop on each **Frame Sync** pulse. More recent panels generate the signal internally and I suspect this has more to do with the size of the driver IC packages than anything else. After all, it's easy to add an LSI flipflop when you have a spare output pin, but impossible to justify adding another whole chip for just one bit.

Using shift registers for both the row and column logic circuits allowed the LCD engineers to split the circuitry into identical units, precisely the right tactic for good LSI chip design. For example, the DMF651 shown in Photo 1 has eight Hitachi H61104 column driver chips and three H61105 row drivers. Each H61104 contains 20 elements of the four-bit column shift register and drives 80 columns. Each H61105 has 80 row selection bits, leaving half of the last chip unused.

The actual process of converting row and column selection bits into visible dots lies, mercifully, hidden in those LSI driver chips. Several different schemes, each applying to a different panel configuration, all boil down to applying a high voltage to the dots at the intersection of each active column with the currently selected row, while not applying quite so much *oomph* to all the other dots. Under the influence of that jolt, the liquid crystal compound twists the transmitted light's polarization and the dot becomes either opaque or transparent, depending on the panel's external polarizer and reflector arrangement.

Photo 1

This Optrex DMF651 640x200 LCD panel uses eight Hitachi H61104 column driver chips and three H61105 row drivers.



Because of the high voltage required to drive the chemistry, graphic LCD panels are *not* friendly 5 V devices. In addition to the usual +5 V logic supply and ground, they require an LCD drive voltage between about -9 and -30 V at perhaps 25 mA. Some displays use both a fixed, relatively high current, LCD bias voltage and a separate contrast adjustment voltage that draws much less current.

You must adjust the negative voltage to compensate for the inevitable temperature changes that affect the liquid crystal material's optical response. A trimpot works well, if your application can stand a manual adjustment, or you can use a DC to DC converter to produce an adjustable negative voltage directly from the +5 V logic supply. In any event, recheck the negative bias voltage when you can't see any dots. Trust me, you'll wonder why the panel is completely blank, too.

Most of the LCD panels you see here originally appeared in laptop computers or similar low power widgets. Their CMOS logic reduces power consumption, but CMOS logic levels make no concession to TTL drivers. For example, the V_{IH} for most panels is typically $0.8 V_{CC}$ (where V_{CC} is the panel's supply voltage), which translates into 4.0 V. One panel expects V_{IH} levels exceeding $0.9 V_{CC}$! You should drive the interface signals with HCT or HCTLS gates, because ordinary LSTTL has a minimum V_{OH} spec of 2.4 V, well below the CMOS threshold.

BEWARE! The panel's logic-level inputs and outputs have *no* protection against those negative LCD bias voltages. Despite the fact that the negative supply lines may be sandwiched between logic signals on the LCD connector, you must *never* short those adjacent pins, not even once, not even when your scope probe slips. Word of advice: buy two LCD panels and save on shipping...

By the way, the Firmware Development Board drivers have no protection, either. One slipped probe can blow away both the panel and your own logic. That's true of nearly all panels and their control logic, as this hardware is really meant for OEM production, not the sort of tinkering and experimentation we do here.

Panels specs also present different sequences in which you must apply and remove their supply voltages and logic signals during startup and shutdown. The penalty for disobedience can be your panel's death due to SCR latchup as its CMOS logic incinerates itself. If you're designing a specific panel into your project, you can meet its peculiar demands, but I don't know of a good general solution that can handle a wide variety of panels. The FDB circuit includes a DPDT relay that disconnects the LCD drive voltages when the ISA bus **ResDrv** signal goes active, but that surely doesn't meet all the specs. So far, though, so good.

Finally, not only do the manufacturers use different signal names, bias voltages, and power sequencing, but each LCD panel also sports a unique connector. Sometimes,

Figure 6

Each of these graphic LCD panels sports unique electrical and connector specs. This table summarizes the important characteristics and pin functions for some of the panels in my stash. An “n/c” entry means a pin is not used, while a blank entry means the connector doesn't have that pin. A separate cable carries power for the backlight.

Panel Spec	Optrex DMF651	Matsushita LG64AA44D	Sharp LM64015T	Toshiba TLY-365-121	Hitachi LM215XB	Epson EG7004
Dots	640 x 200	640 x 400	640 x 400	640 x 200	480 x 128	640 x 400
Bits	4	8	4	4	4	4
Col Drivers	640	640	640 x 2	640 x 2	240 x 2	640 x 2
Row Drivers	200	200 x 2	200	100 x 2	64 x 2	100 x 2
Clocks/Row	160	160	320	320	240	160
Dot Clock Pd	480 ns	480 ns	240 ns	480 ns	960 ns	240 ns

Connector Pinout

Pin	Optrex DMF651	Matsushita LG64AA44D	Sharp LM64015T	Toshiba TLY-365-121	Hitachi LM215XB	Epson EG7004
1	Bezel Gnd	+5 V	Frame Sync	Bezel Gnd	D1 UL	+5 V
2	Line Sync	Bezel Ground	Line Sync	n/c	D2 LL	Ground
3	Dot Clock	Dot Clock	Dot Clock	Frame Sync	Frame Sync	-Contrast
4	Alt Frame	Enable	n/c	Line Sync	Alt Frame	Line Sync
5	-Contrast	Frame Sync	n/c	Dot Clock	Line Sync	Alt Frame
6	+5 V	Line Sync	+5 V	Ground	Dot Clock	Enable
7	Ground	Ground	Ground	D0	D3 UR	Row Shift
8	-23 V	n/c	Adj -21 V	D1	D4 LR	Frame Sync
9	D0	D0 upper	D0	D2	+5 V	Dot Clock
10	D1	D1	D1	D3	Ground	Col Enable
11	D2	D2	D2	Ground	-10 V	D0
12	D3	D3	D3	+5 V	-Contrast	D1
13		-22 V		-Contrast (?)		D2
14		-Contrast		Adj -22.5 (?)		D3
15		Ground		Ground		
16		D0 lower				
17		D1				
18		D2				
19		D3				
20		Ground				

different panels from the same manufacturer have different connectors. Figure 6 summarizes what I know about the panels I've tested, along with my pin name translations. The Graphic LCD Interface uses a 2×13 ribbon cable header that doesn't match *any* of the panels, but I got pretty good at soldering wires directly to panel connectors. Hint: don't waste your time trying to find the mating connector for your panel, unless, of course, you're going into production.

Seeing the Big Picture

The circuitry we'll explore in the next chapter serves as, perhaps, an example of retro design: the *right* way to do it requires the exact LSI controller for the particular panel you're using. But, because this book shows you *how* things work, I don't feel too badly about illustrating key points with a handful of TTL chips and a lot of firmware.

As with the ISA bus interface logic we built in Chapter 3, you can certainly reduce the number of discrete chips using programmable logic: an FPGA seems like a nice fit for the logic in Figure 7. Once again, I used TTL gates to keep things simple for folks who don't have access to that kind of hardware. If you've got the capability, see just how small you can make all this logic.

With that in mind, Figure 7 shows the overall Graphic LCD Interface block diagram. The PC sees the interface as a 32 KB block of RAM, called the **LCD Refresh RAM**, and a write-only output port. The LCD panel sees the interface as data bits and control signals. I'll start on the LCD side, because those functions determine how the PC side must operate.

Displaying one 640×200 frame requires 32,000 clock cycles. Even though each cycle transfers only four bits, a byte-wide, 32 KB (32,768 decimal) static RAM chip is an obvious choice. If we store the dots in the low-order four bits of each byte, a simple 15-bit address counter will extract them in the right order.

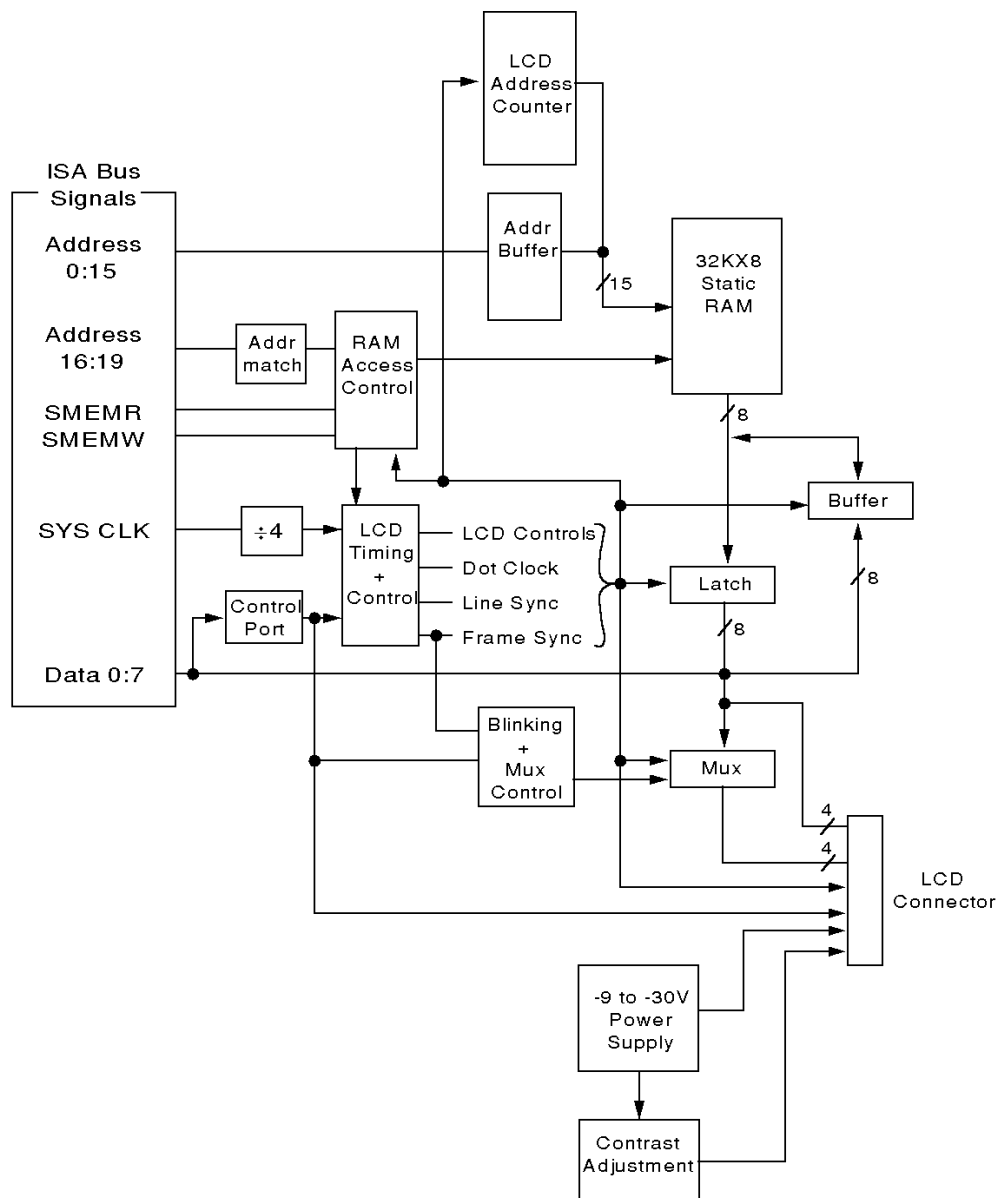
The LCD **Dot Clock** period must be about 520 ns to refresh the entire display at a 60 Hz rate. A key part of this design fell into place when I realized a 480 ns **Dot Clock** would provide 65 Hz refresh. You should notice that 480 ns comes from an ISA Bus magic number: divide the 120 ns **SysClk** signal by four, thus increasing the period by a factor of four, and there you have it.

Most LCD panels specify a refresh rate between 60 Hz and 80 Hz. For a 200-line panel at 65 Hz, each line would take

$$(200 \text{ lines/frame}) \times (65 \text{ frames/sec}) = 1 \text{ line every } 77 \mu\text{s}$$

Figure 7

The Firmware Development Board's Graphic LCD Interface forms a dual-ported RAM. The PC can read and write data while the LCD panel displays dots fetched from the same location. The two sides of the interface access the RAM on alternate Dot Clock phases, so the operations aren't really simultaneous. The output multiplexer and blink logic include additional circuitry (and jumpers!) to support a variety of LCD panels.



and, with 160 **Dot Clocks** per line, the clock period works out to about 480 ns. See how to apply that to your panel?

The panel specs also list a minimum **Dot Clock** period, typically in the 160 ns to 330 ns range, or the corresponding maximum clock frequency. Running the clock at 480 ns stays comfortably above the minimum period, but there's another factor we must consider.

If each **LCD Refresh RAM** access requires the entire 480 ns available in each **Dot Clock** cycle, the PC has no time to write or read data. We must somehow implement the **LCD Refresh RAM** as a dual-ported memory, because forcing the firmware to check status flags simply won't provide the bandwidth required for a bitmapped display. You may recall the gyrations required to prevent visible snow on the old CGA graphics adapter... we don't want to repeat *that* mistake.

The Graphic LCD Interface simulates simultaneous PC and LCD access by using alternate 240 ns phases of each 480 ns clock cycle. While **Dot Clock** is high, the PC can access the RAM through the ISA bus address and data buffers as usual. When it's low, the LCD has full access for its address counters and data latch.

The LCD panel must receive stable data throughout the entire 480 ns cycle, however, so **Dot Clock** captures the RAM output data in a latch. Most panels accept data on the falling edge of **Dot Clock**, giving the bits nearly 240 ns of both setup and hold time. The latch holds the data stable while the **LCD Refresh RAM** processes PC accesses from the ISA bus during the next 240 ns phase, when **Dot Clock** is high.

Although the DMF651 uses only four data bits, it seemed a shame to waste half the RAM. A multiplexer after the latch selects either the high or low nybble under control of a signal from the Blinking and MUX Control logic. That signal, picked from one of the outputs of an 8-bit counter driven by **Frame Sync**, switches the multiplexer between its two 4-bit inputs every 1/16 to four seconds.

A little firmware can easily implement nearly any blinking scheme you'd like, because the two RAM nybbles can hold entirely independent data. If you fill the high nybble with zeros, the ON dots in the low nybble blink. Fill it with ones to get a blinking background while the data dots remain ON. Duplicate the low nybble in the high nybble, complement the bits, and you get a blinking reverse image. Versatile enough?

The overall blink rate must be under firmware control, because different panels have different response times. The 1/16 s blink runs faster than any panel I have available, while a 4 s blink should be glacial enough for a nearly frozen panel. You

can also vary the blink rate to attract attention. Because the blink signal changes only after an entire frame, all the dots will blink at the same rate... unless you get tricky and rewrite the RAM on the fly, a trick I'll mention again in Chapter 16.

All of this hocus pocus depends on dots on the LCD panel being just bits in PC memory. The **LCD Refresh RAM** appears in the PC's address space at D000:0000, just after the FDB's battery backed RAM at C800:0000 through C800:7FFF (a.k.a. C000:8000 through C000:FFFF). That circuit appeared in Chapter 7 and I discussed some of the issues involved in ISA bus memory accesses in Chapter 6.

The RAM Access Control Circuitry synchronizes **Dot Clock** with the ISA bus signals during each memory access, leaving the RAM in the proper state by the time each bus cycle finishes. Basically, the ISA bus runs slowly enough that we can pull a fast one on it! A 120 ns RAM cycles quickly enough for this application, so we don't need any particularly exotic hardware, either.

In desktop PCs, however, the 64 KB between D000:0000 and D000:FFFF may form an Expanded Memory page frame. Address space being the most precious commodity below the 1 MB line, EMS boards or **EMM386** programs may use that 64 KB block as a window into megabytes of expanded RAM. You can move the **LCD Refresh RAM** to A000:0000 or B000:0000 if your system doesn't have a video card at one of those locations... not likely in a desktop system, I dare say.

Variations on a Theme

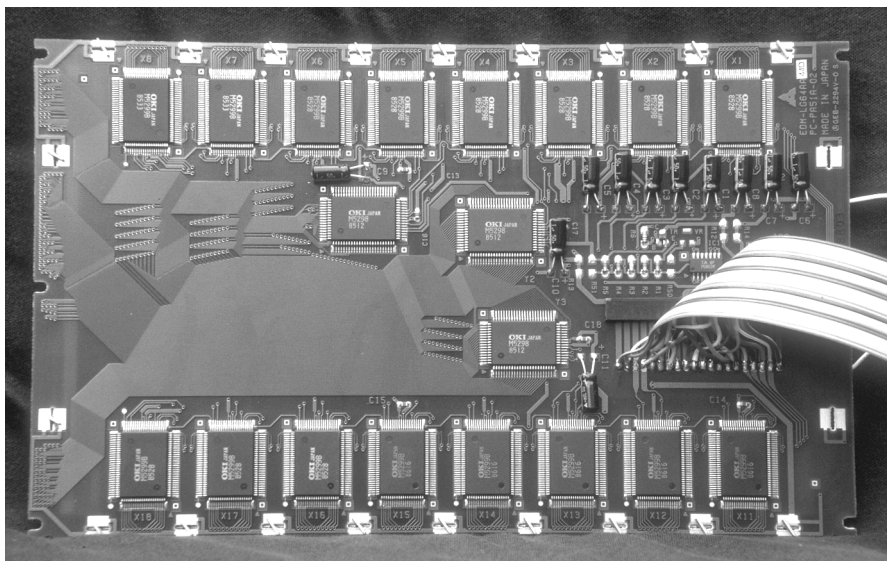
If a 640×200 panel doesn't have enough dots for you, the next step up is 640×400. With VGA resolution at 640×480, SVGA at 800×600, and current laptops hitting the 8514/A level of 1024×768, everyone simply *must* maintain the pace. As a result, you can find older, smaller panels at reasonable prices. In fact, you could probably go into OEM production with surplus panels from previous-generation laptops.

Doubling the number of lines obviously doubles the number of dots visible on the panel. Because the overall refresh rate must remain about the same to avoid flicker, the panel must receive twice as many bits in about the same time. Panel designers have only two choices: transfer twice the number of bits per **Dot Clock** cycle, or send the same bits twice as fast by doubling the **Dot Clock** frequency. I have panels using each method and designed the Graphic LCD Interface for both.

Photo 2 shows the back of a Matsushita EDM LG64AA44D 640×400 panel, which, compared with Photo 1, has eight additional column driver chips. Each group of eight drivers handles four bits, so this panel receives eight bits in each **Dot Clock** cycle.

Photo 2

Compared to the 640x200 LCD in Photo 1, Matsushita's EDM LG64AA44D 640x400 LCD panel has eight additional column driver chips that accept four more bits per Dot Clock cycle. The white wires on the right carry AC power to the electroluminescent backlight.



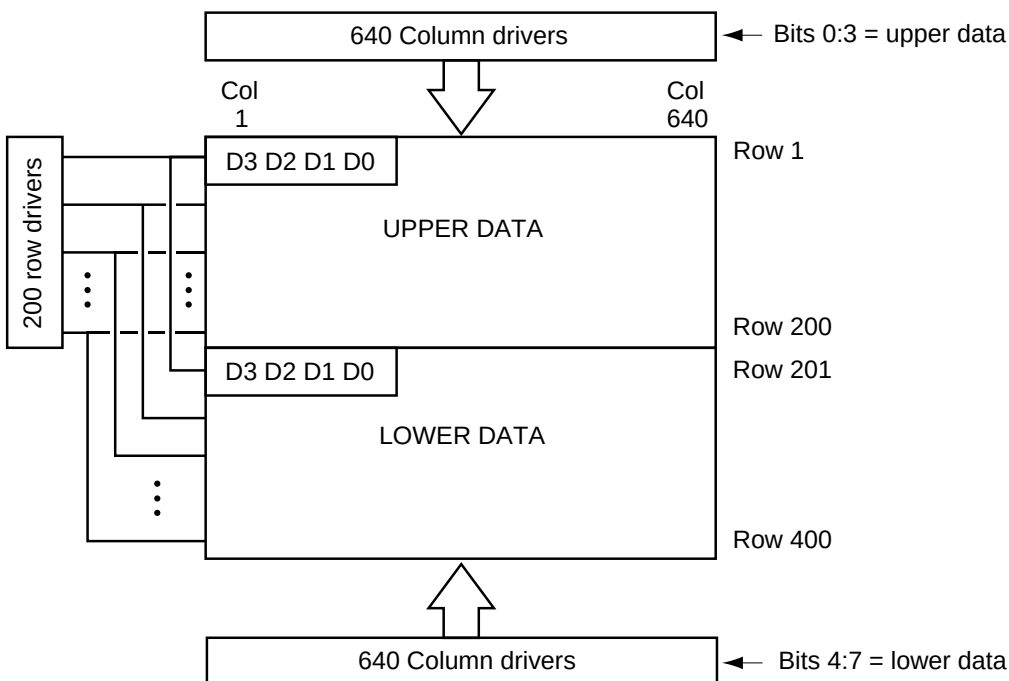
It has only three row driver chips, with each of the 200 outputs connected to *two* rows instead of one. Figure 8 sketches the layout: the display surface is split in two sections, each with separate column drivers. Thus, two separate rows are active simultaneously, with the two column driver sets presenting different data to the active row. You can think of the LG64AA44D as two 640x200 panels, butted together top-to-bottom on the same piece of glass.

Because the LG64AA44D requires eight bits on each cycle, the Graphic LCD Interface's data multiplexer serves no function. The Blinking and MUX Control logic emits a constant zero that routes the low-order nybble from the data latch through the MUX to the LCD connector. The high-order nybble goes directly from the latch to the connector and the panel latches all eight bits simultaneously on each **Dot Clock** cycle.

Bits 0:3 of each RAM byte appear on the upper half of the panel, while bits 4:7 appear on the lower half. The firmware must account for the fact that the bits in each byte show up at two widely separated locations. This is not the worst arrangement you'll meet on an LCD panel, as you'll see in a few pages.

Figure 8

Although the Matsushita LG64AA44D display has 400 rows of 640 dots, it uses only 200 row drivers. Each row driver activates two rows, one in each half of the display. One set of column drivers displays dots in the upper half, while an additional set drives the lower half. Because each set of column drivers handles four bits, the panel accepts eight bits on each 480 ns Dot Clock cycle.



The additional dots in this display are basically free, because the Graphic LCD Interface already includes a byte-wide RAM. All it takes is a little firmware to plunk the dots in the right spots. In fact, the same code can handle either 200- or 400-line panels. Just don't blink that big display! (Go ahead, try it...)

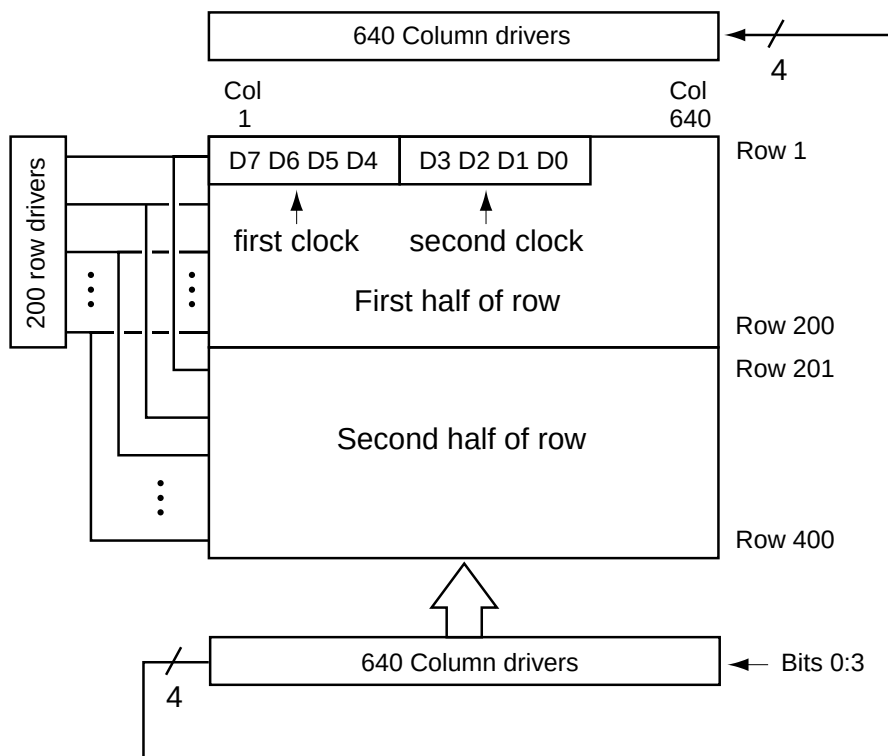
Conversely, the Sharp LM64015T 640×400 panel diagrammed in Figure 9 uses the other technique: it accepts four bits on each double-speed 240 ns **Dot Clock** cycle. The two sets of column drivers run in series rather than parallel. In essence, we have the same chips as the LG64AA44D, but they run faster to keep up with the data. You'll recall that CMOS circuitry dissipates more power when it runs faster, which may affect the type of panel you select for a battery powered system.

In effect, you have a 1280×200 panel whacked in half, with the two 640×200 pieces stacked atop each other. Each logical row, twice the length of a visible row, requires 320 double-speed **Dot Clocks**. The second half of each row appears on the bottom of the panel.

The Blinking and MUX Control logic switches the multiplexer between the two nybbles on each half of the **Dot Clock** cycle. Because the panel accepts one nybble on each half cycle, the data bits show up together on the screen. It's easy to write graphic routines for a panel where bits 4:7 adjoin bits 0:3 in their natural order, even if the rows seem to be twice as long as they really are.

Figure 9

The Sharp LM64015T has 400 rows of 640 dots, with two sets of column shift registers chained together to hold 1280 bits. Each of the 200 row drivers activates two physical dot rows, but, unlike the Matsushita panel shown in Figure 8, they show two halves of a single logical row. Each double row requires 320 cycles of a 240 ns Dot Clock, transferring 1280 bits in 320 groups of four. The Toshiba TLY-365-121 mentioned in the text has a similar layout with 100 row drivers, 200 rows, and a 480 ns Dot Clock.



The Graphic LCD Interface RAM circuitry runs at the same speed as before, fetching and latching a new byte every 480 ns. The trick lies in the multiplexer, which must now present each nybble for 240 ns. The panel runs from a double-speed **Dot Clock**, but that doesn't affect the rest of the interface.

The LM64015T uses a blindingly bright, cold cathode, fluorescent tube backlight that requires about 1 kV to fire up. Although you can read some lower-performance electroluminescent panels when they're unlit, this one is *completely* useless without its backlight. I don't know of a consistent surplus source for the special CCFT inverters and a new inverter costs nearly as much as a surplus panel. But, it's a nice panel if you can get it working...

Along the same lines, the Toshiba TLY-365-121 640×200 panel has two sets of column driver chips and a pair of row driver chips chained in series. It clocks four bits every 480 ns, with each line using 320 **Dot Clocks**. This is also a 1280×100 panel in disguise, with the two halves of each row stacked atop each other.

The panel connections resemble Figure 9, with 100 row drivers simultaneously activating pairs of the 200 display rows. Although you can think of this as a double-speed version of a 960 ns 640×100 panel, I doubt one of those ever appeared, if only because the aspect ratio seems so weird. In any event, the Graphic LCD Interface handles this panel with ease.

My TLY-365-121 can serve as an example of what happens when you buy surplus stuff: it arrived with the wrong documentation. Mercifully, the connector pinout was nearly correct and I didn't fry the electronics while discovering the confusion. It seems the panel expects one variable negative supply on pin 14 rather than a fixed -22.5 V supply with a separate contrast control on pin 13. At least, that's the way it worked for me.

I've also looked at the Hitachi LM215, a 480×128 display sporting *four* sets of column drivers clocked at 960 ns. Each of the four data bits paints one quadrant and, as you can imagine, drawing anything becomes an intricate bit twiddling exercise. The Graphic LCD Interface can handle this one with an additional flipflop that doubles the 480 ns **Dot Clock** period. As you'll see, we must also store duplicate data in successive even/odd bytes to keep the output latch stable throughout the entire 960 ns cycle.

The Epson EG7004 640×200 panel requires an extra set of clock signals that route sync pulses through the shift registers. Think of that one as a challenge... you can find much easier panels to use, even as surplus items. Given the number of other panels in my collection, I gave up.

Release Notes

One thing is absolutely certain: bigger, better, and faster graphic LCD panels will appear for your use as the laptop computer market churns along. Somewhat to my surprise, I recently found a 640×480 panel for \$9.95, *including* that usually impossible-to-find CCFT inverter.

After you acquire a few LCD panels, spend some time reading the documentation and thinking about how they work, before firing up your soldering iron and keyboard. Although the Graphic LCD hardware cannot handle every panel you'll find, you can probably adapt the basic design to cope with nearly anything. The main limitations will be the cramped address space available for the **LCD Refresh RAM** on the ISA bus and the higher refresh rates used by newer panels.

I'll discuss some modifications to the basic design in the next few chapters, as we explore the circuitry and firmware.

13 Bitmapped LCD Panel Hardware

Perhaps we've lost our sense of wonder: name another manufactured product with a quarter of a million parts laid out for your inspection. Every dot on that 640×400 LCD panel must be essentially perfect, because human eyes are very, very good at picking out tiny differences. Sure, a CD-ROM has more dots, but it's just a silver disk to anything but an electron microscope.

In this chapter, we will build the hardware that puts every one of those dots under your control. I'll concentrate on the details of 200- and 400-line displays, because they're readily available, while mentioning some of the changes required for other panels. The block diagram in the previous chapter will guide you through the circuitry in the schematics you'll find here.

Wading through the intricate timing relationships of the sync and data signals takes considerable mental effort. This is, alas, typical of most hardware projects, with the Devil hiding in the details. Getting even one detail wrong can wreck a project, so pay attention as we go along and see if I've missed anything!

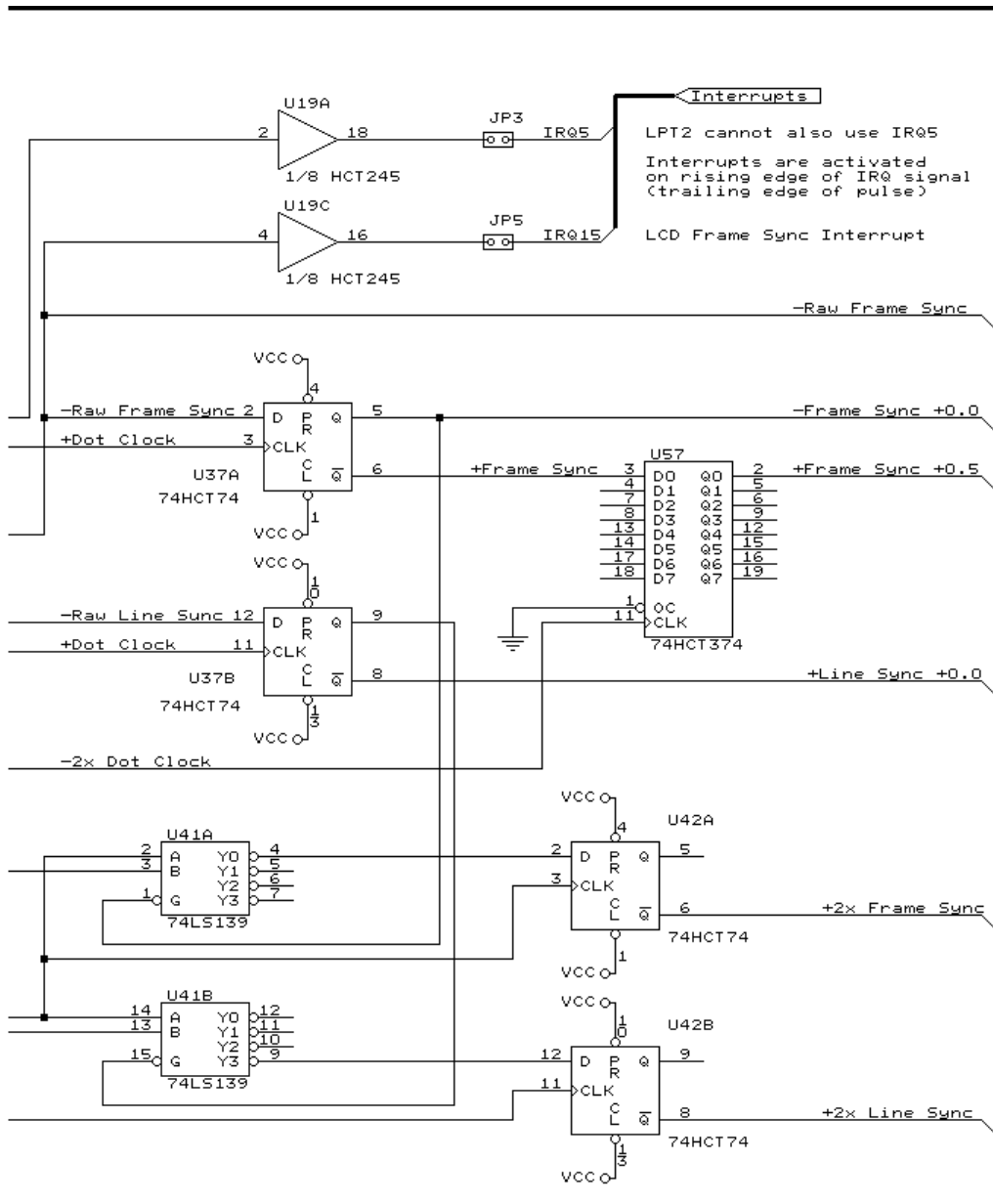
You should peek ahead to Chapter 14, where I cover the test code that checks out the circuitry, before you fire up your soldering iron. By verifying your work in small steps, you'll save yourself lots of effort... I know I did.

Counting Clocks

The LCD Timing and Control section forms the heart of the Graphic LCD Interface. It generates the clock and sync pulses that control the LCD panel, as well as the signals that orchestrate the PC side of the interface. Schematic 1 shows this part of the circuit.

The ISA bus **SysCk** signal defines the basic interface timing. U38, an LS109 dual J/-K flipflop, divides the 120 ns **SysCk** by two to produce the double-speed **2x Dot Clock**, then by two again for the regular 480 ns **Dot Clock**. U39B, part of an HCT74, reduces that rate for the half-speed 960 ns **Dot Clock/2** required by slower panels.

The actual **SysCk** frequency, and thus the fundamental ISA bus cycle, for any given system may range from about 120 to 125 ns, because the system logic generates it from the CPU clock. As a rule of thumb, the bus cycle will be a simple integer multiple of the fundamental clock cycle, although recent systems include phase-locked loops with fractional divisors.



Schematic 1

This circuitry defines the fundamental Dot Clock period that drives the rest of the Graphic LCD Interface. The 82C54, installed in Chapter 4, has new clock and gate lines. The various Dot Clock and Sync signal outputs can control a variety of panels.

For example, an 80 MHz 80486DX2 system uses a 40 MHz external clock that, divided by 5, produces an 8.000 MHz bus clock and a 125 ns cycle time. An old 33 MHz 80386SX, which actually uses a simple 33 MHz clock, has an 8.25 MHz bus and a 121 ns **SysClk** cycle. There's some specsmanship involved with these numbers, so don't be surprised if your **SysClk** seems a bit different than you expect. I'll continue to use 120 ns here, as it makes the numbers work out evenly.

In Chapter 4, when we first wired up the Firmware Development Board's 82C54 timer, we tied all of the **Gate** inputs high and drove the **Clock** inputs from a 7.16 MHz signal. Now, we can put that hardware to practical use! U17 in Schematic 1 represents the same 82C54 you've already checked out, with two channels producing **Line Sync** and **Frame Sync** signals for our panels.

Counter 1 produces the **-Raw Line Sync** signal by counting the number of **Dot Clocks** in each LCD row. The 640×200 and 640×400 panels I have require either 160 or 320 clocks per row, depending on whether they use a single- or double-speed interface. Because other panels have different requirements, using a programmable counter made more sense than a tangle of jumpers and gates.

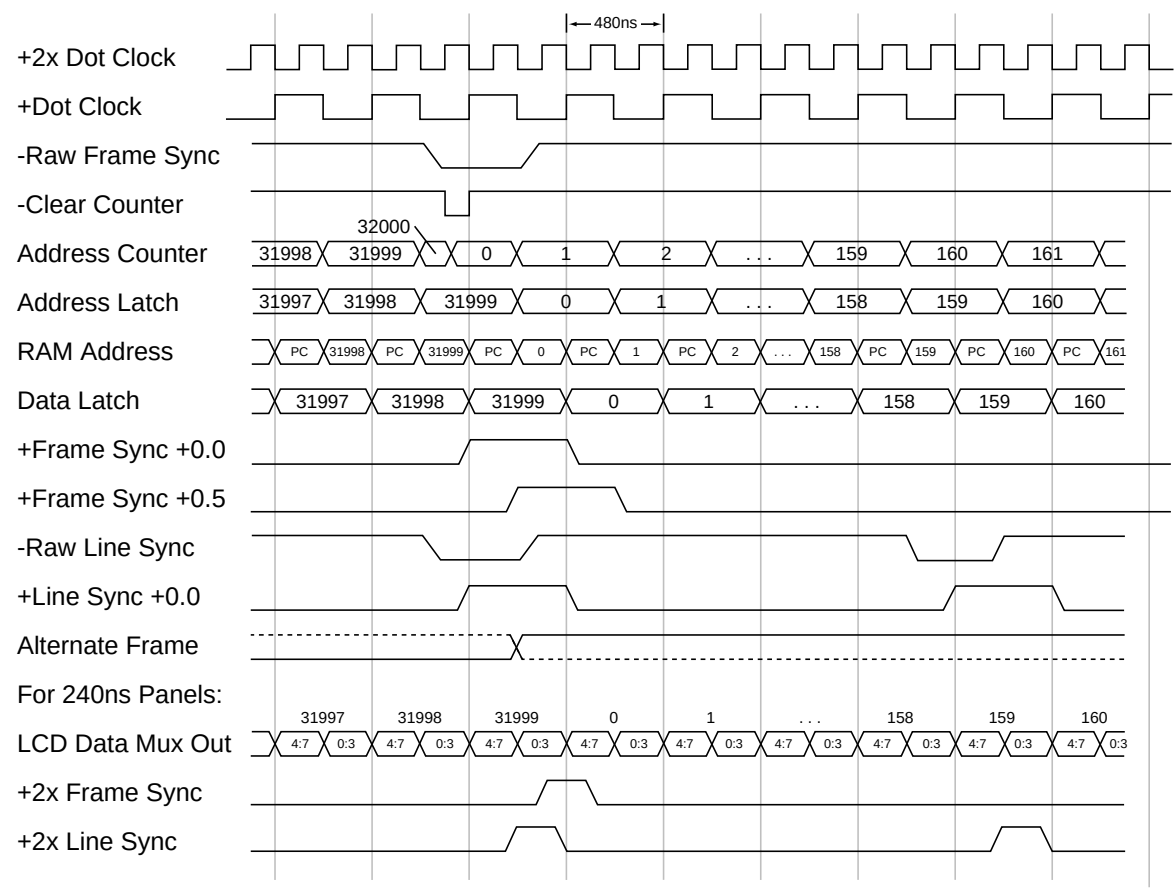
Counter 2 controls the number of **Dot Clock** cycles in one LCD display frame by generating the **-Raw Frame Sync** signal. Both 200- and 400-line panels require 32000 clocks, but other panels have different counts. Here, also, the 82C54 serves as a compact, 16-bit, programmable counter.

The key Graphic LCD Interface timing signals appear in Figure 1. The 82C54 timing specs say the output delay may occur up to 100 ns after the falling **Clock** input edge, which means you can't use the signals for critical timings. I used an HCT74 to synchronize the edges with **+Dot Clock**; the resulting pulses occur at the **"+0.0"** times relative to the rest of the signals.

Both 82C54 channels must begin counting on the same **Dot Clock** cycle to ensure that the two sync signals have the proper relationship. U39A disables the counters until the firmware sets the **+Enable LCD Counters** bit in the **LCD Control Port**. When U39A's clock rises, both 82C54 **Gate** inputs go high simultaneously and the counters are off and running.

While I was debugging this circuitry, the LCD panel would occasionally go blank. It turned out that Counter 1 unpredictably stopped counting for intervals ranging from a few cycles to a few hundred milliseconds. This immediately threw off the critical timing relationship between **Line Sync** and **Frame Sync**. Strangely enough, the 82C54 **Gate** inputs remained high and **Dot Clock** continued to run during the stoppages.

Figure 1
This timing diagram summarizes the Graphic LCD Interface timings. The 32 KB RAM holding the LCD dot patterns is timeshared between the PC ISA bus and the LCD circuitry. The -Clear Counter signal resets the LCD Address Counter at the end of each frame of data; that signal originates in the 82C54 counter described in Chapter 4.



The 82C54 has a classic “read every word” data sheet. The **Clock** input timing specs include this tiny footnote: “Low-going glitches that violate t_{PWH} , t_{PWL} may cause errors requiring counter programming.” Translated into plain English, this says that an imperfect **Clock** will cause the counter to misbehave in strange and mysterious ways. Hence, the RC filter and gate driving U38.

We think of TTL devices as digital: their outputs will be either high or low, never anywhere in between. In reality, of course, digital logic circuits are just analog circuits with a nonlinear transfer characteristic. If the input voltage falls between the valid logic 1 and logic 0 thresholds, you will certainly get *something* out... perhaps not a Boolean value and usually nothing that you really want.

Further investigation revealed **Dot Clock** was something other than the picture-perfect square wave I expected. Once in a while, a low-going glitch dented the high half cycle. After some delicate probing, I discovered an impressively noisy **SysClk** signal on the ISA bus. The clock noise sometimes became bad enough to force the LS109 latches into metastable states. The ensuing invalid output voltages stunned one or both of the 82C54 counters.

The effect was that the counters simply stopped counting for a while, then started up again after an unpredictable interval. Of course, when **Frame Sync** and **Line Sync** pulses don't occur in the proper relationship, the panel doesn't display anything useful. Although scope probes on the sync signals show that they're cooking right along, you can't easily see the phase relationship on a scope and may miss some crucial evidence.

Because we cannot clean up the ISA bus logic signals, we must cope with what arrives at the Firmware Development Board. One possible solution for this problem puts a Schmidt Trigger gate in the clock path to eliminate the noise. Unlike a standard logic gate, a Schmidt Trigger includes an analog feedback path that modifies the input switching thresholds based on the Boolean output. As a result, small glitches around the thresholds simply Go Away. However, I didn't want to squeeze an LS14 on the board to get just one inverter.

So, I used a simple RC filter to knock off the high-frequency noise, followed by a spare LS00 gate to square up the result. Yes, I could have substituted an LS132 (a Schmidt Trigger NAND), but the *other* circuits already using that package couldn't stand the additional delay. An RC filter on a (nominally) digital input, albeit not a textbook solution, works for this application. Take a look at *your* **SysClk**, consider all your options, then decide how you will attack the problem.

You may rest assured that it took me *much* longer to discover and stomp that glitch than to write about it...

Although the 82C54 defines the overall row and frame timing, its output pulses cannot drive the LCD panel directly. The dot data and sync signals must all arrive with the correct polarities at the proper times. Now, let's consider the dots.

A Chain of Dots

A 32 KB static RAM holds the binary data that will become visible dots on the panel. The **LCD Refresh RAM**, U45 in Schematic 2, must timeshare its signals between the ISA bus and the LCD panel. Obviously, only the PC will write into the RAM, but both sides must be able to read the dot data. The **Dot Clock** signal determines which side has access to the RAM at any given moment.

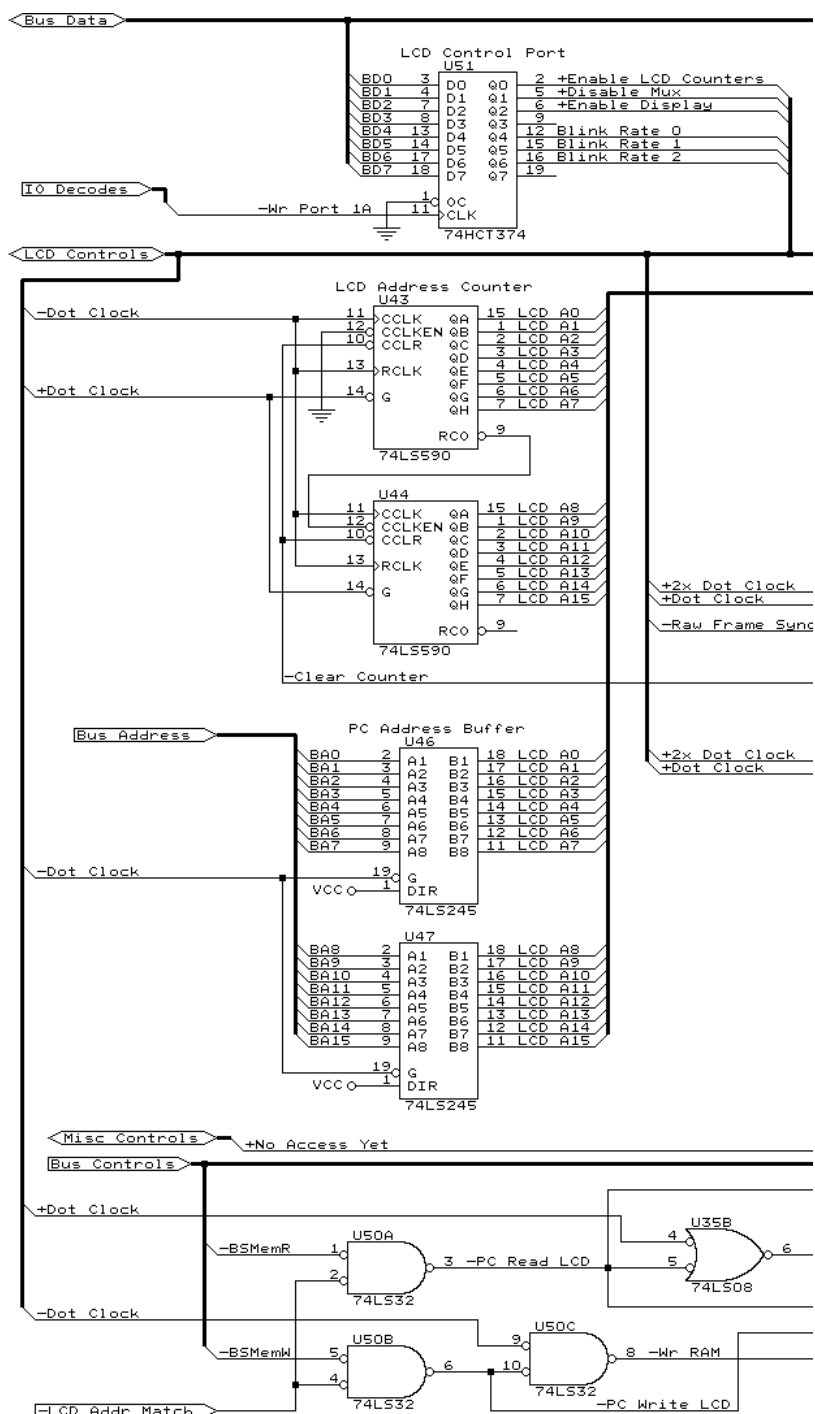
The **LCD Address Counters**, U43 and U44, provide the 15-bit address of the current RAM byte when the LCD has access. Each LS590 chip contains an 8-bit counter followed by an 8-bit latch with three-state outputs. Both the counter and latch change state on the rising edge of the clock input, which, in this circuit, is **-Dot Clock**. Therefore, the LCD address changes on the *falling* edge of **+Dot Clock**.

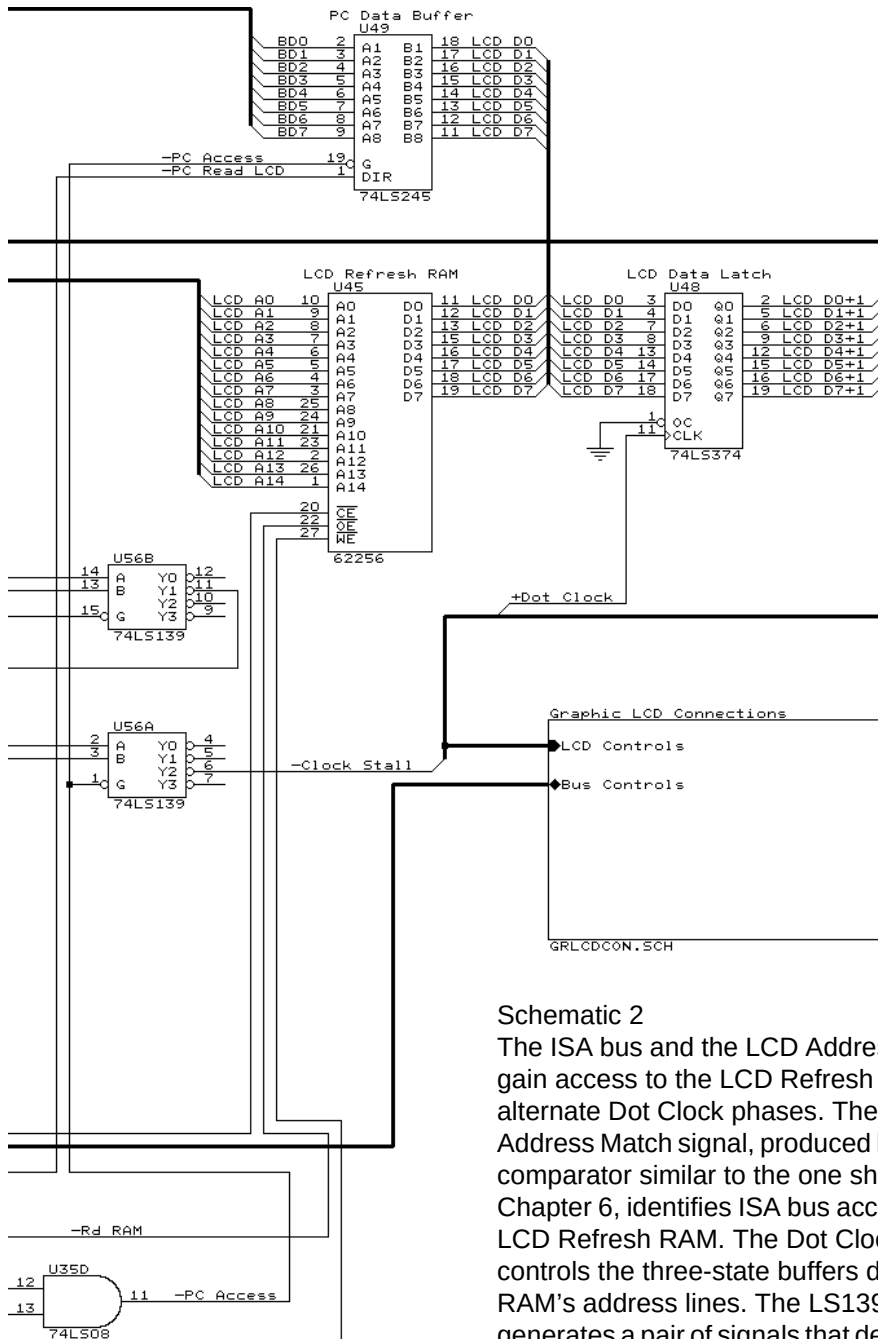
Note that the latch inside the LS590 is always one count *behind* the counter, because they share the same clock input. The latch captures its input value just before the counter changes on the rising edge of the clock input. Check the LS590 data sheet for more details, then ponder the timing diagram in Figure 1.

Frame Sync clears the counter address to zero at the beginning of each panel scan. The actual reset signal occurs during only a quarter of the **Dot Clock** period, thus ensuring that the LS590 counter will step from 0 to 1 on the next clock pulse. Figure 1 shows the timing: the reset pulse must be finished by the next **-Dot Clock** rising edge. Notice that the zero count is shorter than all the rest, due to the reset signal in the first quarter of the period.

The latch inside the LS590 solves a critical timing problem: the path length from the **Frame Sync** reset logic through the counters and RAM into the **LCD Data Latch** takes longer than 240 ns. Latching the address provides a stable signal for the entire LCD phase of **Dot Clock**, while delaying the RAM's output data by one cycle.

The RAM address alternates between the **LCD Address Counter** and the PC's ISA bus address. The LS590 latch outputs become active with **+Dot Clock** low, while U46 and U47, the **PC Address Buffers**, go active when **-Dot Clock** is low. Figure 1 shows the sequence of events; note how the LS590 latch ensures that the RAM address remains valid while **-Clear Counter** resets the **LCD Address Counter**.





Schematic 2

The ISA bus and the LCD Address Counters gain access to the LCD Refresh RAM on alternate Dot Clock phases. The -LCD Address Match signal, produced by a 74F521 comparator similar to the one shown in Chapter 6, identifies ISA bus accesses to the LCD Refresh RAM. The Dot Clock signal controls the three-state buffers driving the RAM's address lines. The LS139 decoder generates a pair of signals that depend on the Dot Clock phase and control other parts of the circuit.

The **LCD Data Latch**, U48, captures the **LCD Refresh RAM**'s output data when **+Dot Clock** goes high, 240 ns after the LCD's half cycle begins. This ensures stable data at the LCD panel when **+Dot Clock** goes low 240 ns later. The panels have various timing specs, with some requiring at least 150 ns of setup and hold time. Without this latch, the data at the LCD panel would disappear long before the panel captured it.

Off by One...

Pop quiz!

1. According to the LCD panel specs, when does the **Frame Sync** pulse occur in relation to the first row of data?
2. In Figure 1, what is the RAM address of the value in the **LCD Data Latch** with **Frame Sync** active and what part of RAM appears on the top line?
3. What Zen koan does this situation suggest?

OK, keyboards down. The answers:

1. After 160 **Dot Clocks**, at the *end* of the first row of panel data.
2. 31999. The top line shows the bottom of the RAM data.
3. Ah, Grasshopper, consider the sound of one hand clapping... forehead. At least, that's the sound you'd hear if you hadn't drawn Figure 1 first.

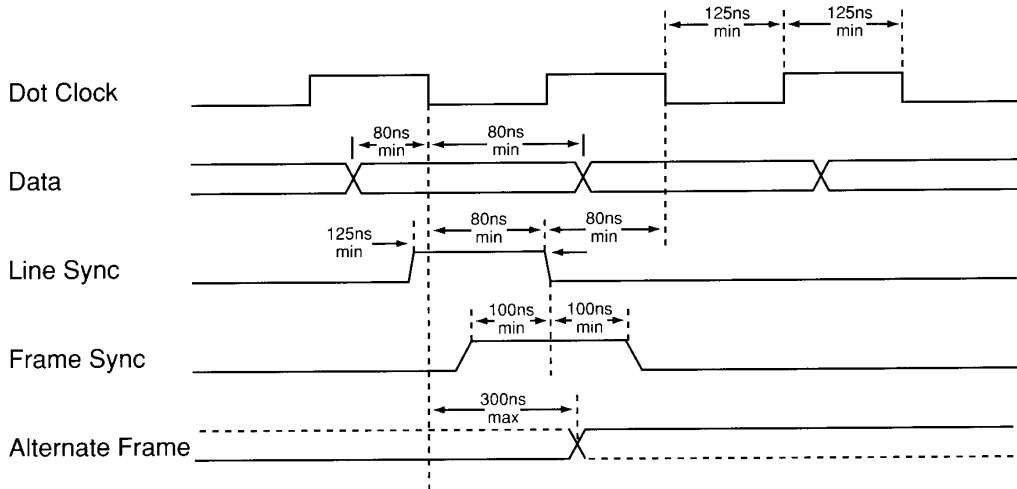
Figure 2 presents the Optrex DMF651 640×200 panel pulse timing diagram, redrawn with my nomenclature. You will see three vital synchronizing relationships: **Line Sync** *vs.* **Dot Clock**, **Frame Sync** *vs.* **Line Sync**, and **Alternate Frame** *vs.* both **Dot Clock** and **Frame Sync**.

The falling edge of **Line Sync** must be more than 80 ns away from a falling **Dot Clock** edge. U37B delays the 82C54's **-Raw Line Sync** output until the next rising **Dot Clock**. Figure 1 shows that **+Line Sync +0.0** rises when the last data byte in each row appears in the **LCD Data Latch** and falls on the first byte of the next row.

Frame Sync must be high for at least 100 ns on either side of the first **Line Sync** falling edge in the frame. U37A synchronizes **-Raw Frame Sync** to the next rising **+Dot Clock** and U57 delays it by half a **Dot Clock** cycle. Figure 1 shows the result: the pulse called **+Frame Sync +0.5** lies neatly centered on the *falling* edge of **+Line Sync +0.0**, with about 240 ns setup and hold times. The other outputs of U57 delay **-Raw Frame Sync** even more, should your panel require a later signal. If not, replace U57 with a single flipflop.

Figure 2

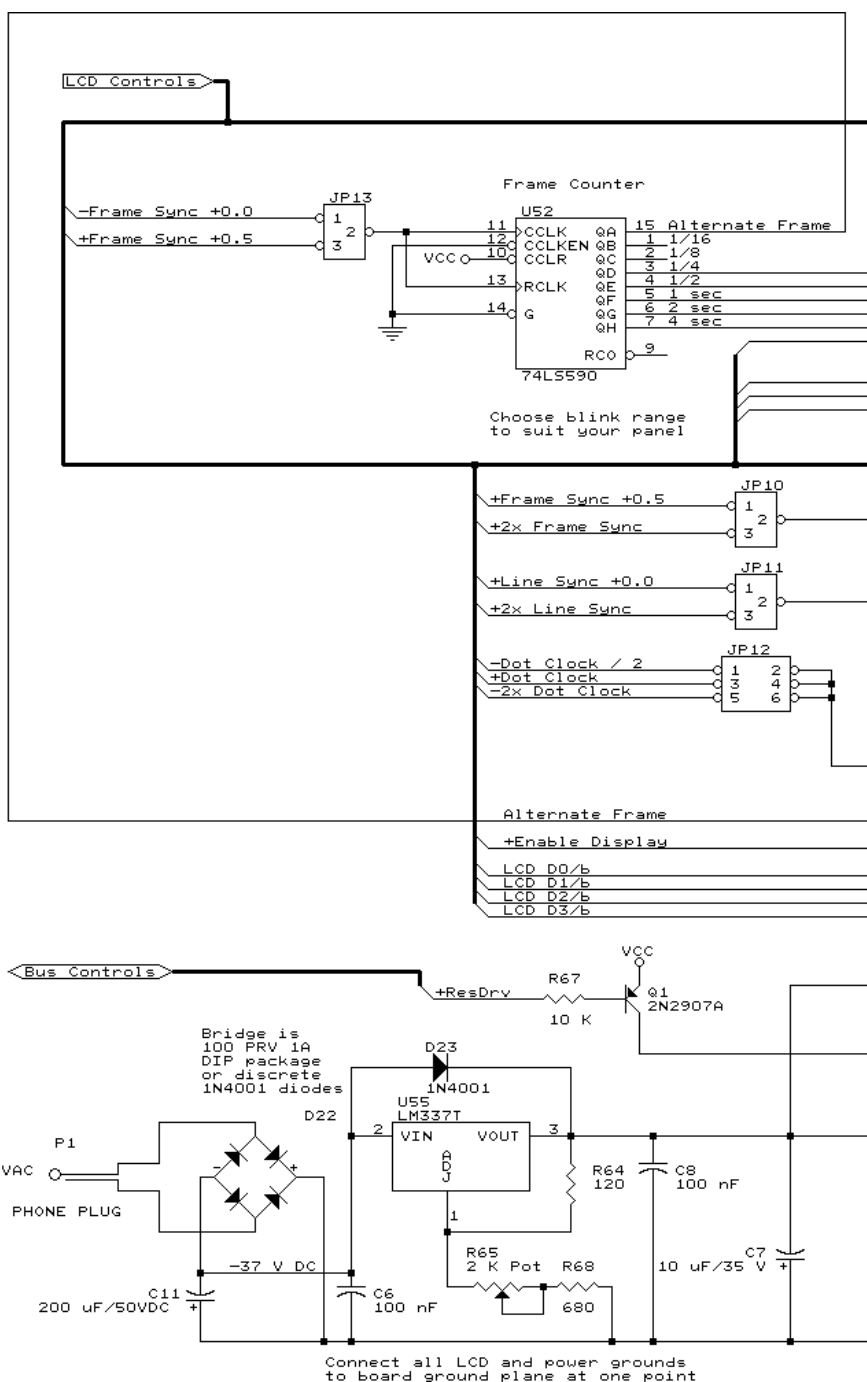
The Optrex DMF651 signal timing specifications dictate how the Graphic LCD Interface must operate. Notice that both data and Line Sync must be valid on the falling edge of each Dot Clock pulse, but Frame Sync's timing is related to the falling edge of Line Sync.

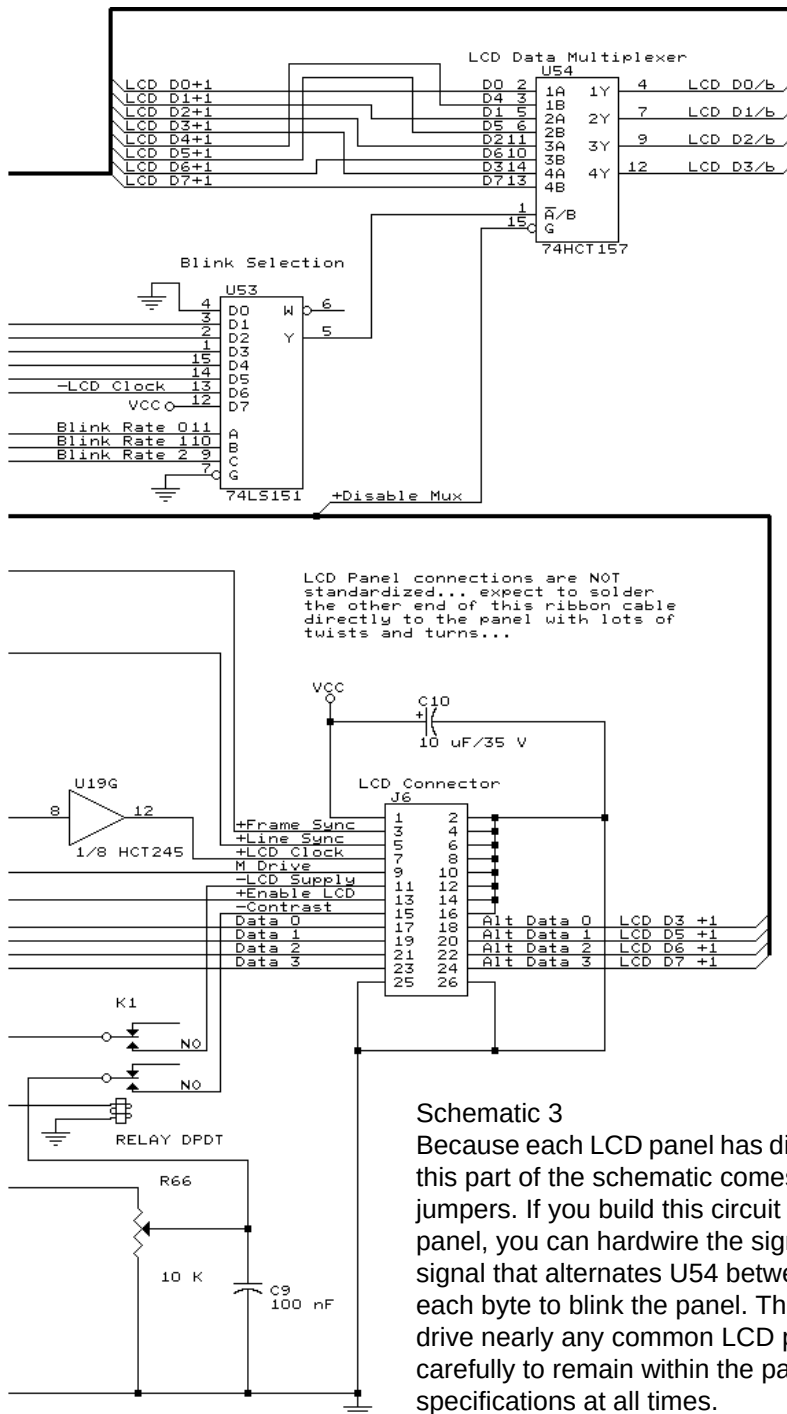


The **Alternate Frame** signal comes from U52 in Schematic 3. For the DMF651, this signal must change state within 300 ns of the **Dot Clock** falling edge at each **Frame Sync**, so I clocked U52 with **+Frame Sync +0.5**. Because the Hitachi LM215 refers **Alternate Frame** to the falling edge of **Line Sync** rather than **Dot Clock**, a jumper allows you to select **-Frame Sync + 0.0** instead.

U41 and U42 generate sync signals for panels that accept data every 240 ns. The double-speed **2x Dot Clock** changes the allowable width and alignment of the sync signals, leaving the basic Graphic LCD Interface hardware unchanged. The firmware sets up the **LCD Data Multiplexer** to switch between the two nybbles every half **Dot Clock** cycle and provide the data bits on time. I'll go into more detail about the blinking functions later.

As I mentioned earlier, you must drive all the LCD signals with CMOS logic gates to meet the panel's minimum V_{IH} spec. Because I used several spare gates from other sections of the FDB's logic, some of the LS gates mentioned earlier in this book metamorphosed into HCT. In actual practice, LS gates will probably work, if only by the grace of wide tolerances... which is bad practice.





Schematic 3

Because each LCD panel has different requirements, this part of the schematic comes festooned with jumpers. If you build this circuit to drive a particular panel, you can hardwire the signals. U53 selects a signal that alternates U54 between the nybbles of each byte to blink the panel. The power supply can drive nearly any common LCD panel. Adjust it carefully to remain within the panel's voltage specifications at all times.

Porting a RAM

When **+Dot Clock** is high, the PC controls the **LCD Refresh RAM**. The outputs of U46 and 47, the **PC Address Buffers**, become enabled during that half cycle. U49, another LS245, shuttles PC data to or from the RAM. These buffers run in series with those already on the FDB.

Access from the PC side of the interface encounters a gotcha: the LCD's **Dot Clock** cycle is not synchronized with ISA bus read and write cycles. If **+Dot Clock** goes low near the end of a bus cycle, the CPU will read invalid data or write junk into the RAM as the PC's buffers turn off prematurely and the LCD circuitry takes control. Obviously, this is a Bad Thing.

Solving this problem requires a delay: either the PC must wait until **+Dot Clock** goes high or the LCD panel must wait until the ISA bus cycle finishes. I chose the latter course, because you can pause an LCD clock for a short time without visible effect. You can do it the other way if you prefer, bearing in mind that the ISA bus runs slowly enough already and more delay might affect your program's ability to update the entire buffer in a timely manner. Decide which is more important in your application, then measure the results to be sure it does what you want.

The multiple exposure traces in Photo 1 show how this works. The top trace, **-SMemW**, controls the ISA write cycle timing. The lower trace, **+Dot Clock**, normally appears as a 480 ns square wave. Notice that **+Dot Clock** is always high when **-SMemW** rises, even though it may not be synchronized when the cycle begins. Normal **Dot Clock** cycles resume after **-SMemW** goes high at the end of the bus cycle.

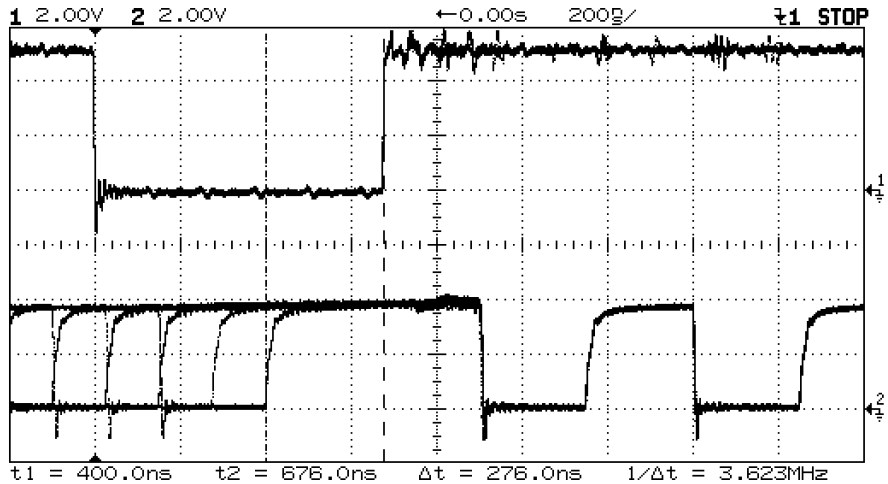
A complete 8-bit ISA bus memory write access requires six **SysClk** cycles, with **-SMemW** active for about the last 4.5 cycles. Because **Dot Clock** occupies four **SysClk** cycles, the maximum delay until **+Dot Clock** goes high after **-SMemW** becomes active will be three **SysClk** periods.

U56A monitors the double-speed **2x Dot Clock**, **Dot Clock**, and the (negative active) OR of **-SMemW** and **-SMemR**. When **+Dot Clock** goes high during a PC access, U56A's output freezes the **SysClk** divider flipflops and holds them unchanged until the PC access finishes.

The shortest RAM access thus occupies about 1.5 **SysClks**. Allowing for buffer and logic delays, a 120 ns RAM is just fast enough. I suspect some systems may require a 100 ns RAM or 74F buffers instead of 74LS parts. In any event, work out your system's timings and see how close it comes.

Photo 1

An ISA bus read or write access stalls +Dot Clock in the high state until the operation completes. The top trace, -SMemW, stays active for about 4.5 SysClk cycles. Dot Clock, shown in the bottom trace, is not synchronized with -SMemW when the write cycle begins, so it may end at any of four times after the falling edge of -SMemW. The RAM access starts when +Dot Clock goes high and ends when -SMemW goes high, allowing enough time for a complete LCD Refresh RAM access, even from the latest Dot Clock.



At worst, you may need an additional wait state or two on each PC access. Don't make the mistake of stalling both **Dot Clock** and the ISA bus, though... if *both* sides of the interface stop, *neither* will restart!

Even though you can stretch a **Dot Clock** cycle by up to three **SysClks** when it collides with a PC access, you'll never see it as long as it's rare enough. One of my test routines writes data into the RAM while toggling a parallel port bit that provides scope sync: one byte every 33 μ s affects an invisible 1.5% of all **Dot Clock** cycles.

On the other hand, another test routine runs a REP MOVSB that reads all 32 KB at 960 ns/byte. That *severely* distorts the **Dot Clock** cycle, holding it high for 720 ns and allowing one 240 ns low half cycle between each access. The refresh rate drops to 33 Hz and the panel develops an obvious flicker.

But remember, even if you read or wrote the entire 32 KB with a REP MOVSB, you wouldn't do it in a loop, as I did in the test routine. The LCD panel's optical

response falls in the 100–200 ms range, so it doesn't have time to respond to 30 ms of furious writes. Those new dots take another hundred milliseconds or so to replace the old ones. You'll see both types of access in Chapter 15, where we will generate dots in the buffer one by one, then copy the entire buffer into system RAM with a block move.

Do note, however, that this interface is *not* suitable for DMA or busmastering ISA bus board access, because the timings depend on a specific relation between **SysC1k** and **-SMemW** that will be valid only during system board accesses. If your project demands a more versatile interface, check Solari's books for the grim details.

Even when all your logic works correctly, you won't see anything on the LCD panel unless you apply the proper LCD drive voltages. I won't mention how often I've suffered palpitations at the sign of a blank panel, only to realize I left the power supply lead dangling once again.

Practical Power

The power supply circuitry shown in Schematic 3 certainly counts as overkill. Because any particular panel draws perhaps 20 or 30 mA at a single voltage, there's obviously no need for a wall wart transformer, a huge capacitor, and a 1.5 A, 15 W voltage regulator in a TO-220 case. Suffice it to say you can find easier and more compact ways to generate drive voltages for any specific LCD panel. Check the Maxim data books and Web page for hints and tips about their line of single-chip LCD bias supplies.

But, when you're working with a variety of panels, it makes perfect sense to build a versatile supply using parts from your junk box. Even better, you can probably find a power supply designed for precisely this purpose from any of the surplus parts dealers listed in the Sources appendix... at an attractive price, too.

Relay K1 marks my concession to the variety of power supply sequencing specifications found in the data sheets. All of the panels I've seen remain happy if you apply +5 V first and remove it last, which the DPDT relay accomplishes by disconnecting the drive voltages. The ISA bus **+ResDrv** signal goes active when the power supplies fall out of tolerance or you reset the system. That ensures the +5 V supply always becomes active first and remains active last.

CAUTION! Set the LCD drive voltage *before* you connect the panel, then monitor it with a voltmeter as you adjust the trimpot. The supply shown in Schematic 3 can apply enough voltage and current to destroy a panel with the twist of a screwdriver. Be overly hypercautious here, if nowhere else.

Greater & Lesser Arrays

The circuitry I've just described can handle panels with 480 ns **Dot Clock**s. The situation will be slightly different for panels requiring 240 ns or 960 ns clocks, but the hardware requires few changes. You can build only the sections for your panel and omit the jumpers festooning my prototype board.

The Sharp LM64015T 640×400 panel I mentioned in the previous chapter expects 320 data transfers on each of 200 rows, synchronized with a 240 ns clock signal. The **LCD Refresh RAM** still delivers eight bits every 480 ns, but the firmware sets up the **LCD Data Multiplexer** to switch between the two nybbles on each half of the **Dot Clock** cycle. As a result, the panel's dots line up quite nicely with the data bytes: the high and low nybbles come together at last.

The **Frame Sync** and **Line Sync** pulses last 240 ns and are synchronized by the double-speed **2x Dot Clock**. I used U41, an LS139 decoder, to control the two flipflops in U42 with the result shown in Figure 1, and drove the panel with **-2x Dot Clock** to place a falling edge in the middle of each nybble.

A Sharp LM641481 640×480 panel I found resembles the LM64015T, except that the extra 40 (not 80... remember how the panel hardware works?) rows reduce the refresh rate to about 54 Hz. Unfortunately, this high-performance panel has a brilliant cold cathode fluorescent tube backlight and flickers badly at 54 Hz. As with the LM64015T, the CCFT inverter seems particularly hard to find from the usual surplus sources.

Also, remember that a 640×480 panel requires 38400 bytes of RAM, which exceeds the standard Graphic LCD Interface design. Although it's not shown on the schematic, you can piggyback two 32 KB RAMs with their respective **-CE** inputs driven by **+LCD A15** from U44 and **-LCD A15** from an inverter. Disconnect ISA bus line **A15** from the F521 comparator to grant the PC access to all 64 KB between D000:0000 and D000:FFFF.

In contrast to that vast array of dots, the Hitachi LM215 480×128 panel is, electrically, a quartet of dinky 240×64 (!) arrays. Four data bits arriving every 960 ns refresh it at about 68 Hz. Each data bit drives a separate quadrant, with 240 half-speed **Dot Clock/2** pulses on each of 64 rows. Use the **-Dot Clock/2** jumper and run the **Alternate Frame** divider from **+Frame Sync + 0.5**.

The LM215 displays only 15360 nybbles of data, so you can use the high order nybbles in the **LCD Refresh RAM** for blinking as usual. Remember that, because the panel runs at half speed, you must duplicate the dots in successive even-and

odd-numbered bytes for the first 30720 RAM locations. The test pattern for this panel looks somewhat different than the others in order to cover all four quadrants.

You will certainly find a different assortment of panels today than I did when I set out to design this project. The most significant change comes from the PC laptop market, where 640×480 panels are now pretty much standard equipment. As a result, the prices of those panels have dropped rapidly and they're no longer nearly as exotic as they once were. Make sure that you get the specs for any panel you purchase, particularly if you go the surplus route, because you cannot figure out many of the essential connections without the documentation.

Release Notes

Reminder! Be careful with that LCD power supply. Crosscheck the LCD pin numbering with an ohmmeter: the supply voltages and grounds should appear on the panel's filter caps with the appropriate polarities, right? As woodworkers say, think thrice, measure twice, and cut once... smoking a new panel can really mess up the rest of your day.

14 Testing the Graphic LCD Panel

Although Steve Ciarcia still names “Solder” as his favorite programming language, even he admits most projects nowadays are useless without some programming. Now, with the molten metal part of the Graphic LCD Interface finished, it’s time for the bit twiddling!

I’ll describe the test code that exercises my panels and explain some tricks and techniques. Although Micro-C predominates in this chapter, you’ll surely want assembler code once you understand how your panel works. If pushing dots around doesn’t make a good case for hand tweaked optimization, well, what does?

But first, let’s check out your wiring. Because this project has lots of fairly tricky hardware, I’m going to spend more time than usual on bringup testing. With any luck at all, your circuitry and LCD panel will work perfectly the first time. If they don’t, these hints may come in handy.

Testing...

The `GraphLCD` files include all the Graphic LCD Interface test code. You can use the `BIN` file directly or, if you must make changes for your panel, recompile the Micro-C source into an Intel `HEX` file and convert it into binary. Copy the result to a diskette with the appropriate boot sector loader from Chapter 1, connect a serial cable, pop the floppy into your target system, hit the Reset button, and select the appropriate option when `GraphLCD` squirts its menu through the serial port to the comm program running on your host PC.

If you prefer Borland or Microsoft C and have Paradigm’s `Locate` (or a similar utility) available, you can translate the Micro-C code into their dialects fairly easily, using the hints in Chapter 11. In any event, don’t despair: the hardware neither knows nor cares which C you use.

You’ll recall that the CPU’s view of the Graphic LCD Interface includes one write-only output port, two 82C54 timer channels, and a 32 KB block of RAM. `GraphLCD`’s first four tests wiggle the control port bits, give you manual control over the 82C54, and read and write the **LCD Refresh RAM** in loops.

These tests work correctly even without an LCD panel, allowing you to start wiring and testing while waiting for your new panel to arrive. If you *do* connect a panel, remember that it won’t display anything without the correct sync signals and jumper settings. I *strongly* recommend delayed gratification: keep that panel in the box, safely wrapped in its antistatic bag, until your hardware passes its tests!

First, verify that all eight bits in U51, the **LCD Control Latch**, go ON and OFF in the proper order. As it turns out, I managed to wire the port backwards and this simple test prevented some serious headscratching.

Next, set the 82C54 row and frame counters. With no LCD panel attached (right?), you can specify three clocks per row and nine clocks per frame to simplify your scope display. Each row must have at least two clocks and, of course, the number of clocks in each frame must be a multiple of the row clock count.

Scope the **LCD Address Counter**, U43 and U44, to verify its inputs and outputs. The LS590 latch outputs go active (well, they *should* go active) only during the low half of the **+Dot Clock** cycle. The **Frame Sync** pulse resets the counters with a signal from U56B, so check that the count begins with zero at the right time. Refer back to Figure 1 in Chapter 13 for guidance.

The RAM read and write tests toggle printer port bits at key points in the loop to provide oscilloscope triggering. The read test uses a single **REP LODSB** to touch every **LCD Refresh RAM** byte and produces one sync pulse at the start of that lengthy instruction. The write test touches each RAM byte in a C loop and, as this requires much more time, issues a sync pulse before every RAM access. In either case, U56A should stall **+Dot Clock** in the high half of its cycle at the end of each **-SMemR** or **-SMemW** pulse.

The fifth test generates and writes a 32 KB pseudorandom pattern into the **LCD Refresh RAM**, reads the pattern back, and verifies that the bytes return from their journey unchanged. While this isn't an exhaustive RAM test, it will reveal obvious wiring errors, short circuits, and dead chips. If you don't have the equipment required for the other tests, just run this one and hope it works. Should it fail, you now have a problem that merits investigation and, perhaps, some motivation to acquire that collection of test equipment you've always wanted.

The RAM test may reveal intermittent data errors if your **LCD Refresh RAM** or buffers run too slowly for the ISA bus accesses. Remember that the RAM access must occur in slightly less than half of a **Dot Clock** cycle, about 240 ns. My numbers indicate that a 120 ns RAM works OK with my hardware, but your mileage may differ. In any event, don't install your LCD panel until the RAM works perfectly: it's hard to find other bugs when you can't trust your test data.

Once your bits fly in formation, check and recheck the LCD power supply voltages. If you're using the LM337 circuit shown in Chapter 13, remember that you can destroy your panel with one twist of R65. Set the LCD bias supply voltage to match the value in your panel's data sheet. If your panel requires a contrast voltage, also adjust R66 to midrange.

Next, build a cable that mates the Graphic LCD Interface's 2×13 header to your panel. If you can find a connector for your panel, great. Otherwise, just solder the appropriate ribbon cable wires directly to the panel's header, as you saw in Photos 1 and 2 in Chapter 12, and be done with it. I generally tape the cable to the back of the panel while testing the hardware. You can surely come up with a better, more permanent arrangement for your final installation.

If you do find an LCD backlight inverter, wire it to the panel through a separate cable. Don't succumb to the temptation of running that 1 kV, high frequency, AC power through the same cable as your precious sync and data signals. Just say "No!"

The final step requires nothing more than a simple ohmmeter continuity check: verify the connections from the Graphic LCD Interface circuitry to the LCD panel. You have only one last chance to get the power supply voltages on the right pins: don't blow it now.

Refer back to Photo 1 in Chapter 12, where you'll see that the Optrex DMF651 640×200 panel sports a helpful, silk screened legend on the upper left corner of the circuit board:

UP ↑

All of the other panels in my collection assume you can derive this key datum from their documentation. Alas, the doc you get with the panel sometimes won't give even the barest hint as to which end is up. In that case, the remaining tests will not only help you debug the hardware, but get the panel bolted down properly.

OK, plug it in and let's do some dots...

...Testing! One...

Perhaps the first thing you asked about a new graphic LCD panel was "How big is it?" From the number of dot rows and columns, you can calculate the required size of the **LCD Refresh RAM** in bytes. The DMF651 is a 640×200 panel that displays 128,000 bits from 16,000 bytes (not quite 16 KB) of data.

The Graphic LCD Interface measures LCD panels somewhat differently. Rather than rows and columns, both you and the hardware must know the number of **Dot Clocks** in each row, the total number of **Dot Clocks** in a complete frame, and the number of data bits transferred on each **Dot Clock**. As you saw in Chapter 13, though, there may be only the slightest relation between a panel's number of dots and the signals that control it.

For example, the DMF651 accepts four bits on each **Dot Clock** and, thus, transfers each 640-dot row in 160 clocks. The panel has 200 row drivers, one for each physical row, and a frame requires 32,000 **Dot Clock** cycles. The dots occupy one nybble in each of the first 32,000 **LCD Refresh RAM** bytes.

The test code in Listing 1 initializes the Graphic LCD Interface hardware for a DMF651 panel, then writes the test pattern into the **LCD Refresh RAM**. Listing 2 defines the **LCD Control Port** bit patterns used in the code. The code for

Listing 1

The Optrex DMF651 640x200 LCD panel has a comparatively simple layout: a single frame requires 200 sets of 160 Dot Clocks, each transferring four bits. This code fragment shows how to load the 82C54 timers and set the LCD Control Port. The SetData routine puts four data bits in the low nybble and creates the right bits for the high nybble to produce the desired blinking effect.

```
outpw(GLCD_CTL5,GLCD_OFF);          /* disable clocking      */
LoadTimer(1,0x04,160,I8254_BASE);    /* clocks per row        */
LoadTimer(2,0x04,32000,I8254_BASE);  /* total clocks/frame    */
outpw(GLCD_CTL5,GLCD_ON + GLCD_BLMED); /* start and set blink   */
for (Row = 0; Row < 200; Row++) {
    RAMAddr = Row * 160;
    for (Col = 0; Col < 160; Col++) {
        poke(GLCD_SEGMENT,RAMAddr+Col,0x00); /* clear row            */
    }
    SetData(RAMAddr+0,BLINK_OFF,Row >> 4); /* show row number      */
    SetData(RAMAddr+1,BLINK_OFF,Row);
    SetData(RAMAddr+2,BLINK_FG,0x09);
    SetData(RAMAddr+3,BLINK_OFF,RAMAddr >> 12); /* dividing line        */
    SetData(RAMAddr+4,BLINK_OFF,RAMAddr >> 8); /* show RAM address     */
    SetData(RAMAddr+5,BLINK_OFF,RAMAddr >> 4);
    SetData(RAMAddr+6,BLINK_OFF,RAMAddr);
    SetData(RAMAddr+7,BLINK_FG,0x09); /* dividing line        */
    SetData(RAMAddr+8+(Row >> 2),BLINK_BG,0x08 >> (Row & 0x03)); /* diagonal */
}
```

Listing 2

These definitions combine useful LCD Control Port bit patterns into easily remembered groups. You should change the GLCD_OFF and GLCD_ON values to match your panel.

```
#define GLCD_ENCTR      0x0001      /* enable addr counters  */
#define GLCD_DISMUX     0x0002      /* 1 disabled output mux */
#define GLCD_ENDISP     0x0004      /* enable LCD (if used)  */

#define GLCD_BLSLOW     0x0050      /* slowest blinking      */
#define GLCD_BLMED      0x0040      /* moderate blinking     */
#define GLCD_BLFAST     0x0030      /* fastest blinking      */
#define GLCD_BLZERO     0x0000      /* output logic zero     */
#define GLCD_BLCLOCK    0x0060      /* output LCD clock      */
#define GLCD_BLONE      0x0070      /* output logic one      */

#define GLCD_OFF        (GLCD_DISMUX | GLCD_BLZERO)
#define GLCD_ON         (GLCD_ENCTR | GLCD_ENDISP)
```

this chapter also produces several trace outputs which I've removed from these listings to save space; refer to the complete source files for all the details.

Recall that both 82C54 timer channels must begin counting on the same **Dot Clock** cycle. The first line disables the counters by lowering their **Gate** inputs. After loading the row and frame lengths, the code enables the 82C54 and sets the blink rate. The Graphic LCD Interface immediately begins sending data, **Dot Clocks**, and sync pulses to the panel, even though the **LCD Refresh RAM** contains no useful data.

Although it may seem strange to start the sync signals before clearing the RAM, that ensures you'll see *something* on the panel, even if the memory interface isn't working quite right. Should your panel display random bits instead of the test pattern, you know where to start looking for the problem. If the pattern changes, then you can assume the program tried and failed to write the test pattern.

The test loop then iterates 200 times, once for each row on the LCD panel. The rows occupy contiguous regions in RAM and, because each row uses 160 bytes of RAM, their starting addresses are 160 (decimal) times the row number: 0000,

Listing 3

Panels such as the Optrex DMF651 accept four bits per Dot Clock cycle, so the Graphic LCD Interface can produce blinking by switching between the two nybbles in each byte several times per second. This routine puts the data bits in the low nybble, then sets the high nybble to produce the desired blink pattern.

```
SetData(Addr,Blink,Data)
WORD Addr;
int Blink;
WORD Data;
{
    Data &= 0x000F;                /* force high nybble off          */
    switch (Blink) {
    default :
        printf("Invalid blink mode = %u\n",Blink);
    case BLINK_OFF :
        Data |= Data << 4;
        break;
    case BLINK_FG :
        break;
    case BLINK_BG :
        Data |= 0x00F0;
        break;
    case BLINK_INV :
        Data |= (~Data) << 4;
        break;
    }
    poke(GLCD_SEGMENT,Addr,Data);
}
```

00A0, 0140, and so on. I set **RAMAddr** to the beginning of the current row at the start of each iteration to simplify the rest of the code in the loop.

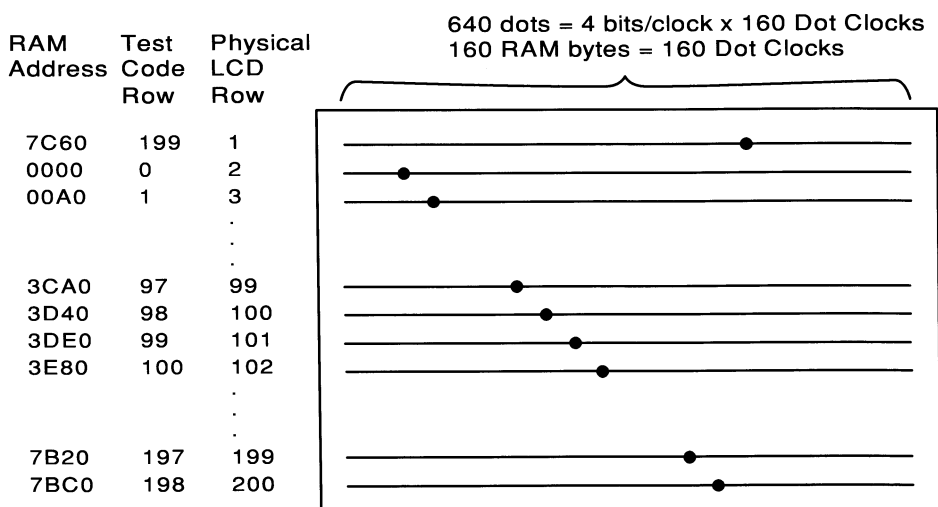
The test routines take advantage of the LCD's high dot density to display the row number and RAM address in binary format in the first few dozen dots of each row. If your eyes work like mine, expect to whip out a magnifying glass to read the dot patterns that give each line an unambiguous identifier. A diagonal line provides an easy visual check that all the rows appear in the right order.

Remember the sound of one hand clapping in Chapter 13?

Figure 1 sketches the row numbers, **LCD Refresh RAM** addresses, and the diagonal line for a DMF651. The test code numbers the top row 199 because it's the *last* one in the buffer... but the panel displays those dots on the *first* row because they *precede* the **Frame Sync** pulse. That puts one lonely dot off on the right of the top row, aligned above the end of the diagonal line.

Figure 1

The Optrex DMF651 has a fairly simple layout. The top row of dots appears to be out of sequence, because the test code numbers the visible rows starting with Row 0 at address 0000. The Graphic LCD Interface hardware produces a Frame Sync pulse when it resets the LCD Address Counters to 0000 and the data immediately before the pulse comes from the highest RAM addresses. The LCD panel displays the bits arriving before the Frame Sync pulse on the top row, which puts the test code's Row 199 on top.



Listing 3 shows the `SetData` routine that writes bit patterns into RAM. Although I defined only four blinking patterns, you can certainly come up with some truly bizarre effects. The initial **LCD Control Port** setup determines the blinking rate, as shown in Listings 1 and 2.

I started with the DMF651, because it's about as simple a panel as you're likely to encounter: the physical dot layout even matches the electrical sync signal pattern! Sad to say, such simplicity forms the exception rather than the rule. Many other panels can give you a nasty case of brain burn...

The Toshiba TLY-365-121, also a 640×200 panel, uses 320 **Dot Clocks** that transfer 1280 bits on each of 100 rows. The firmware sees it as a 1280×100 array and maps the dots accordingly. Listing 4 shows the test pattern code, with a few key differences in the 82C54 setup and `RAMAddr` calculation.

The loop still iterates 200 times, once for each visible dot row, but finding the row starting address becomes somewhat more complex. The first 160 bytes appear on one row, with the remaining 160 bytes starting 100 rows further down the panel. Row 99 is on top, with Row 199 in the middle of the panel. Ah, yes, do you remember row counts use *decimal* notation?

To prove I'm not making this up, Photo 1 shows the actual test pattern on the panel. If you can't see the dots clearly, Figure 2 sketches the layout.

Listing 4

The Toshiba TLY-365-121 has 1280 dots in each of 100 logical rows, with 320 Dot Clocks on each row. Compare the `LoadTimer()` parameters and `SetData` patterns with Listing 1 to see the effect of the double-length lines.

```
outpw(GLCD_CTL5,GLCD_OFF);                /* disable clocking      */
LoadTimer(1,0x04,320,I8254_BASE);          /* clocks per row        */
LoadTimer(2,0x04,32000,I8254_BASE);        /* total clocks/frame    */
outpw(GLCD_CTL5,GLCD_ON + GLCD_BLMED);     /* start and set blinking*/
for (Row = 0; Row < 200; Row++) {
    RAMAddr = 320 * (Row % 100) + ((Row > 99) ? 160 : 0);
    for (Col = 0; Col < 160; Col++) {
        poke(GLCD_SEGMENT,RAMAddr+Col,0x00); /* clear row             */
    }
    SetData(RAMAddr+0,BLINK_OFF,Row >> 4);  /* show row number       */
    SetData(RAMAddr+1,BLINK_OFF,Row);
    SetData(RAMAddr+2,BLINK_FG,0x09);        /* divider               */
    SetData(RAMAddr+3,BLINK_OFF,RAMAddr >> 12); /* show RAM address      */
    SetData(RAMAddr+4,BLINK_OFF,RAMAddr >> 8);
    SetData(RAMAddr+5,BLINK_OFF,RAMAddr >> 4);
    SetData(RAMAddr+6,BLINK_OFF,RAMAddr);
    SetData(RAMAddr+7,BLINK_INV,0x09);        /* divider               */
    SetData(RAMAddr+8+(Row >> 2),BLINK_BG,0x08 >> (Row & 0x03)); /* diagonal */
}
```

Photo 1

The test pattern on this Toshiba TLY-365-121 640x200 LCD panel is based on Figure 2. The first several dozen dots in each row display the test code's row number (0 through 199) in binary, the row's RAM address, and a diagonal line. The code also inserts blinking separator patterns between the data values (which, of course, appear solid in this photograph).

You may not be able to see the two "stray" dots on the top and middle rows of the panel. Trust me: they're visible on the original slide and you'll see them on your own panel!

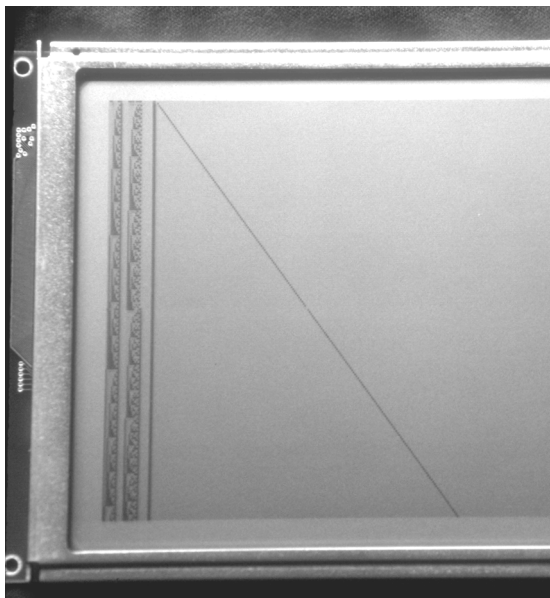
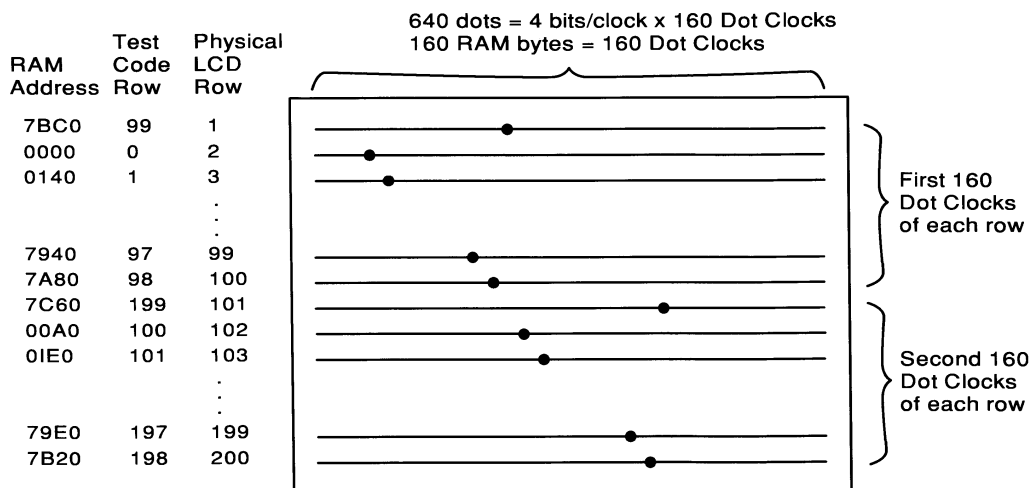


Figure 2

The Toshiba TLY-365-121 splits each logical row of 320 Dot Clocks in two: the first 160 clocks transfer 640 dots to the upper half of the panel, while the second 640 dots appear in the lower half. Therefore, the 320 Dot Clocks before the Frame Sync pulse appear on physical rows 1 and 101. Photo 1 shows this test pattern in real life.



... Two...

As I mentioned in the previous chapter, LCD designers have recourse to only two methods that provide the increased bandwidth required by 400-line panels: send more dots per clock or stuff more clocks into each frame. The Graphic LCD Interface can handle either method, provided that you set up the right bits and jumpers before turning the power on.

The Matsushita EDM LG64AA44D accepts eight data bits on each **Dot Clock** cycle. The **LCD Control Port** value sends a constant zero to the **LCD Data Multiplexer**, forcing it to gate only the low nybble to the panel. The high nybble connects directly to the panel, presenting the entire **LCD Data Latch** on each **Dot Clock** cycle. Listing 5 shows the test code.

As the loop iterates through all 400 visible lines, **RAMAddr** goes through the same set of 200 addresses twice. With no spare bits for blinking, the **StoreNybble**

Listing 5

The Matsushita EDM LG64AA44D panel has 640x400 physical dots, but the firmware sees it as two 640x200 panels. The test code loops through all 400 rows and sets **RAMAddr** to the start of each one. The value of **Align** selects the appropriate nybble and the **StoreNybble** function inserts the data bits. Because the row number exceeds 255 for this panel, the firmware displays it with 16 dots.

```

outpw(GLCD_CTL5,GLCD_OFF);                /* disable clocking      */
LoadTimer(1,0x04,160,I8254_BASE);          /* clocks per row        */
LoadTimer(2,0x04,32000,I8254_BASE);        /* total clocks/frame    */
outpw(GLCD_CTL5,GLCD_ON | GLCD_BLZERO);    /* get bits 0-3 from mux */

for (Row = 0; Row < 400; Row++) {
    RAMAddr = 160 * (Row % 200);            /* buffer address        */
    Align = (Row >= 200) ? 4 : 0;          /* high/low bit align    */

    for (Col = 0; Col < 160; Col++) {
        StoreNybble(RAMAddr+Col,Align,0);  /* clear our nybble      */
    }
}
for (Row = 0; Row < 400; Row++) {
    RAMAddr = 160 * (Row % 200);            /* buffer address        */
    Align= (Row >= 200) ? 4 : 0;            /* high/low bit align    */

    StoreNybble(RAMAddr+0,Align,Row >> 12);
    StoreNybble(RAMAddr+1,Align,Row >> 8);
    StoreNybble(RAMAddr+2,Align,Row >> 4);
    StoreNybble(RAMAddr+3,Align,Row);
    StoreNybble(RAMAddr+4,Align,0x08);      /* divider lines        */
    StoreNybble(RAMAddr+5,Align,RAMAddr >> 12);
    StoreNybble(RAMAddr+6,Align,RAMAddr >> 8);
    StoreNybble(RAMAddr+7,Align,RAMAddr >> 4);
    StoreNybble(RAMAddr+8,Align,RAMAddr);
    StoreNybble(RAMAddr+9,Align,0x06);      /* divider lines        */
    StoreNybble(RAMAddr+10+(Row >> 2),Align,0x08 >> (Row & 0x03));
}

```


function shown in Listing 6 inserts four data bits in the proper half of the byte, as dictated by the `Align` variable.

The Sharp LM64015T panel uses a double-speed 240 ns **Dot Clock**, selected by jumper JP12. The **LCD Refresh RAM** runs at the same 480 ns rate as the other panels, so the code in Listing 7 routes **Dot Clock** to the **LCD Data Multiplexer**, switching it between the two **LCD Data Latch** nybbles every 240 ns. The high nybble goes out in the first half cycle, which, conveniently enough, lays the dots out left to right on the panel.

Because only the LM64015T sees the 240 ns **2x Dot Clock**, the rest of the Graphic LCD Interface thinks the panel has just 160 ordinary rows with 480 ns **Dot Clock** pulses apiece and 32000 clocks per frame. The timing remains the same as the DMF651, but you must disable blinking because all the bits in each byte contribute to the normal display. JP10 and JP11 select the double-speed sync signals that stay active for only half of the normal **Dot Clock** cycle.

Unlike the other panels, the LM64015T turns a 1 bit into a transparent dot, as the panel starts off opaque and the backlight shines through the ON bits. I find this disconcerting, but you can easily display dots either way. The test pattern uses dark data dots to show the technique: note that the lighted dots each have a dark border.

The LM641481 640×480 panel resembles its 400-line ancestor, although, as you saw in Chapter 13, you must enlarge the **LCD Refresh RAM** to hold all the new dots. `GraphLCD` includes a test pattern for this panel and others like it, assuming that you build the hardware for it. Running with a 32 KB RAM will duplicate some of the dots, because a 15-bit address repeats the RAM contents on the higher rows. Refer back to Chapter 13 for the hardware solution.

Listing 6

The LG64AA44D accepts twice as much data per frame through its 8-bit interface as 200-line panels. This code fetches a byte from the buffer, clears the selected nybble, inserts the new bits, and then stores the result back. Because ISA memory runs quite slowly, you should use a duplicate buffer in system RAM to eliminate the extra read.

```
storeNybble(Addr, Align, Data)
WORD Addr;
int Align;
WORD Data;
{
    WORD oldData;

    oldData = peek(GLCD_SEGMENT, Addr) & (0xf0 >> Align);
    Data = ((Data & 0x0f) << Align) | oldData;
    poke(GLCD_SEGMENT, Addr, Data);
}
```

Listing 7

The Sharp LM64015T runs at twice the speed of the other panels, accepting a nybble on each half cycle of the 480 ns Dot Clock. The LCD Data Multiplexer switches between the two nybbles of each Refresh RAM byte. A jumper on the Graphic LCD Interface routes the double-speed 2x Dot Clock to the panel. Note the inverted data: a 1 bit sent to the panel appears transparent rather than opaque.

```

outpw(GLCD_CTL5,GLCD_OFF);                                /* disable clocking      */
LoadTimer(1,0x04,160,I8254_BASE);                         /* Dot Clocks/row        */
LoadTimer(2,0x04,32000,I8254_BASE);                       /* total clocks/frame    */
outpw(GLCD_CTL5,GLCD_ON | GLCD_BLCLOCK);                  /* alternate at clk rate */
for (Row = 0; Row < 400; Row++) {
    RAMAddr = 160 * (Row % 200) + ((Row > 199) ? 80 : 0);
    for (Col = 0; Col < 80; Col++) {
        poke(GLCD_SEGMENT,RAMAddr+Col,0xff);              /* 1 = clear             */
    }
}
for (Row = 0; Row < 400; Row++) {
    RAMAddr = 160 * (Row % 200) + ((Row > 199) ? 80 : 0);
    poke(GLCD_SEGMENT,RAMAddr+0,~(Row >> 8));
    poke(GLCD_SEGMENT,RAMAddr+1,~(Row & 0x00FF));
    poke(GLCD_SEGMENT,RAMAddr+2,~0x99);                   /* divider lines        */
    poke(GLCD_SEGMENT,RAMAddr+3,~(RAMAddr >> 8));
    poke(GLCD_SEGMENT,RAMAddr+4,~(RAMAddr & 0x00FF));
    poke(GLCD_SEGMENT,RAMAddr+5,~0x99);                   /* divider lines        */
    poke(GLCD_SEGMENT,RAMAddr+6+(Row >> 3),~(0x80 >> (Row & 0x0007)));
}

```

... Four...

The Hitachi LM215XB 480×128 panel is a leopard of a different spot: it uses half the data, runs at half the speed, and sprinkles dots all over RAM in four quadrants. Listing 8 shows the rather complex test pattern code. Remember to select the half-speed **Dot Clock/2** in JP12 for this panel.

Each of its 64 logical rows requires 240 half-speed **Dot Clock/2** pulses, so the 82C54 behaves as though this panel has 480 **Dot Clock** cycles per row. The panel uses the data from odd-numbered RAM addresses, putting $2 \times 240 \times 64 = 30720$ pulses of the regular 480 ns **Dot Clock** in each frame, but we must duplicate the data in successive even- and odd-numbered bytes to keep the output of the **LCD Data Latch** stable during both regular **Dot Clock** cycles.

The test code iterates through all 128 visible rows. Displaying the row number requires the routines shown in Listing 9, because the layout separates the RAM data. The complexity involved in addressing the individual dots makes this code considerably harder to follow.

The `writeBit` function computes `RAMAddr` based on the dot's row and column location. The `Align` variable then specifies the bit location within that byte, which

Listing 8

The Hitachi LM215XB uses a half-speed 960 ns Dot Clock and directs the four data bits to separate quadrants. This code duplicates the dots in successive even and odd RAM addresses to keep the data stable during the extended clock cycle.

```
outpw(GLCD_CTL5,GLCD_OFF);          /* disable clocking      */
LoadTimer(1,0x04,480,I8254_BASE);    /* Dot clocks/row        */
LoadTimer(2,0x04,30720,I8254_BASE); /* total clocks/frame    */
outpw(GLCD_CTL5,GLCD_ON | GLCD_BLFAST); /* enable & set blinking */
for (Row = 0; Row < 64; Row++) {
    for (Col = 0; Col < 240; Col++) {
        RAMAddr = 2 * (240 * Row) + Col;
        poke(GLCD_SEGMENT,RAMAddr+Col,0x00); /* even address          */
        poke(GLCD_SEGMENT,RAMAddr+Col+1,0x00); /* odd address           */
    }
}
for (Row = 0; Row < 128; Row++) {
    RAMAddr = 2 * (240 * Row) + 1;
    StoreByte(Row,0,BLINK_OFF,Row);
    StoreByte(Row,8,BLINK_FG,0x81);
    StoreByte(Row,16,BLINK_OFF,RAMAddr >> 8);
    StoreByte(Row,24,BLINK_OFF,RAMAddr);
    StoreBit(Row,32+Row,BLINK_FG,1);

    StoreByte(Row,240+0,BLINK_OFF,Row);
    StoreByte(Row,240+8,BLINK_BG,0x7E);
    StoreByte(Row,240+16,BLINK_OFF,RAMAddr >> 8);
    StoreByte(Row,240+24,BLINK_OFF,RAMAddr);
    StoreBit(Row,240+32+Row,BLINK_FG,1);
}
```

obviously depends on how you wire the Graphic LCD Interface's data bits to the LM215 connector. The `Blink` parameter controls the corresponding bit in the high nybble.

`writeByte` simply calls `writeBit` for each bit in a byte-sized value. This process entails considerable overhead, but remains reasonably perky because the panel has so few dots... and things will certainly go better with assembler!

The LM215 test pattern displays the row and `RAMAddr` values in all four panel quadrants to verify all the data bits. If you have one of these panels, examine the dots (with a magnifying glass?) to see that the same RAM address appears four times, once in each quadrant, on the proper lines.

... and More!

That covers the interesting LCD panels in my stash, without coming close to exhausting the possibilities. You can probably adapt the Graphic LCD Interface to drive whatever panel you've got, although some are sufficiently bizarre that they just won't work. As a rule of thumb, the more complex the panel, the older it's likely to

Listing 9

The Hitachi LM215XB panel has a peculiar bit arrangement that makes these two routines far more complex than you'd expect for such a small panel. The low nybble of each odd-numbered RAM address holds the normal dots, the high nybble holds the blinking dots, and each bit drives a separate quadrant. The StoreByte function distributes each bit to the proper location, making each incoming byte seem contiguous in storage.

```

StoreBit(Row,Col,Blink,Data)
WORD Row;
WORD Col;
WORD Blink;
WORD Data;
{
WORD Temp;
WORD Mask;
WORD RAMAddr;
WORD Align;

    RAMAddr = 2 * (240 * (Row % 64) + (Col % 240)); /* even address */
    Align = ((Col > 239) ? 2 : 0) + ((Row > 63) ? 1 : 0);
    Mask = 0x0001 << Align;
    Data &= 0x0001;

    Temp = peek(GLCD_SEGMENT,RAMAddr + 1); /* fetch the old bits */
    Temp &= ~(Mask | (Mask << 4)); /* strip target & blink */
    Temp |= Data << Align; /* insert new data bit */

    switch (Blink) {
    default :
        putstr("Invalid blink mode in Storebit: %u\n",Blink);
    case BLINK_OFF:
        Temp |= Data << (Align + 4); /* no blink dupe data bit*/
        break;
    case BLINK_FG :
        /* zero blink bit, done */
        break;
    case BLINK_BG :
        Temp |= Mask << 4; /* set the blink bit */
        break;
    case BLINK_INV :
        Temp |= ((~Data) & Mask) << 4; /* inverse of data bit */
        break;
    }
    poke(GLCD_SEGMENT,RAMAddr ,Temp);
    poke(GLCD_SEGMENT,RAMAddr+1,Temp);
}

StoreByte(Row,Col,Blink,Data)
WORD Row;
WORD Col;
WORD Blink;
WORD Data;
{
    StoreBit(Row,Col ,Blink,Data >> 7);
    StoreBit(Row,Col+1,Blink,Data >> 6);
    StoreBit(Row,Col+2,Blink,Data >> 5);
    StoreBit(Row,Col+3,Blink,Data >> 4);
    StoreBit(Row,Col+4,Blink,Data >> 3);
    StoreBit(Row,Col+5,Blink,Data >> 2);
    StoreBit(Row,Col+6,Blink,Data >> 1);
    StoreBit(Row,Col+7,Blink,Data );
}

```

be: more recent panels have better LSI chips and simpler interfaces. Things just keep getting better and better!

Remember that you're limited to 32 KB of RAM, unless you piggyback another chip atop the **LCD Refresh RAM** or modify the circuit to use a 62512-type static RAM. If you find a panel requiring more memory than that, you'll probably wind up with a paged refresh buffer that fits within a single 64 KB memory segment. Writing the code for *that* should be quite a challenge...

Release Notes

The test code for this chapter, **GraphLCD**, exercises the Graphic LCD Interface hardware and generates test patterns for several common panels. You can use my code as a template to check out other panels with different specs.

Assuming that you can get the power supply and signal wires connected properly, **GraphLCD** can help you figure out the panel's sync requirements. The panels can tolerate incorrect signals (at least for a while), if you don't exceed their voltage specs, so try a few experiments. In fact, once you get a panel working, lie to **GraphLCD** just to see what the results look like.

CAUTION: even though the panels won't detonate if you apply the wrong sync signals, a DC bias will gradually degrade both the liquid crystal material and the panel's transparent electrodes. Turn the power off while you dope out the jumpers and analyze the clocking. Don't use incorrect signals longer than it takes you to realize that things just aren't working right.

So... if that doesn't get your juices flowing, you really are reading the wrong book. Heat up those soldering irons, get those compilers whirring, and let's see some dots!

15 Bringing the Graphic LCD Panel to Life

Have you ever repaired something by taking it apart, contemplating the pieces, then putting them back together again? It's hard to believe a simple laying-on of hands can fix an inanimate object, but quite often that's exactly what happens.

Textbooks ascribe such problems to contact corrosion, random glitches, metastability, software errors, and similar maladies. Experienced engineers and technicians know differently. "That widget just wanted some attention," they'll say with a smile and go on to the next problem. To hear them talk, you might almost think the hardware was alive.

So far, we've covered the Graphic LCD Interface's design, hardware, and the test firmware that exercised the circuitry. It's now time for the code that manipulates individual dots on the panel and, as an added bonus, puts your *x86* target system's extended memory to good use. You'll see several chunks of assembly language here, as we begin writing code where performance matters a bit more.

While the gadgetry isn't quite alive yet, we're getting there...

Putting Life to Work

Graphic LCD panels have a voracious appetite for data: testing the drawing routines means exercising a quarter-million dots in a 640×400 array. Just writing a byte into the RAM and reading it back correctly doesn't mean that the dots appear at the right place on the panel. You must actually *look* at the results to verify that the hardware is working correctly.

But... who wants to watch a test pattern?

Conway's venerable Game of Life is the great-to-the- n^{th} grandfather of today's Artificial Life creations. It produces an easily recognizable set of patterns, exercises the whole LCD panel, is fun to watch, and can be falling-off-a-log easy to code... if performance isn't much of an issue, anyway.

The playing field consists of a rectangular grid of cells, each of which may be either *alive* or *dead*. Ideally, the field should be infinite, but typical implementations surround the visible part of the field with permanently dead cells or join the four edges to simulate a torus. The contents of the initial field uniquely determine the course of the game, making Life a spectator sport rather than an interactive game.

Play proceeds by generation. In principle, all cells change state in one instant at the end of each generation. The state of each cell in the next generation depends on its current state and the state of its eight immediate neighbors, nine cells in all.

A dead cell remains dead unless it has exactly three living neighbors in the current generation, in which case it springs to life in the next generation. A live cell remains alive when exactly two or three of its neighbors are also alive, otherwise it dies of either loneliness or overcrowding. Variations on those simple rules are possible, but tend to produce less interesting patterns.

Listing 1 shows the few lines of code required for this algorithm in its most basic form. A little arithmetic reveals the prodigious computation required for a single generation on 640×400 LCD panel: $9 \times 640 \times 400 = 2.3 \times 10^6$ LCDGetDot calls! Because relatively few cells change state after the first few generations, the LCDSetDot calls that update the playing field don't influence the overall time.

My oscilloscope reports that the calculations take about 100 μ s per cell on a 33 MHz '386SX CPU. Therefore, computing just one generation requires

Listing 1

One of the attractions of Conway's Game of Life is the simplicity of the algorithm: these few lines determine the changed cells in the next generation. Because our graphic LCD panels have a quarter million dots, however, the complete computation can take tens of seconds even on a fast processor. This is fast enough to test our hardware, but you can have fun optimizing the code beyond recognition.

```
for (Row=0; Row<NumRows; Row++) {
    ByteToLEDS(Row);
    outpw(SYNC_ADDR,0x80);
    for (Col=0; Col<NumCols; Col++) {
        Neighbors = LCDGetDot(Row-1,Col-1) +
            LCDGetDot(Row-1,Col) +
            LCDGetDot(Row-1,Col+1) +
            LCDGetDot(Row,Col-1) +
            LCDGetDot(Row,Col) +
            LCDGetDot(Row,Col+1) +
            LCDGetDot(Row+1,Col-1) +
            LCDGetDot(Row+1,Col) +
            LCDGetDot(Row+1,Col+1);
        if (LCDGetDot(Row,Col)) {
            if ((Neighbors < 2) || (Neighbors > 3)) {
                LCDSetDot(Row,Col,0);
            }
        }
        else {
            if (Neighbors == 3) {
                LCDSetDot(Row,Col,1);
            }
        }
    }
    outpw(SYNC_ADDR,0x00);
}
```

$640 \times 400 \times 100 \mu\text{s} = 26$ seconds. Watching this process demands nearly the same patience as studying the life cycle of glaciers in real time. Smaller panels run faster, of course, while remaining in the icicle growth competition. Obviously, we have here a program where optimization pays off handsomely.

Michael Abrash presented a Game of Life Optimization Challenge in the June '92 *PC Techniques* magazine, with results appearing in the Dec '93 issue. The two winning entries run at 125 ns/cell (yes, *nanoseconds* per cell) on a 33 MHz '486 (yes, a doorstep system by contemporary desktop PC standards) and their source code is a wonder to behold. Before you start tweaking my simple code, please read those articles... and weep. The contest and results also appear in his *Zen of Code Optimization* book, the contents of which you should commit to memory.

Because I was more interested in LCD panels than the Game of Life, I decided to finesse the problem. I added a few routines that store each successive generation in the target system's extended memory, then play them back as a slide show. You'll get the details after we cover the LCD firmware.

Doing Dots

As you saw in Chapter 13, the Graphic LCD Interface maps the LCD's dots into the **LCD Refresh RAM**, a 32 KB chunk of the PC's address space. From the CPU's viewpoint, the LCD panel operates as a typical, albeit relatively slow, RAM on the ISA bus, with the exceedingly useful side effect that its bits are visible to the naked eye. Unfortunately, as we discovered in Chapter 14, each panel has its own dot layout and requires unique code to locate each bit in the RAM.

Finding a particular dot requires two steps: selecting the RAM byte holding the bit, then isolating the bit within that byte. Obviously, both steps depend on the dot's row and column address. To simplify the high level code, rows and columns both start with number 0 in the upper left corner of the panel, even though many LCD panel documents number them differently. Other mappings work equally well: if an upside down, backwards, or rotated mapping suits your application, go for it!

Because of the hardware we're using, the top row of dots on the LCD panel does not come from the first group of bytes in RAM. Although the Graphic LCD Interface produces a **Frame Sync** pulse when it resets the **LCD Address Counter** to zero, the panel expects **Frame Sync** to *follow* the first row. As a result, the dots starting at address 0000 appear on the *second* panel row, not the first.

This was not a problem in Chapter 14, as we were interested in verifying the hardware and showing which RAM addresses appeared on which rows. Now that we must work with specific dot positions, it's time to get precise.

The quick and easy firmware solution for this appears in Listing 2, where I stored the starting address of each row in the `RowStarts` table. That table has one entry for each visible row in the LCD panel: the LG64AA44D table has 400 entries. The first entry points to the bytes at address $199 \times 160 = 31840$ decimal = 7c60 hex, while the second entry holds address 0000. The dot drawing routines consult the table to find out where each row starts: the scrambled addresses unscramble the panel layout and put Row 0 at the top of the panel where it belongs.

I create and fill the table as part of an LCD initialization routine which is specific to each panel. Entries 0 and 200 are special-cased at the top of the loop, while the remaining 2×199 entries form a simple ascending sequence. I suspect you could define and fill the table using some assembler macro magic, but this way is easy enough and relatively straightforward to modify.

The `LCDMakeAddr` function in Listing 3 converts a dot's row and column address into a byte address and a bit shift amount. The byte address depends on both the row, which specifies the lookup table entry, and the column, which specifies an additional number of bytes relative to the start of the row.

Listing 2

A lookup table simplifies finding the RAM address of a particular dot by holding the starting address of its row. This code fragment defines and loads the 400 table entries for a Matsushita LG64AA44D 640x400 dot panel. The data for the lower half of the panel is located in the high-order nybble of the same bytes displayed on the upper half, so the second half of the table (Rows 200-399) contains the same addresses as the first half (Rows 0-199).

```
LCDNAME      EQU      "Matsushita EDM LG64AA44D"
NUMROWS      =        400
NUMCOLS      =        640
COLCLOCK     =        160
COLMODULUS   =        4
TOTALCLOCK   =        32000
ROWMODULUS   =        NUMROWS/2
FILLVALUE    =        000h

RowStarts    DW        NUMROWS DUP (?)          ; row starting addresses

              MOV       [RowStarts],(ROWMODULUS-1)*COLCLOCK
              MOV       [RowStarts+2*ROWMODULUS],(ROWMODULUS-1)*COLCLOCK

              MOV       DI,(OFFSET RowStarts) + 2
              MOV       CX,ROWMODULUS-1
              MOV       AX,0                    ; row 0 = second entry
              PUSH      DS                      ; set up STOSw segment
              POP       ES

@@1:          MOV       [DI+2*ROWMODULUS],AX      ; second half of table
              STOSW                               ; first half
              ADD       AX,COLCLOCK              ; offset of next row
              LOOP      @@1
```

Listing 3

This LG64AA44D panel routine converts a dot's row (in BX) and column (in AX) into the offset of a byte from the start of the buffer (in DI) and the shift amount (in CL) that moves that bit into Bit 0. The high nybble of each byte stores the dots for rows 200 through 399, the low nybble contains dots for rows 0 through 199, and, thus, the shift amount depends on which half of the panel holds the dot.

```

PROC      LCDMakeAddr

ADD       BX,BX          ; make word table index
MOV       DI,[RowStarts+BX]

DIV       [BYTE ColModulus]
XOR       AH,03h         ; flip shift direction
MOV       CL,AH          ; save remainder for shifting
MOV       AH,0
ADD       DI,AX          ; DI points to the byte

CMP       BX,2*ROWMODULUS ; rows >=200 use high nybble
JB        @@1
ADD       CL,4           ; ... so shift into it

@@1:

RET

ENDP      LCDMakeAddr

```

The bit shift value in CL moves the selected bit to the low-order bit or vice versa. Depending on which panel we're using, the shift amount may also depend on the row, the column, or both, because these LCD panels pack several widely separated dots together in each **LCD Refresh RAM** byte.

Homework assignment: step through the code samples with a pencil and paper to see how the algorithm works. Draw a picture of the LCD panel's bit layout, indicate the bit addresses at the start of each row, then verify that the code really does produce the right answers. You won't believe it until you do...

Listing 4 pulls all of this together to write a single dot. As you saw in Listing 1, the `main()` C routine calls `LCDSetDot` directly, which means that routine's register and stack usage *must* match C's expectations. Interfacing assembler with C used to be fiendishly tricky, but current PC assemblers include several high level directives to ease the task. I won't go into the details here, other than to say it's *much* easier than counting bytes and tweaking registers on your own.

Recall that `LCDMakeAddr` returns a byte offset in DI rather than a complete `seg:off` address. `LCDSetDot` adds the far pointer in `pNewBuff` to DI to create the final address in `ES:DI`. You can aim `LCDSetDot` at the actual **LCD**

Listing 4

This routine sets or clears a dot on the LG64AA44D LCD panel. The global variable `pNewBuff` contains a far pointer to the target buffer, which may be the LCD panel or a chunk of system RAM. Depending on your application, you may want to include the target buffer address as one of the function parameters. This code can be called directly from the main C program, because the first four lines handle the inter-language register and stack conversions.

```

PUBLIC  C_LCDSetDot
PROC   LCDSetDot
ARG    Row:WORD,Col:WORD,value:WORD
USES   ES,DI

MOV     AX,[Col]
MOV     BX,[Row]
CALL    LCDMakeAddr

LES     AX,[pNewBuff]      ; get buffer base pointer
ADD     DI,AX              ; ... update byte pointer

MOV     BX,0101h          ; set up bit mask
SHL     BX,CL
NOT     BL                 ; BL = AND mask, BH = OR mask

TEST    [value],0001h     ; what do we want?
JNZ     @@2               ; nonzero means set the bit
AND     [ES:DI],BL        ; ... clear it
JMP     SHORT @@3

@@2:
OR      [ES:DI],BH        ; ... set it

@@3:
RET

ENDP    LCDSetDot
```

Refresh RAM addresses or, for more speed, at a separate working buffer in system RAM, just by changing `pNewBuff`.

Normally, you would include the buffer base address in `LCDSetDot`'s arguments. I used a global variable to reduce the amount of stack shuffling, but I agree that global variables can cause problems in a real application. As always, use this code as the basis for your own efforts, rather than The Final Word on Programming Style.

The `BX` register holds the bit masks that set or clear the selected bit. Although only one instruction references the target byte in each pass through Listing 4, the CPU actually makes two memory accesses: one to read the current contents and another to write the updated dots back into the same byte. This instruction runs as fast as

the hardware can do it, but the PC Compatibility Barnacles make the **LCD Refresh RAM** on the ISA bus achingly slow compared to system board DRAM.

Listing 5 shows how `LCDGetDot` tests an LCD bit. In this routine, the global far pointer `pOldBuff` contains the buffer's RAM address. The shift amount in `CL` now moves the selected bit into the LSB of `AX`, where we mask it to produce a Boolean return value for the calling routine.

`LCDGetDot` and `LCDSetDot` use two different global pointers, because the Game of Life's playing field must remain unchanged until *after* the current generation tests all the old cells. I aimed `pNewBuff` at the **LCD Refresh RAM** buffer to allow us to see the dots appear, but you could put them in system RAM. In that case, just copy all of them to the **LCD Refresh RAM** at the end of each generation to reveal the new field in one blink.

Listing 5

This routine tests a dot on the LG64AA44D LCD panel and returns its state as a C Boolean value. If the row or column lies outside the LCD's boundaries, the return value is always zero. While this simplifies the higher level code for the Game of Life, it may not be the right action for all programs!

```

PUBLIC C LCDGetDot
PROC LCDGetDot
ARG Row:WORD, Col:WORD
USES ES, DI

XOR AX, AX ; set up zero return value

MOV BX, [Col] ; verify column
TEST BH, 80h ; negative?
JNZ @@Done ; ... yes
CMP BX, NUMCOLS ; too big?
JAE @@Done ; ... yes

MOV BX, [Row] ; verify row
TEST BH, 80h ; negative?
JNZ @@Done ; ... yes
CMP BX, NUMROWS ; too big?
JAE @@Done ; ... yes

MOV AX, [Col] ; set up column number
CALL LCDMakeAddr

LES AX, [pOldBuff] ; get buffer base pointer
ADD DI, AX ; ... update byte pointer

MOV AL, [ES:DI] ; ... fetch the byte
SHR AL, CL ; skid our bit into LSB
AND AX, 0001h ; ... isolate it

@@Done:
RET
ENDP LCDGetDot

```

The main loop calls `LCDSetDot` only when a cell changes, which means the new field must contain all the unchanged cells at the start of each generation. The `C memcpy()` library function copies the entire 32 KB buffer from the **LCD Refresh RAM** into system RAM at the end of each generation. Compared to the time required to compute a generation, `memcpy()` runs in essentially zero time.

Variations on a Theme

Even through the LG64AA44D panel has a moderately complex dot layout, the code remains fairly simple. Rather than list the routines for the 640×200 DMF651 and the TLY-365-121 panels here, check the source code files for the full details. I decided not to do the Sharp LM64015T 640×400 panel, because I still don't have a reliable source for its backlight inverters. If you can get one, have at it!

The 480×128 LM215 merits some discussion, if only because it has such a peculiar layout. Each of the four bits in the low nybble drives a separate quadrant: in effect, the panel has four 240×64 subpanels. It uses a 960 ns **Dot Clock**, so the firmware

Listing 6

The LM215 panel has more complex addressing, because it is actually four 240×64 panels butted together on one piece of glass. Both the bit shift amount and the byte offset depend on the dot's quadrant. The SETAE instructions, peculiar to the '386 and higher CPUs, set CL to 1 if the preceding CMP was "above or equal" and to 0 otherwise.

```
LCDNAME      EQU      "Hitachi LM215 (960 ns, set jumpers!)"
NUMROWS      =        128
NUMCOLS      =        480
TOTALCLOCKS  =        30720
ROWCLOCKS    =        480                      ; every other byte in buffer!
COLMODULUS   =        NUMCOLS/2
ROWMODULUS   =        NUMROWS/2
FILLVALUE    =        000h

PROC         LCDMakeAddr

CMP          AX,COLMODULUS      ; column
SETAE       CL                  ; CL = 1 if in right half
ADD         CL,CL               ; ... make it 2
CMP         BX,ROWMODULUS      ; upper or lower half?
SETAE       CH                  ; CH = 1 if in lower half
ADD         CL,CH               ; get total shift amount

ADD         BX,BX               ; make word table index
MOV         DI,[RowStarts+BX]   ; ... to get row start

DIV         [BYTE ColModulus]   ; get column offset
MOV         AL,AH               ; remainder = col offset
XOR         AH,AH               ; always less than 256...
ADD         AX,AX               ; times 2 for addressing
ADD         DI,AX               ; set up complete address

RET
ENDP         LCDMakeAddr
```

must also duplicate the dots in successive even- and odd-addressed **LCD Refresh RAM** bytes to present stable data to the panel.

The LM215's row address table has 128 entries, corresponding to the 128 visible rows. The code I used to fill it resembles that shown in Listing 2. I put only the even addresses in the table and modified the `LCDGetDot` and `LCDSetDot` functions to generate the corresponding odd addresses.

Listing 6 shows the LM215 `LCDMakeAddr` function. The bit shift amount depends on the dot's quadrant, a perfect application for the '386 `SETAE` instruction that generates a Boolean value from the flags set by `CMP`. The byte offset added to the table entry is the same for the first and last 240 columns in each row. Because `SETAE` doesn't exist on CPUs prior to the '386, you *must* tweak the code if you have an 8088 or 80286 target system.

Listing 7

The LM215 requires a 960 ns Dot Clock, so identical data must appear in successive even- and odd-addressed bytes to provide stable signals to the panel. The remainder of this code resembles that used for other, more normal, LCD panels, because `LCDMakeAddr` hides the addressing differences.

```

PUBLIC C LCDSetDot
PROC LCDSetDot
ARG Row:WORD,Col:WORD,Value:WORD
USES ES,DI

MOV AX,[Col]
MOV BX,[Row]
CALL LCDMakeAddr

LES AX,[pNewBuff] ; get buffer base pointer
ADD DI,AX ; ... update byte pointer

MOV BX,0101h ; set up bit mask
SHL BX,CL
NOT BL ; BL = AND mask, BH = OR mask

TEST [Value],0001h ; what do we want?
JNZ @@2 ; nonzero means set the bit
AND [ES:DI],BL ; ... clear it
AND [ES:DI+1],BL ; ... do the odd byte, too
JMP SHORT @@3

@@2:
OR [ES:DI],BH ; ... set it
OR [ES:DI+1],BH ; ... do the odd byte, too

@@3:
RET
ENDP LCDSetDot

```

The LM215 `LCDSetDot` function in Listing 7 updates both bytes with the new dot value. Apart from that, it resembles the code for the other panels because `LCDMakeAddr` handles the panel's peculiar bit and byte addressing.

The LCD initialization routines for all these panels disable blinking by gating a constant zero to the **LCD Data Multiplexer**. While you may have good reasons for showing blinking dots, the Game of Life doesn't need any. I leave that as an exercise for the interested reader (yes, *you*).

Stashing Slides

I used the Game of Life because its outputs make a far more interesting test pattern than a bland bit-by-bit diagnostic program. Unfortunately, waiting half a minute for each update puts a strain on even the most placid attention span. Rather than get involved with code optimization, I decided to store successive generations in RAM and play them back quickly. While the slide show repeats after a while, that allows you to study problems as they crop up over and over again... it's a perfect test-pattern application.

So far in this book, our *x86* target CPUs have been running in real mode, which allows only 20 address bits and limits RAM addresses to 1 MB. The PC's memory addressing restrictions, which we discussed in Chapters 6 and 7, limit contiguous RAM to the 640 KB starting at physical address 00000. After loading a program, allocating a few work buffers, and leaving room for a decent FAR heap area, the system may have only a few hundred KB available below the video buffers.

In contrast, when a '386 or higher CPU runs in protected mode, it can use all 32 address bits to reach 4 GB of memory. Even the lowly '386SX chip has 24 address pins and can access 16 MB, still significantly more than a paltry 640 KB. In most PC systems, the first megabyte of the protected mode address space remains devoted to the familiar real mode RAM and I/O devices, with the remaining storage appearing as a contiguous expanse starting at the 1 MB line.

This RAM area, generally called *extended memory*, differs from *expanded memory* that can be page mapped into some part of the lower 1 MB. The (now largely irrelevant) LIM 4.0 spec defined the operation of a standard DOS Expanded Memory Manager that controlled actual memory mapping hardware. The **EMM386** DOS device driver controls extended memory and can simulate expanded memory with no additional hardware. Refer back to Chapter 6 for a discussion of how that affected our ability to locate memory on the ISA bus.

Although the CPU must run in protected mode to access extended memory, we don't need a complete protected mode program to use that RAM. The system

BIOS includes functions that copy blocks of data in extended memory to and from the 1 MB available to real-mode programs, allowing us to create the data as usual and store it through a BIOS call.

The process begins by finding out how much extended memory resides on the target system. Listing 8 shows the code to access BIOS function INT 15h, AH=88h, which returns the extended memory size in kilobytes in register AX. Because AX holds only 16 bits, this function can handle a mere 64 MB of RAM... enough for an old '386SX, but painfully cramped in the days of Pentium Pro CPUs with nearly 1 GB of RAM on their system boards.

The various references disagree as to whether the INT 15h, AH=88h function clears the Carry flag to indicate success. In my test systems, Carry remains set even though AX contains the correct value, so I added a test to weed out some obviously bogus return values. If it misbehaves on your system, write a few testcases and see how your BIOS behaves.

The firmware turns bit 0 of the printer port at SYNC_ADDR (typically, LPT1) ON before invoking the software interrupt and OFF when it returns. If that bit stays ON, you know your system has problems. This will *certainly* happen on 8088 systems and will probably happen on oddball '286 systems, because the BIOS functions aren't quite compatible.

Function INT 15h, AH=88h reads the extended memory size from the battery backed RAM in the CMOS Real-Time Clock chip, where, on some systems, the

Listing 8

BIOS function Int 15h, AH=88h returns the extended memory size in KB. This memory starts just beyond the 1 MB of memory addressable in real mode. Because AX has only 16 bits, the function can report only the first 64 MB of extended memory!

```
regs.h.ah = 0x88;
outp(SYNC_ADDR,0x01);                /* mark the query          */
int86(0x15,&regs,&regs);
outp(SYNC_ADDR,0x00);

XMemSize = regs.x.ax;
printf(" BIOS reports %u KB\n",XMemSize);

if ((XMemSize >> 8) >= 0x0080) {
    puts(" ... on a small PC you can't have that much, so it's ignored");
    XMemSize = 0;
}

XMemLimit = (long)(XMemSize / FRAMESIZE);    /* truncate limit to a    */
XMemLimit *= (long)FRAMESIZE * 1024L;       /* multiple of the slide */
XMemLimit += ONE_MEG;                       /* above the line        */
printf(" Room for %u frames of %u KB each between %08lx and %08lx\n",
       XMemSize/FRAMESIZE,FRAMESIZE,ONE_MEG,XMemLimit-1);
```


BIOS stored it during the power-on tests after each reset. Other BIOSes store this value as part of a manual setup and complain if it doesn't match the actual amount of RAM found during POST. In any event, the return value should be exactly 1 MB less than the total amount of RAM installed in your system.

BIOS function INT 15h, AH=87h copies a block of data between any two locations in memory. The process looks straightforward: enter the source and destination addresses in a table in RAM, specify the number of 16-bit words to move in CX, aim ES:SI at the table, and issue the software interrupt. However, the references disagree on exactly what the table should contain in addition to the addresses, if anything.

It turns out the table actually forms the GDT (Global Descriptor Table) used when the CPU enters protected mode. Listing 9 shows the structure of the GDT entries: even when the code runs on a 32-bit '386 or higher CPU, the BIOS expects a 16-bit, 80286-style GDT for compatibility with the Original PC AT. The two addresses form part of a pair of segment descriptors that the CPU uses to access the data blocks in extended memory.

Listing 10 shows how I copied the Life fields into extended memory. The GDT requires memory addresses in 24-bit linear format, not the `seg:off` pairs we all love to hate. I filled in the GDT entries holding the segment length and access flag bits, even though some of the references imply the BIOS will do this automatically.

The code copies the entire 32 KB block of storage in order to make the extended memory addresses easy to decode by eyeball inspection. If you apply data

Listing 9

BIOS Int 15, AH=87h copies memory between any two addresses in RAM, specified as 24-bit linear values in a Global Descriptor Table, rather than the familiar `seg:off` pairs used below 1 MB. The GDT, made up of an array of the structures shown here, defines the CPU's access to memory in protected mode.

```
#pragma option -a                /* structure must be packed          */
typedef struct {                  /*                                     */
    WORD SegLimit;                /* size of segment in bytes          */
    WORD SegBaseLow;              /* low bytes of linear address       */
    BYTE SegBaseHigh;            /* high byte of linear address      */
    BYTE SegFlags;                /* 9B=code, 93=data, 91=RO         */
    WORD Reserved;               /* reserved, must be zero           */
} GDT_286;
#pragma option -a                /* restore default setting          */

#define MK_LINEAR(fp) (((long)FP_SEG(fp) << 4) + (long)FP_OFF(fp))

GDT_286 MoveGDT[6];             /* GDT for BIOS Move Block fn      */
```

Listing 10

In addition to the (obviously) necessary source and destination addresses in the GDT, this code fills in the segment length and access flags even though they're not required on my target systems. Bit 1 on the printer port marks the CPU's journey through protected mode; if that LED remains lit, your CPU fell off the tracks. The code to copy data back from extended memory is similar: just interchange the addresses.

```
memset(MoveGDT,0,sizeof(MoveGDT));           /* clear the GDT          */

MoveGDT[2].SegLimit = FRAMESIZE*1024;
MoveGDT[2].SegFlags = 0x93;
MoveGDT[2].SegBaseLow = (WORD)MK_LINEAR(pOldBuff);
MoveGDT[2].SegBaseHigh = (BYTE)(MK_LINEAR(pOldBuff) >> 16);
MoveGDT[3].SegLimit = FRAMESIZE*1024;
MoveGDT[3].SegFlags = 0x93;
MoveGDT[3].SegBaseLow = (WORD)XMemAddr;       /* target                  */
MoveGDT[3].SegBaseHigh = (BYTE)(XMemAddr >> 16);

sregread(&sregs);                             /* get current seg regs   */
regs.h.ah = 0x87;                             /* move block             */
regs.x.cx = FRAMESIZE*(1024/2);               /* number of words        */
regs.x.si = FP_OFF(MoveGDT);                 /* offset of GDT          */
sregs.es = FP_SEG(MoveGDT);                   /* ... segment            */

outp(SYNC_ADDR,0x02);                         /* mark the move          */
int86x(0x15,&regs,&regs,&sregs);               /* shazam!                */
outp(SYNC_ADDR,0x00);

printf(" stored at %08lx",XMemAddr);

XMemAddr += (long)FRAMESIZE * 1024L;          /* step to next block     */
```

compression before storing the frames, remember to tweak CX and step the addresses by the appropriate amount.

The two `outpw()` functions bracket the CPU's journey through protected mode by blinking bit 1 on the printer port: if that bit stays ON, you know your system got lost in hyperspace. My 33 MHz '386SX system takes about 32 ms to transfer a 32 KB block, about 1 MB/s. Because the transfer stays entirely within the system board RAM, the usual ISA bus slowdown doesn't come into play. You can easily see that switching to and from protected mode adds considerable overhead! A little experimentation with different buffer locations should reveal the size of the penalty. Hint: put *both* buffers in system memory and time it again.

Because the BIOS doesn't have access to any interrupt handlers in protected mode, it must disable all hardware interrupts during the entire transfer. If your code must receive high speed serial data or process other closely spaced interrupts, this function simply will not work for you. Also, the references lists several possible error return codes that I simply ignore here. You must deal with all those issues in a real application, of course.

After filling extended memory with Life generations, the code once again invokes `INT 15h, AH=87h` to copy the patterns back into memory below 1 MB. One of the FDB's DIP switches selects either `memcpy()` or a nested pair of C loops for the subsequent copy into the **LCD Refresh RAM**, allowing you to see the effects. A hardcoded 500 ms delay between each slide also gives you time to think about what's happening and recognize any problems.

One megabyte of extended memory holds 32 generations of 32 KB each, about 16 seconds. An 8 MB system, with 7 MB above the 1 MB line, stores 224 generations and presents a two-minute slice of Life. The low-order byte of the DIP switches on the Firmware Development Board sets the initial random number seed and, thus, selects one of 255 different slide shows. You can modify the code to pick more bits from the DIP switches or use an entirely different random number generator.

Because 200-line panels use only the low-order nybble of each byte, you can obviously store twice as many generations with a no-brainer data compression scheme. The Life field has many dead cells after the first few generations and you can probably pick up another 50% by run length encoding what's left... have fun!

Don't be surprised to find yourself rooting for those clumps of dots as they expand, contract, and consume each other. Life is like that...

Release Notes

The sample program files include the source code and **BIN** files for the four panels I mentioned in the text, written in Borland C and processed through Paradigm's **Locate** utility. Copy the **BIN** files to a diskette with the boot loader from Chapter 11 and bring some life to the LCD panel on your target system.

The code unique to each panel resides in an assembler file bearing the panel's name. The **MakeFile** produces a similarly named **BIN** file on drive A, so have a diskette with the boot loader ready when you recompile the code. If you add another type of panel, use an existing file as a template, update the **MakeFile**, and you're all set.

You also get my version of Paradigm's **Console.C** file. I replaced their demo code with BIOS serial functions to support simple, polled console I/O. If you're using **Locate**, you must also set up the **FARHEAP** constant that allocates a decent sized area for the program's work buffers. I used 0x8000, but you may find a smaller value works fine.

Homework: you can build some useful graphic code atop the dot drawing routines. For vector graphics, add line and arc drawing routines. For bitmaps, you'll need block move and copy operations that properly handle the scattered bits.

16 All Text Is Graphics

Have you noticed how much the few remaining DOS text-mode programs resemble Windows apps? Otherwise sensible companies spend precious programmer hours cross-dressing their programs... as though 2000 characters allowed enough screen real estate for fancy borders and pushbuttons.

Ah, well, such is progress.

This chapter marks, perhaps, a step in the same backwards direction. We're now going to turn a perfectly serviceable, bitmapped, graphics LCD panel into a moderately intelligent Glass Teletype with ANSI cursor positioning and attribute controls. We can then display status information without using a serial port or the video screen. If your project requires status output that *can't* appear on a standard VGA-compatible screen (perhaps because your embedded project puts the user interface on the VGA), this may be just what you need.

First, we'll see where the character bitmaps come from, then cover the assembler code that shuffles them into the **LCD Refresh RAM**. Next come the TTY routines that decode standard ASCII control characters and ANSI commands to provide full screen positioning.

I'll wrap up with a chilling murder mystery, The Case of the Capital T, that shows why you can't take anything for granted. Hint: you already know both the problem and the solution...

Filching Fossil Fonts

The Original IBM PC had two display options: the text-only Monochrome Display Adapter and the bitmapped Color Graphics Adapter. You could install both boards in the same system: the MDA presented high quality text on one screen and the CGA displayed moderately good graphics on another.

The catch was that, in graphics mode, the CGA hardware didn't support text output. The CPU painstakingly drew every character, dot by dot, whenever a program wrote text on the screen. Even though those functions *really* belonged with the CGA hardware, the PC's designers included them in the BIOS ROMs on the system board. The architecture for installable BIOS extensions came somewhat later, after the IBM PC/XT hit the scene.

Now, nearly two decades later, the PC Compatibility Barnacles dictate that every PC BIOS must include those ancient CGA functions, even though the standard

video board has become a Super VGA accelerator with its own BIOS extension. You may have the latest SuperTurbo 100 MegaWinMark video accelerator, but your PC's system board BIOS stands ready for the day when you reinstall that good old CGA. It could happen...

The CGA presented 640×200 dots in one color on a background of another color, which means anything it could do maps directly to our 640×200 LCD panels, at least if we're willing to give up those colors. In particular, the BIOS includes an 8×8 dot character font that, while it may not have the nicest looking characters in the world, lies waiting for our firmware to put it to good use.

We cannot, unfortunately, use the BIOS character drawing routines. The CGA used a simple, linear memory map quite different from our LCD panels, which, as you've seen, distribute dots from a single byte into widely separated locations on the panel. Worse, each panel has a different memory map that precludes one-size-fits-all firmware.

The Original IBM PC put those font bitmaps at address F000:FA6E. The PC Compatibility Barnacles guarantee that address remains valid in every PC clone ever built. The bitmap for a particular character starts at:

`F000:FA6E + 8×(character value)`

and occupies eight bytes. You can see how the process works in Listing 1, which copies a single character from the bitmap into the **LCD Refresh RAM**.

Listing 1

This C function writes a character bitmap, a.k.a. the character's glyph, into the LCD Refresh RAM. Because the assembly language routines hide all the panel-specific bit twiddling, this routine remains the same for all the LCD panels. The font width in our code is always eight bits, the same as the BIOS bitmaps, just to keep things simple.

```
void LCDWriteGlyph(int CharCode,int DotRow,int DotCol) {
    int ScanLine;

    if ((CharCode < 0) || (CharCode >= NumFontChars)) {
        CharCode = '?'; /* an obvious filler char*/
    }

    CharCode *= FontHeight; /* point to start of char*/

    for (ScanLine=0; ScanLine < FontHeight; ++ScanLine) {
        LCDWriteByte(*(pFont+CharCode+ScanLine),
            DotRow+ScanLine,DotCol,BlinkMode);
    }
}
```

The BIOS bitmaps include only the first 128 characters (0x00 through 0x7F), so `LCDwriteGlyph` substitutes a question mark if the character value exceeds 127. It then iterates through the eight bitmap rows, copying one byte at a time. In this case, `FontHeight` and `FontWidth` must both equal eight dots, as the BIOS table knows no other character size.

If you want different, smaller, or very detailed characters, you can modify this routine to copy the appropriate bits from your own custom bitmap table to the **LCD Refresh RAM**. This also requires changes in the `LCDwriteByte` routine that handle partial bytes, should your glyphs not be a multiple of eight bits wide.

On the other hand, double size characters, at least *ugly* double size characters, are comparatively easy. Just modify the code to duplicate each dot from the font table into two adjacent bits and duplicate that byte in the next LCD row, wherever that row may fall in the buffer.

The BIOS ROMs may not be the only source of bitmap characters in a stock ISA bus PC: any graphics board compatible with the VGA includes a BIOS extension with the fonts appropriate for it, ready for use. IBM's EGA and VGA adapters standardized a method for locating these bitmaps, giving you a wider choice of characters if your embedded PC has such a board in it. Because our LCD code operates entirely independently of the built-in BIOS, you can use any of the bitmaps without regard to the current BIOS video mode or display resolution.

The corresponding Bad News is that there's no tidy way to decide if the target system contains an MDA, a CGA, a VGA, an SVGA, or no video board at all. The code shown in Listing 2 checks the `INT 1Fh` vector, which *should* point to the bitmap for character 0x80 if the font can support more than 128 characters, which is the case for all video adapters other than the CGA.

The vector should not point to an actual interrupt handler, although some BIOSes seem to aim it at an unexpected-interrupt handler during their power-on self tests and leave it set if they don't find a video board. As a result, your code cannot assume that a nonzero value indicates a valid pointer to a font table. I know which board I have installed here and I'll leave you to experiment on your own target system.

The algorithms I've seen in other books assume that the system will have at least one video adapter, so they don't test for the possibility of no adapter at all. If you adapt a standard algorithm, make sure it doesn't fall into the same trap.

If the code finds a nonzero `INT 1Fh` vector, it issues `INT 10h`, `AX=1130h` to find the address of one of the board's font bitmaps. VGA boards include both 8×8 and 8×16 fonts, the latter of which works eminently well with 400-line LCD panels.

Listing 2

This routine sets up the font bitmap for the rest of the code. If the system has no video board, the INT 1Fh vector should be 0000:0000 to indicate that the default 128 character CGA font is in effect. That vector will be nonzero if an EGA or VGA board is installed, in which case Int 10h, AX=1130h returns a pointer to the desired font table. I used the 8x8 font for 200 line panels and the 8x16 font for larger panels to show that more dots can produce a slightly less ugly font. If you prefer more information over prettier characters, use the 8x8 font regardless of the panel size.

```
void LCDInitFont(void) {
struct REGPACK rp;
void far *Int1FVector;
int VidMode;

/*--- figure out what character table is available */
puts("Initializing character generator...");

Int1FVector = (void far *)*(long far *)MK_FP(0x0000,4*0x1F);
printf(" Int 1F vector is %Fp\n",Int1FVector);
if ((void far *)NULL == Int1FVector) { /* have chars 80..FF? */
puts(" Using ROM BIOS CGA font");
pFont = (BYTE far *)MK_FP(0xF000,0xFA6E); /* nope, use BIOS ROM */
FontHeight = 8; /* which is 8x8 only */
FontWidth = 8;
NumFontChars = 128; /* basic ASCII only! */
}
else {
puts(" Using video card font");
VidMode = *(BYTE far *)MK_FP(0x0040,0x0049);
printf(" ... resetting video mode: %u\n",VidMode);

rp.r_ax = (0x00 << 8) | VidMode; /* Reset video mode */
intr(0x10,&rp);

rp.r_ax = 0x1130; /* Get Font Pointer info */
if (NumDotRows > 200) {
rp.r_bx = 0x0600; /* BH = 06 for 8x16 font */
FontHeight = 16;
}
else {
rp.r_bx = 0x0300; /* BH = 03 for 8x8 font */
FontHeight = 8;
}
printf(" ... fetching %u line font pointer\n",FontHeight);
intr(0x10,&rp);

#if 0
printf("ax %04X bx %04X cx %04X dx %04X ds %04X es %04X bp %04X\n",
rp.r_ax,rp.r_bx,rp.r_cx,rp.r_dx,rp.r_ds,rp.r_es,rp.r_bp);
#endif

pFont = (BYTE far *)MK_FP(rp.r_es,rp.r_bp);
FontWidth = 8;
NumFontChars = 256;
}

printf(" pFont=%Fp, FontHeight=%u, NumFontChars=%u\n",
pFont,FontHeight,NumFontChars);
}
```

Listing continues on next page

Listing continues from previous page

```
/*--- set up screen control variables */
    NumCharRows = NumDotRows / FontHeight;
    NumCharCols = NumDotCols / FontWidth;
    VScrollAllowed = 1;

    printf(" Panel has %u character rows and %u character cols\n",
        NumCharRows, NumCharCols);

    return;
}
```

You'll quickly see that with more dots available the characters don't look *quite* so ugly. The code then sets several variables detailing the font's height, width (always eight bits for now), and the number of characters in the bitmap.

This should work with VGA video boards, but I won't be surprised to find that it fails in systems with other display boards. In particular, EGA boards don't include the 8×16 font table. I much misdoubt any of you still have an EGA; if you do, you must tweak the code to support 400-line panels. In any event, should strange things happen, enable the `#if 0` code stub in Listing 2 to dump the registers returned by `INT 10h` and see what went wrong.

Incidentally, I used the `intr()` function from the Borland C library, rather than the more familiar `int86x()`, because the BIOS function returns the font pointer in `ES:BP`. Check your compiler doc, but as near as I can tell, `intr()` provides the only way to examine `BP` without resorting to inline assembler.

You can short circuit the entire font search routine with your own code to install a completely custom font. Keep in mind that the rest of the code assumes that the bitmaps are eight bits wide, with the sky the limit in the other direction.

Also remember that the **LCD Refresh RAM** contains bitmap patterns, not the ASCII character values. Although the CGA included a screen readback function that searched the font bitmap table to find the pattern matching the one at the cursor location, I didn't attempt to duplicate that stunt. If you *must* read characters back from the **LCD Refresh RAM**, I suggest allocating a 2000 (or 4000, or whatever) byte buffer and storing the ASCII characters in parallel with the bitmaps. Trust me, it's a *whole* lot easier and faster than pattern recognition.

Now that we have a source of character bitmaps, the next step drops them into the **LCD Refresh RAM**. As you will recall from Chapter 15, most LCD panels are not nearly so accommodating as the good old CGA.

Divvying up the Dots

Listing 3 shows `LCDwriteByte`, a function that writes a single byte into the DMF651 **LCD Refresh RAM**. The character coordinates use dot increments, but the column must be a multiple of eight to position the characters correctly. Similarly, positioning the character rows on multiples of the font height produces a regular vertical spacing.

Listing 3

The fundamental character operation involves writing a single byte into the LCD Refresh RAM at a specific row and column. This DMF651 routine computes the buffer address, splits the byte into two nybbles, and sets the blinking bits for each. Aligning characters on 8-dot column boundaries considerably simplified the code!

```

PUBLIC C LCDwriteByte
PROC   LCDwriteByte
ARG    CharCode:WORD, DotRow:WORD, DotCol:WORD, Blinking:WORD
USES   ES, DI

    MOV     AX, [DotCol]           ; set up column number
    AND     AX, NOT 0007h          ; ... force byte alignment
    MOV     BX, [DotRow]           ; set up row number
    CALL    LCDMakeCharAddr

    LES     AX, [pLCDBuff]         ; get buffer base pointer
    ADD     DI, AX                 ; ... update byte pointer

;--- insert the left nybble and blinking pattern

    MOV     AX, [CharCode]         ; fetch new byte value
    MOV     DL, AL                 ; save high bits for later
    SHR     AL, 4                 ; align nybble, clear blink bits

    CMP     [Blinking], 1
    JE      SHORT @@1
    MOV     AH, DL                 ; no blinking, copy data bits
    AND     AH, 0F0h
    OR      AL, AH
@@1:
    MOV     [ES:DI], AL            ; wr blinking + data to buffer

;--- insert the right nybble and blinking pattern

    INC     DI                     ; step to next buffer byte
    AND     DL, 00Fh              ; get nybble, clear blink bits

    CMP     [Blinking], 1
    JE      SHORT @@2
    MOV     BH, DL                 ; no blinking, copy data bits
    SHL     BH, 4
    OR      DL, BH
@@2:
    MOV     [ES:DI], DL            ; write blinking + data to buffer

    RET

ENDP   LCDwriteByte
```

The Graphic LCD Interface supports blinking characters on the DMF651 panel. The code splits each byte from the BIOS bitmap in half and stores the fragments in the low nybbles of two successive buffer bytes. Because the hardware accepts four bits at a time, we can update the bytes without using the direct dot plotting code from Chapter 15.

The **LCD Data Multiplexer** displays bits 4:7 alternately with bits 0:3 from each byte, switching between them at the selected blink rate. To suppress visible blinking, the high and low nybbles in each byte must be identical. If one nybble holds zero, the dots in the other nybble appear to blink on and off against a blank background. You can implement other blinking schemes if you like, as well as inverse characters and so forth.

If you want unaligned or proportionally spaced characters on your LCD panel, your code must handle bit patterns that don't fit neatly into each byte. In fact, a single bitmap byte may affect three successive **LCD Refresh RAM** bytes. The code gets particularly tricky near the right edge of the panel, where a character may run off the end of the panel. Should you wrap, clip, or ignore that character?

Aligned characters also simplify the buffer address calculation routine. Listing 4 shows the few lines of `LCDMakeCharAddr` for a DMF651 panel. Recall that the dot-drawing `LCDMakeAddr` routine in Chapter 15 set `CL` to the value that shifted a dot to or from the low-order bit. Now, because we always fill the entire nybble with data, we need no further shifting.

Adapting this code to other panels should be reasonably straightforward. For example, the 640×400 LG64AA44D panel accepts eight bits on each **Dot Clock**

Listing 4

This routine for a DMF651 panel computes the LCD Refresh RAM address corresponding to a character's location given as a dot column in `AX` and a dot row in `BX`. The characters always align on an 8x8 dot grid, eliminating the shift amount computation.

```
PROC      LCDMakeCharAddr

ADD       BX,BX                ; make word table index
MOV       DI,[RowStarts+BX]

DIV       [BYTE ColModulus]   ; col / modulus
MOV       CL,0                 ; force shift amount to zero
MOV       AH,0
ADD       DI,AX                ; DI points to the byte

RET

ENDP      LCDMakeCharAddr
```

and, thus, doesn't support blinking. Bits 0:3 of each byte appear on rows 0 through 199, while bits 4:7 fill rows 200 through 399. You must insert the new data into each nybble without disturbing the existing bits in the *other* nybble of the byte.

Listing 5 shows the LG64AA44D LCDWriteByte function. For this panel, LCDMakeCharAddr sets CL to align the new nybble in either the low or high half of the **LCD Refresh RAM** byte, depending on which row it occupies. The code reads the existing byte from the buffer, strips out the old bits, inserts the new ones,

Listing 5

The LG64AA44D LCD Refresh RAM layout is more complex than the DMF651. Because the 400-row panel accepts and displays eight bits on each Dot Clock, the hardware cannot support blinking. This routine splits a byte into two nybbles and inserts them into the appropriate parts of the two target bytes. As with the DMF651, the character's column address must be a multiple of 8.

```

PUBLIC C LCDWriteByte
PROC    LCDWriteByte
ARG     CharCode:WORD, DotRow:WORD, DotCol:WORD, Blinking:WORD
USES    ES, DI

    MOV     AX, [DotCol]           ; set up column number
    AND     AX, NOT 0007h          ; ... force byte alignment
    MOV     BX, [DotRow]          ; set up row number
    CALL    LCDMakeCharAddr

    LES     AX, [pLCDBuff]         ; get buffer base pointer
    ADD     DI, AX                ; ... update byte pointer

;--- insert the left nybble

    MOV     AX, [CharCode]         ; fetch new byte value
    MOV     DL, AL                ; save high bits for later
    SHR     AL, 4                 ; set up for alignment
    SHL     BH, CL                ; align to target nybble
    MOV     BH, 0F0h              ; set up bit mask
    ROL     BH, CL                ; ... align to strip old bits

    MOV     AH, [ES:DI]           ; fetch target bits
    AND     AH, BH                ; strip old bits
    OR      AL, AH                ; tuck in new bits
    MOV     [ES:DI], AL           ; put 'em back in the buffer

;--- insert the right nybble
;   this uses the same nybble mask because it's in the same screen half

    INC     DI                    ; step to next buffer byte
    AND     DL, 00Fh              ; extract low nybble
    SHL     DL, CL                ; align to target nybble

    AND     BH, [ES:DI]           ; fetch target bits, strip old
    OR      DL, BH                ; tuck in new bits
    MOV     [ES:DI], DL           ; put 'em back in the buffer

    RET
ENDP    LCDWriteByte
```

and writes the modified byte back into the same buffer address. Although it's slightly slower than the DMF651 code, you won't notice the difference in real life.

So much for a single character. Now, to put them where we want...

Gazing in the Glass

First of all, a confession. I have not implemented a real blinking character cursor for the LCD panel, because I use it as an output-only status device. Nevertheless, we must have a name for the spot where the next character will appear and *current cursor location* seems as good as any. If you want a visible cursor at the current cursor location, well, it's a simple matter of firmware: load a blinking block and be done with it! Of course, if your hardware doesn't support blinking, you may have a bit more difficulty.

Extra credit project: implement blinking on an 8-bit panel without using the dot multiplexer. Hint 1: you can hitch an interrupt routine to the **Frame Sync** pulse, count off refresh intervals, and rewrite the character at the cursor position on the fly. Reread the discussion of output-only ports in Chapter 9 before you proceed, to avoid winding up with two routines clobbering each other's output bit patterns. Hint 2: consider the dots on the top row before you write the interrupt handler.

The ASCII character set includes about 32 control characters that are not ordinarily displayed. Instead, they affect how the remaining 96 characters appear on paper, the current rage when that character set coalesced into a standard. The control characters include such old friends as **Carriage Return** and **Line Feed**, as well as obscure relatives like **End of Medium**.

Because the font tables in the BIOS and VGA ROMs include visible bit patterns for all those characters, your code may ignore their control functions and simply display them as ordinary, albeit funny looking, text.

Which control character functions you implement and exactly what they do depends on how thorough you are. Nobody (well, hardly anybody) uses paper output any more, making the mappings into video functions somewhat arbitrary. The term *Glass Teletype* pretty well describes the level of control you can achieve: no matter how hard you try, the results still look a lot like a really short roll of paper.

Listing 6 shows the bare bones implementation I chose for the LCD panel code. **Line Feed** and **Carriage Return** do pretty much what you'd expect, **Form Feed** clears the panel, **Back Space** moves the cursor one space to the left, **Tab** moves it rightward to the next multiple of four columns, and **Delete** erases the character at the cursor

Listing 6

Emulating a Glass Teletype requires picking control characters out of the incoming data that can affect either the current cursor location or the LCD Refresh RAM contents. This code handles a basic set of controls that suffice for simple text output applications. The VScrollAllowed variable defaults to TRUE, so that a Linefeed on the bottom row of the panel scrolls the entire panel contents up by one character row.

```
void LCDSendChar(int CharCode) {
    switch (CharCode) {
    case LF :
        if (CurDotRow >= (NumDotRows - FontHeight)) { /* last line? */
            if (VScrollAllowed) {
                LCDScrollUp(FontHeight); /* yes, scroll up a line */
                CurDotRow = NumDotRows - FontHeight; /* force to last line */
            }
            else {
                CurDotRow = 0; /* no scroll, wrap to top */
            }
        }
        else {
            CurDotRow += FontHeight; /* no, so step down */
        }
        break;

    case CR :
        CurDotCol = 0; /* to leftmost col */
        break;

    case FORMFEED :
        LCDClear();
        break;

    case BS :
        if (CurDotCol >= FontWidth) { /* if not at edge */
            CurDotCol -= FontWidth; /* back up one column */
        }
        break;

    case TAB : /* tab to next stop */
        do {
            LCDSendChar(' ');
        } while (((CurDotCol/FontWidth)+1) % TABINTERVAL);
        break;

    case DEL :
        LCDScrollLeft(CurDotRow, CurDotCol, FontHeight, FontWidth);
        break;

    default :
        LCDWriteGlyph(CharCode, CurDotRow, CurDotCol); /* display it */
        CurDotCol += FontWidth; /* tick column counter */
        if (CurDotCol > (NumDotCols - FontWidth)) { /* past end of line? */
            CurDotCol = 0; /* reset to left edge */
            if (!HWrapAllowed) {
                LCDSendChar(LF); /* and step to next line */
            }
        }
    }
}
```

position. The code treats everything else as a displayable character and passes it to the LCD panel character routine.

The `CurDotRow` and `CurDotCol` variables hold the current cursor location in terms of the dot coordinate of the character's upper left corner. Although I force the characters to lie in a grid the size of the font bitmap, you can tweak the code to handle multiple character sizes and other effects.

The code supports vertical scrolling, displaying the last 25 (or 50, or whatever) lines sent to the panel. I won't show the code here, but, contrary to what you might think, you can't just schlep the bits around with a single `REP MOVSB` instruction. As an example, consider moving characters on an LG64AA44D panel, where the same byte can hold both source and target dots. Not a pretty sight, I assure you.

The **Delete** character requires a routine that removes a rectangular block of dots by shifting the remaining characters in the line leftward by one position. In that case, you must copy a group of dots to a single position, then repeat that for each row in the character while filling the right end of the line with blanks. I did not include a corresponding **Insert** function, but you can add it fairly readily should your application require it.

In today's embedded systems, most of the output characters will come from C strings, so I included an `LCDSendString` function that handles the familiar zero-terminated strings. You can embed control characters in the strings or send them directly to `LCDSendChar` as needed. Perforce, ASCII character 0 (that's a binary zero, not 0x30) becomes a nondisplayable control character.

The code also includes several utility functions that enable and disable vertical scrolling, end-of-line wrapping, and so forth. Examine the source code on the diskette, if you're interested in this sort of thing (as you *should* be, having gotten this far in the book!)

One thing you *can't* do with ASCII control codes, though, is change the position of the next output character without affecting the existing text. For example, once you reach the last row of the screen, you can't get back to the top without sending a **Form Feed** character that clears everything. Seems a bit extreme, doesn't it?

Presenting a nicely formatted status display using only ASCII control codes is nearly impossible, although you've surely seen some noteworthy attempts at pulling it off. Fortunately, the ANSI standards committee set down some rules for cursor control on serial display terminals that we can adapt to our graphics LCD panels. The end result is well worth the programming effort.

Assume the Position...

You're probably familiar with ANSI control sequences, if only from their use in PC-DOS **PROMPT** strings. For our present purposes, I'll simply say that they allow the same output string to produce similar results on anything that can interpret the sequences. Because they're a genuine standard (pretty much), you'll find lots of *anythings* that can interpret them.

Figure 1 summarizes the control sequences used in the Graphic LCD character interface. Each sequence starts with the same two characters: 0x1B (**Escape**) and 0x5B (the left square bracket: [). Next come any numeric parameters, represented with ASCII digits, and the terminating letter that identifies the command. The case of that final letter is significant: **ESC[2j** puts the cursor at row 2, column 1 while **ESC[2J** clears the panel and homes the cursor. Pay attention to your typing, OK?

Figure 1

All ANSI Cursor Control Sequences begin with two common characters:

ESC ASCII 27, the Escape character

[ASCII 91, the left square bracket

There are no blank characters within these command strings. Numeric parameters, represented by an octothorpe (#), use ASCII decimal notation (characters 0x30 through 0x39) and default to 1 if omitted. Row and column numbers start with 1 at the upper left. The upper/lower case of the trailing letter is significant!

Command	Example	Function
ESC[#A	ESC[2A	Cursor up # rows (up 2)
ESC[#B	ESC[B	Cursor down # rows (down 1)
ESC[#C	ESC[10C	Cursor right # columns (right 10)
ESC[#D	ESC[5D	Cursor left # columns (left 5)
ESC[#;#H	ESC[H	Set cursor to row;column (default 1,1)
ESC[#;#f	ESC[1;2f	Set cursor to row;column (1,2)
ESC[#;#j	ESC[3j	Set cursor to row;column (3,1) (H,f, and j are all synonyms)
ESC[s	ESC[s	Save current cursor location (1 level)
ESC[u	ESC[u	Restore saved cursor location
ESC[2J	ESC[2J	Clear display and home cursor
ESC[K	ESC[K	Clear from cursor to end of row
ESC[#h	ESC[7h	Set display mode, ignored except for: wrap at end of each row
ESC[#l	ESC[7l	Reset display mode, ignored except for: no wrap at end of row
ESC[#m	ESC[0m ESC[7m	Set display attributes, ignored except for: disable all attributes set blinking attribute

Decoding these sequences can be somewhat difficult, because the numeric parameters for some commands may be omitted or replicated. The state machine shown in Listing 7 tracks the possible combinations and calls a decoder routine when it encounters the final command character. Every character sent to the display *must* pass through this routine, imposing a moderate performance penalty for the privilege of standardized cursor control.

The actual ANSI function routines are nearly anticlimactic, as you can see in Listing 8. Most require just a line or two of code that adjusts the cursor position or sets a control variable to affects subsequent characters. Adding new commands is also easy: just insert a new `switch case` clause and the requisite code.

One conflict between ANSI cursor controls and Glass Teletype mode occurs when you write a character into the extreme lower right corner: row 25, column 80 on a 200-line display. In TTY mode, the panel scrolls upward by the font height to make room for the next line of characters, discarding the top row of characters in the process. Because ANSI controls generally create a static display image, perhaps

Listing 7

Decoding the ANSI control sequences depends on a state machine to track all the allowable parameters and command terminators. Each new character sent to the LCD panel invokes this function. It extracts the numeric parameters and passes control to the function decoder when it encounters the terminating letter in the command.

```
void LCDAnsiDriver(int NewChar) {  
    ANSIBuffer[ANSICharCtr] = NewChar & 0x7F;    /* strip & save      */  
    ++ANSICharCtr;  
    ANSICharCtr = min(ANSICharCtr,ANSI_MAXLEN-1);  
  
    switch (ANSIState) {  
    case ANSI_WAIT :    /* waiting for ESC char      */  
        ANSICharCtr = ANSIPParamCtr = 0;    /* force buffer restart */  
        if (ESC == NewChar) {  
            memset(ANSIBuffer,0,sizeof(ANSIBuffer));  
            ANSIState = ANSI_MODE;  
        }  
        else {  
            LCDSendChar(NewChar);    /* just dump it      */  
        }  
        break;  
    case ANSI_MODE :    /* expect [ as next char */  
        if ('[' == NewChar) {  
            ANSIState = ANSI_PARAM;  
        }  
        else {  
            LCDAnsiNull(NewChar);    /* ditch sequence    */  
        }  
        break;  
    }
```

Listing continues on next page

Listing continued from previous page

```
case ANSI_PARAM :
    if ('=' == NewChar) {
        break;
    }
    else {
        if (isdigit(NewChar)) {
            ANSIState = ANSI_NUM;
        }
        else {
            if (isalpha(NewChar)) {
                LCDAnsiGetNumber();
                LCDAnsiDoCmd(NewChar);
            }
            else {
                LCDAnsiNull(NewChar);
            }
        }
    }
    break;
case ANSI_NUM :
    if (',' == NewChar) {
        LCDAnsiGetNumber();
    }
    else {
        if (isdigit(NewChar)) {
            ANSIState = ANSI_NUM;
        }
        else {
            if (isalpha(NewChar)) {
                LCDAnsiGetNumber();
                LCDAnsiDoCmd(NewChar);
            }
            else {
                LCDAnsiNull(NewChar);
            }
        }
    }
    break;
default :
    LCDAnsiInit();
}
/* error, restart decoder*/
```

with a tidy Windows-ish border around the whole screen, you definitely *don't* want vertical scrolling when you write a character in that position!

You can enable line wrapping by sending **ESC[7h** and disable it with **ESC[7l** around the offending characters, but that's a nuisance. Because I plan to use the panel as a static display, I simply disable vertical scrolling in the ANSI initialization routine. I can't find an ANSI control sequence to handle this, but I won't be surprised if one of you folks comes up with the Official Doc defining that command.

So, there you have it: a reasonably fast, large, and cheap character output device that leaves all the standard PC hardware untouched. Now you can run your

embedded PC program, use the target system's serial port or video display as usual, and still present rapid, intelligible diagnostic information for your own use without affecting anything else.

Listing 8

This excerpt from the ANSI function decoder shows that handling the commands is mostly a matter of a few calls to routines in the LCDChars module that twiddle the cursor location. We absorb the unused ANSI functions to prevent confusing the LCD panel's code with functions it cannot perform.

```
void LCDAnsiDoCmd(int NewChar) {
    BYTE Counter;

    ANSIState = ANSI_WAIT;          /* by default... */
    switch (NewChar) {
        case 'A' :                  /* cursor up */
            LCDSetCharCursor(LCDGetCharRow()-ANSIParams[0],LCDGetCharCol());
            break;
        <<< cases omitted >>>
        case 'H' :                  /* set cursor position*/
        case 'f' :                  /* (synonym) */
        case 'F' :                  /* (synonym) */
            for (;ANSIParamCtr < 2; ++ANSIParamCtr) {
                ANSIParams[ANSIParamCtr] = 1;          /* force defaults */
            }
            LCDSetCharCursor(ANSIParams[0]-1,ANSIParams[1]-1);/* fix origin! */
            break;

        case 'J' :                  /* erase, home cursor */
            if (2 == ANSIParams[0]) {
                LCDClear();
            }
            break;

        case 'K' :                  /* erase to end of line */
            Counter = NumCharCols - LCDGetCharCol();
            while (Counter) {
                LCDSendChar(DEL);
                --Counter;
            }
            break;
        <<< cases omitted >>>
        case 'S' :                  /* save cursor location */
            ANSISavedRow = LCDGetCharRow()+1;
            ANSISavedCol = LCDGetCharCol()+1;
            break;

        case 'u' :                  /* restore cursor location */
            LCDSetCharCursor(ANSISavedRow-1,ANSISavedCol-1);
            break;

        default :
            LCDAnsiNull(NewChar);    /* dump all others... */
    }
}
```

The Rest of the Stories

The 640×200 TLY-365-121 panel resembles the DMF651, although you must remember that each row contains 1280 dots. Tweak the DMF651 character address code supplied on the diskette and it'll work fine for you.

The 480×128 LM215, as always, presents a challenge. Because the four dots in each byte appear in different quadrants you must set one bit at a time. Drawing the dots for a single 8×8 character thus requires 64 calls to the `LCDSetDot` function we used in Chapter 15. Modify that code to support blinking if you really must have it. All this, just for a lousy 960 characters? I didn't think so, either, which is why you won't find any code for it on the diskette.

The 640×400 LM64015T should be a piece of cake because each bitmap byte appears in a single LCD Refresh RAM byte. I didn't write code for it, because, lacking both the backlight inverter and the desire to tinker with kilovolt power supplies, I can't see the results.

Now, to tie everything you've read in this book together, here's a terrifying tale...

The Case of the Capital "T"

After I'd finished the Graphic LCD Interface hardware described in this book, I used it to present debugging and tracing output from the x86 protected mode kernel I described in my *Firmware Furnace* column in *Circuit Cellar INK* magazine Issues 48 through 65. The LCD provided a convenient way to present rapidly changing status information without using the PC's video display.

However, when I wrote the protected-mode driver for the TLY365 panel and ran a set of test patterns, my target system developed a curious problem. Although the LCD panel worked fine, the system hung midway through the *next* BIOS boot sequence after I pressed the Reset button. Cycling the PC's power cured the problem, but it still hung reliably after every manual reset.

Hmmm...

Some scope probing revealed that the CPU was running in a tight loop doing a little I/O and a lot of memory writing. It surely wasn't any of my code because I didn't have any BIOS extensions installed. Just to make sure, I pulled the battery backed RAM out of the Firmware Development Board's socket. The CPU still got wedged after each reset.

Row 97, the *second* row of the character cell, begins at address 7800. The dot pattern for that row begins with the tips of the T's serifs and its two-dot central stroke. The hex value for that pattern is 5A, which should look suspicious already. Now, recall that the TLY365 accepts four bits of data on each **Dot Clock** and the Graphic LCD Interface implements blinking by alternating between the nybbles of each **LCD Refresh RAM** byte.

Because the **T** *isn't* a blinking character, the refresh buffer bytes at addresses 7800 and 7801 must have identical nybbles. Therefore, those two bytes are 55 AA.

Now, if *that* doesn't raise your hackles, you flunk...

The BIOS boot routine scans memory between C0000 and EFFFF for BIOS extensions after each reset. By definition, a BIOS extension starts on a 2 KB memory boundary with two flag bytes. Do you remember, from Chapter 6, that the BIOS extension flag bytes are 55 and AA?

The second character on line 12 was either a lowercase **a** or a blank, neither of which has any dots on row 97. That means a pair of binary zero bytes following the flag bytes. The *third* byte of a BIOS extension header gives the extension's length in multiples of 512 bytes.

So, the first three bytes of that line just happened to define a BIOS extension at address 7800 with a length of either zero or $512 \times 256 = 128$ KB. The fact that I didn't deliberately write an extension doesn't matter; the BIOS found those bytes during its memory scan and didn't inquire as to my intent.

Gotcha!

A valid BIOS extension includes a checksum byte that makes the sum of all the bytes in the extension equal to zero. Evidently, the BIOS checksum routine in my PC concluded that an extension of zero length, lacking any content, is always valid. My guess is that the BIOS recognized the header, but the checksum routine simply gave up when it saw a zero length byte.

So the situation goes a little something like this...

Immediately after turning the power on, the BIOS finds nothing particular in the **LCD Refresh RAM** and boots normally. My protected-mode code wrote its test pattern on the LCD and, quite inadvertently, plunked what looked like a BIOS extension header precisely at a 2 KB boundary within the refresh buffer.

When I pressed the system's Reset button, the BIOS ran through its extension scan again, whereupon it found the bogus header at address D000:7800. It erroneously concluded that the extension's checksum was valid and branched to offset 7803 in the **LCD Refresh RAM**. What happened after that was up for grabs, although we know the first byte of the "program" was a binary zero.

The solution was easy enough: disable the **LCD Refresh RAM** when the ISA bus **ResDrv** line goes active and keep it disabled until our firmware gains control. With the RAM disabled, the BIOS scan cannot find an extension in the buffer. Any program that uses the Firmware Development Board must enable the RAM before writing to the LCD.

You'll recall that we added a MAX691 watchdog timer with a latch to stretch its timeout interval after a reset. That guardian hardware has a signal that will keep the BIOS under control. Page back to Schematic 2 in Chapter 7 for the details.

The signal **+No Access Yet** from U18 goes high whenever **ResDrv** is active and returns low when the firmware writes to port 031C. That port contains the watchdog, character LCD, serial number, and other miscellaneous bits.

You might solve this another way, too. Because the BIOS scans in ascending address order, you could install a deliberate BIOS extension in the battery backed RAM. That extension, which would get control before the BIOS hits the **LCD Refresh RAM**, could set up the Graphic LCD Interface for the particular panel and clear the buffer to ensure that the BIOS doesn't find anything disturbing during the remainder of its scan.

This error was an oversight, pure and simple. I knew (and so did *you*) that the **LCD Refresh RAM** contents survived a reset. It *never* occurred to me that the BIOS might discover an extension in the bits left over from the last message!

If you build anything with dynamic bit patterns in the region where the BIOS expects extensions, take heed. You, too, may spend hard time wondering why your system doesn't boot correctly once in a while.

Release Notes

The files for this chapter check out the Glass Teletype and ANSI code on both the DMF651 and LG64AA44D panels, which are the "best of breed" LCDs in my collection. I wrote the code in Borland C and processed it through Paradigm **Locate**. Use the appropriate boot sector loader from Chapter 11 to start the files from diskette.

17 For Further Study

If you've worked your way through the book to this point, you will have a good understanding of how the Industry Standard Architecture bus operates, how to build useful ISA bus hardware, and write functional low-level firmware.

In this chapter, I'll discuss some aspects of the ISA bus that I either glossed over or simply omitted from the Firmware Development Board's hardware and firmware. These are relatively complex topics that don't have much relevance to current ISA bus designs, primarily because contemporary PC systems provide better ways to achieve the goals.

Crossing the 1 MB Line

Each ISA bus board can access 16 MB of memory using the 24 memory address bits that appear on each ISA connector. However, both the battery backed RAM and the Graphic **LCD Refresh RAM** on the Firmware Development Board decode only 20 bits and reside entirely below the 1 MB line. Why is that?

In short, back when the ISA bus was new and 80286 CPUs were state-of-the-art, the ISA bus matched the CPU's performance rather well and 16 MB of RAM was a *lot*. Neither condition holds true today.

Today, typical system boards accommodate 128 MB or more of RAM and even desktop operating systems require 32 MB for *normal* operation. Given the 24-bit address space available to an ISA bus board, its RAM must reside well below the middle of the system board RAM. But very few operating systems allow a block of special purpose RAM within the memory space above 1 MB. If you expect to use your ISA bus memory for, say, a big LCD buffer, you'll watch your operating system's data in action, not your own bits!

Even if you put general purpose RAM on that board, it will run at ISA bus speeds instead of the 60 ns access time found in system board RAM. Compare 60 ns with the times shown in Photo 1 of Chapter 13 and you'll see why using ISA bus RAM in a contemporary system is an Exceedingly Bad Idea.

You can assign and use system memory as you wish in an embedded system project, because you're writing the code yourself. Some BIOS setup routines can carve a hole in the system board RAM for an ISA bus board's memory, either by disabling the system's RAM at those addresses or remapping it to a different address. Just don't expect much help if you're trying to run a standard operating system as part of your embedded project. They simply don't do the things we need.

To gain access to any memory above 1 MB, your program must either run in protected mode or call routines that do. Nowadays, toolkits from Phar Lap Software and others make writing protected mode embedded programs relatively straightforward, but there's a whole lot of unexplored territory between the down-to-the-metal firmware we've been using and the code you write with a protected mode toolkit. If you go that route, spend some time making the detailed measurements you've seen in this book *before* assuming that protected mode operation provides the performance you need.

Of course, you could use the BIOS extended memory copying routines shown in Chapter 15, provided you work around their limitations. Most embedded programs cannot stand long periods without interruptions, so make sure you understand how your application will cope as the BIOS switches into and out of protected mode.

Check the references for information on the **-MemR** and **-MemW** bus signals that control access above 1 MB. The **LA17** through **LA23** address bits also have slightly different timing requirements that will affect your logic design.

Direct Memory Access

Everything we've done with the ISA bus requires direct supervision from the CPU, whether to read an I/O port, write data into memory, or respond to an interrupt. In effect, the CPU must execute at least one instruction per ISA bus operation. As it turns out, the system board and ISA bus include hardware that can transfer data between I/O ports and memory without involving the CPU, a process called DMA: Direct Memory Access.

The diskette controller uses DMA to transfer data between the diskette drive and memory, but, because the BIOS can handle only one task at a time, it puts the CPU into a tight loop until the DMA transfer finishes. That's not a particularly good use of the DMA hardware, but that's how the real mode BIOS works.

Many sound boards transfer digital audio data using the DMA hardware, leaving the CPU free to render game graphics (for example) at the same time. This works reasonably well, although the overhead involved at the beginning and end of each DMA transfer requires either two separate channels, an on-board FIFO buffer, or brief silent periods while the CPU sets up the next chunk of the sound file.

Regardless of the CPU speed, the ISA bus DMA hardware executes roughly one transfer per microsecond. Because the ISA bus can handle at most two bytes of data on each bus cycle, DMA transfers run at 1 or 2 MB/s. That's slightly less than the rates we've measured throughout the book, so, contrary to popular belief, using DMA does *not* move data at a particularly high speed!

If you're familiar with microcontrollers that include DMA support, you know that you can generally gain a significant speed advantage by block-moving memory with DMA hardware. On any PC faster than a '386, however, the *x86* `REP MOVSB` instruction outruns the system board's DMA hardware by a considerable margin, with considerably less overhead.

Surprise!

So... unless you have a compelling reason to use DMA, don't bother. If you absolutely must use DMA, consult Solari's book for the myriad details required to get it working on all possible systems. There are significant differences in the system board hardware available that will affect how you design your board.

The PCI bus is a better choice for systems with exceedingly stiff response time specs and high data transfer rates, despite the additional complexity of designs using that bus. If you find your project becoming encrusted by the PC Compatibility Barnacles on the ISA bus, the PCI bus may provide a clear path.

My back of the envelope estimate says that you can do about as well as ISA bus DMA with a moderately large FIFO buffer and a well written interrupt handler transferring data across the bus using `REP OUTSB` and `REP INSB` instructions. You'll avoid all the hassles of finding and using a DMA channel, too.

Busmastering Boards

Pin D17 on the ISA bus sports the label **-Master**. In principle, when an ISA bus board asserts that signal the system board relinquishes the bus and allows the board to drive the bus control lines. In practice, this doesn't work nearly as well as you'd expect, because the ISA bus was designed to support a single CPU on the system board that controlled a bunch of relatively dumb peripheral boards.

An ISA bus board that would become the master must first set up a DMA transfer using a DMA channel. When the DMA controller asserts that channel's **DRQ** line, the board responds by asserting the corresponding **-Dack** signal and then asserting **-Master**. At that point, the DMA controller has released the bus and will not proceed with the next DMA cycle.

A busmaster board must allow normal DRAM refresh cycles by either asserting **-Refresh** every 15.6 μ s or releasing control of the bus. This can pose a problem if you expect to process large blocks of data at high speeds with a CPU on an ISA bus board. In any event, you certainly won't get the bus throughput you might expect.

I've worked with intelligent ISA bus boards on several projects and can say that the best bus interface is the simplest. If you can *possibly* get along with a small block of shared memory, similar to the Graphic **LCD Refresh RAM**, don't bother with a busmastering board. Even better, if you can pass data through a few I/O ports, you've reduced your board's footprint to the point where it's certain to work.

Unfortunately, that simplicity may not be feasible if your board must share and process huge blocks of data. In that case, rather than force-fit the project into the ISA bus, it's time to break out those PCI design tools and do the job right.

Release Notes

Gather your tools, collect some parts, and settle down for some construction!

Remember to check the Web page for more information and send me a note when you come up with a neat trick.

Now it's *your* turn...

Sources

You can obtain all of the hardware and software mentioned in this book from mail order sources, if you can't find it closer to you.

For those of you outside North America, I've included as many non-800 numbers as I can find. Note that some companies do not ship orders outside the US and Canada or may add significant shipping charges to cover customs and insurance.

If you're not Internet-enabled, get started *now*. Nearly all of these companies have complete information available on their Web sites, online ordering, and email access to tech support. You'll need the latest graphical Web browsers, as the days of text-only access are long gone. URLs change constantly, so plan on some rummaging around using any of the search engines.

Semiconductor Manufacturers

Nowadays, the single most effective way to get chip data sheets and application notes is through the Internet. You will also need a copy of Adobe Acrobat to read and print the files; the source and directions for downloading Acrobat appear on the relevant Web pages.

AMD

www.amd.com

Dallas Semiconductor

www.dalsemi.com

Intel Corporation

www.intel.com/design

Maxim Integrated Products

www.maxim-ic.com

Microchip Technology

www.microchip.com

Publications

Now that the PC revolution isn't quite so new, you'll find far too many "me too!" books loading your local bookstore shelves. In contrast to that drivel, these magazine and book publishers provide the technical information required to actually design and build things, not just use the commercial products *du jour*. Here, too, the Web provides convenient access if you're not near a good bookstore... or when you need a book that, for some reason, they can't order!

Addison Wesley Longman

www.awl.com

One Jacob Way, Reading MA 01867
Orders 800-824-7799, 617-944-3700
Fax 617-944-7273

Annabooks

www.annabooks.com

11838 Bernardo Plaza Court, San Diego, CA 92128-2414
800-462-1042, 619-673-0870
Fax 619-673-1591

C/C++ Users Journal

www.cuj.com

PO Box 52582, Boulder CO 80328-2582
Subscriptions 800-365-1364, 303-678-0439, Fax 303-661-1885

Circuit Cellar INK

www.circellar.com

4 Park Street, Vernon CT 06066
860-875-2751
Reprints 860-875-2199
BBS 860-871-1988
Subscriptions 800-269-6301

Coriolis Developer's Club

Visual Developer (formerly PC Techniques)

www.coriolis.com

14455 N. Hayden Road, Suite 220, Scottsdale AZ 85260
800-410-0192, 602-483-0192
Fax 602-483-0193

Embedded Systems Programming

www.embedded.com

PO Box 420235, Palm Coast FL 32142-0235

800-829-5537, 415-905-2200

Peer-to-Peer Communications, Inc.

www.peer-to-peer.com

P.O. Box 640218, San Jose CA 95164-0218

800-420-2677, 408-435-2677, Fax 408-435-0895

To round out the collection, try these Internet sources. As with many mail-order suppliers, their normal discounts offset the shipping charges, at least if you order several titles at once.

Amazon.com Books

www.amazon.com

The Book Pool

www.bookpool.com

The Book Site

www.booksite.com

If you use the Intel α 86 architecture, you should drop by the Unofficial Intel Secrets site for lots of undocumented hints and tips that can help you get results when all else fails. For those of us working near the bare silicon, sometimes the Official Sources don't have quite enough detail.

The Unofficial Intel Secrets Site

www.x86.org

Hardware

These companies provide new, commercial electronic parts in reasonably small quantities. You should also consult your phone book for local electronic part distributors that represent major manufacturers.

Digi-Key Corporation

www.digikey.com

701 Brooks Avenue South, Thief River Falls MN 56701-0677

800-344-4539, 218-681-6674

Fax 218-681-3380

Jameco Electronic Components

www.jameco.com

1355 Shoreway Road, Belmont CA 94002

800-831-4242, 415-592-8097

Fax 800-237-6948, 415-592-2503

JDR Microdevices

www.jdr.com

1850 South 10th Street, San Jose CA 95112

800-538-5000, 408-494-1400

Fax orders 800-538-5005, 408-494-1420

After 1 Aug 97, use area code 650 for these numbers.

MCM Electronics

650 Congress Park Drive, Centerville OH 45459-4072

800-543-4330, Fax 937-434-6959

Software

These companies created many of the tools I used throughout the book. Even for pure embedded systems work, you'll need a zillion small utilities and programs to keep your PC running. The Web is a good source of both information and programs, although you must be wary of viruses and nuisances.

Dunfield Development Systems

www.dunfield.com

PO Box 31044, Nepean, Ontario K2B 8S8 CANADA

Automated voice response 613-256-5820

Fax 613-256-5821

Paradigm Systems

www.devtools.com

3301 Country Club Road, Suite 2214, Endwell NY 13760

800-537-5043, 607-748-5966

Fax 607-748-5968

Phar Lap Software, Inc.

www.pharlap.com

60 Aberdeen Avenue, Cambridge MA 02138

617-661-1510, Fax 617-876-2972

Mansfield Software Group, Inc.

www.kedit.com

PO Box 532, Storrs CT 06268

860-429-8402, Fax (860) 487-1185

Microsoft Corporation

www.microsoft.com

Borland Corporation

www.borland.com

Surplus Electronics

If you're building a one-of-a-kind gadget, surplus electronics can provide features and functions you simply couldn't afford from the usual retail sources. Make certain that you obtain the data sheets, though, as the technical support folks may not know much more than you do about a particular device!

All Electronics Corporation

www.allcorp.com

PO Box 567, Van Nuys CA 91408-0567

Orders 800-826-5432, Information 818-904-0524

Fax 818-781-2653

B. G. Micro

www.bgmicro.com

PO Box 280298, Dallas TX 75228

Orders 800-276-2206, 972-271-5546

Tech support 972-271-9834

Fax 972-271-2462

Marlin P. Jones and Associates

www.mpja.com

PO Box 12685, Lake Park FL 33403-0685

Orders 800-652-6733, Information 561-848-8236

Fax 561-844-8764, Fax orders 800-432-9937

Mendelson Electronics, Inc. (MECI)

www.meci.com

340 East First Street, Dayton OH 45402

Orders 800-344-4465, Information 513-461-3525

Fax 800-344-6324, 513-461-3391

Timeline Inc.

www.digisys.net/timeline

2539 West 237th Street, Building F, Torrance CA 90505

Orders 800-872-8878, Information 310-784-5488

Fax 310-784-7590

Bibliography

This appendix lists some of the books I've found useful in my projects. Due to the incredible pace of publishing in the PC industry, many books go out of print within a year or two, with the sad result that you may not be able to find these at your local bookstore. New revisions of these books, with new ISBN codes, may also appear, so expect to search for them based on the author or title as well as ISBN.

Check the Sources appendix for places to find these books.

The Zen of Assembly Language, Volume 1: Knowledge

Michael Abrash

Scott, Foresman and Company, 0-673-38602-3

Zen of Code Optimization

Michael Abrash, Coriolis Group Books, 1-883577-03-9

Abrash is the high priest of PC code optimization and these books summarize his magic. *Zen of Code Optimization* includes key chapters from his earlier (and long out of print) work, along with expanded information from his *Dr. Dobb's Journal* columns. Believe me, if you follow his precepts, you need not apologize to anybody for anything you write!

Parallel Port Complete: Programming, Interfacing, and Using The PC's Parallel Printer Port

Jan Axelson, Lakeview Research, 0-9650819-1-5

Distributed by Peer-to-Peer Communications

Finally, a complete guide to the PC's parallel port! If you've wondered how to use the Enhanced Parallel Port or Extended Capabilities Port hardware found on your PC, this is the book you need. The software includes DLLs for direct, controlled port access from those GUI Windows 95 programs.

PC Interrupts, Second Edition

Ralf Brown and Jim Kyle

Addison Wesley Longman, 0-201-62485-0

A huge tabulation of every hardware and software interrupt used by every PC program inside the asteroid belt. Each entry includes a brief functional description, input and output register contents, warnings on conflicts with other interrupts, and occasionally some functional details. The text may be available online from several sources, but I suspect the book is cheaper than the download charges.

Microprocessors: A Programmer's View

Robert Dewar and Matthew Smosna

McGraw-Hill, 0-07-016639-0 cloth, 0-07-016638-2 paper

A savage romp through CPU taxonomy. You get a detailed explanation of CPU functions and addressing modes, lengthy descriptions of the 80386 and 68030, the 387 coprocessor, as well as a chapter apiece devoted to the SPARC, Intel 860, IBM RISC, and Transputer RISCs. After reading about the 80386 protection mechanisms you find a section titled "Is All This Worthwhile?" with the memorable sentence "The previous section is virtually incomprehensible." Don't miss it!

The Undocumented PC, Second Edition

Frank Van Gilluwe

Addison Wesley Longman, 0-201-47950-8

Unlike the other "Undocumented" books you've seen, this one covers *hardware*. It skips over information you can find elsewhere to cover the hidden functions and features within nearly every PC chip. Whether you need to know about the CPU instructions you won't find in the Intel manual, how the keyboard hardware works, or all the data hidden in the Real-Time Clock RAM in an EISA bus PC, you'll find it here. Lots of sample code, diagnostics, and test routines.

The Programmer's PC Sourcebook, Second Edition

Thom Hogan

Microsoft Press, 1-55615-321-X

A category killer book with nearly everything you could possibly want to know about your PC, albeit in desiccated tabular format. Sporting nary a paragraph of narrative text, you must depend on other sources for the *why* behind each function. This is the book I reach for first when trying to decipher a connector, socket, interrupt, DOS/BIOS function, or chip pinout. Other books have the explanations, but this one covers the territory. I've run across a few errors, so don't make this one the only book on your shelf.

PC Magazine Programmer's Technical Reference: The Processor and Coprocessor

Robert Hummel

Ziff-Davis Press, 1-56276-016-5

As it says on the back cover, "Finally — a single, concise technical reference on Intel's entire family of 80x86 processors and 80x87 coprocessors." Exhaustive detail on how the machinery at the heart of your PC munches instructions and data, with lots of info on protected mode operation and memory management. It includes a chip bug summary by CPU mask step, with workarounds and hints. Unfortunately, I haven't seen a more recent edition than the 1992 copy on my shelf.

Writing Solid Code

Steve Maguire

Microsoft Press, 1-55615-551-4

How to write code that works dependably, assure yourself (or your boss) that it will meet its specifications, and how to armor it against unforeseen gotchas. Although I'm no longer such a big fan of Hungarian notation, I do agree with his advice that single stepping through your code can reveal more problems in a shorter time than the fanciest ICE. Apply this book's suggestions early and often to all your projects.

Code Complete

Steve McConnell

Microsoft Press, 1-55615-484-4

As the cover blurb puts it, "A practical handbook of software construction." While not directly covering embedded programming, the precepts in this book convert the often confusing advice about high level design into practical techniques you can use in your own projects. The advice on naming variables and formatting your source code may just pay for the book... wish I'd had this one a long time ago!

Macro Magic with Turbo Assembler

Jim Mischel

John Wiley & Sons, 0-471-57815-0

This book suffers from a terminally cute title. It will, however, give you the dope on an aspect of assembler programming that you won't find elsewhere. He even builds a useful language entirely in macros.

Serial Communications: A C++ Developer's Guide

Mark Nelson

M&T Publishing, 1-55851-281-0

Firmware with a somewhat high-level language bias. A good introduction to low-level operations, although I fear that C++ has progressed to the point where much of the speed available in earlier versions has gone away.

System BIOS for IBM PC/XT/AT Computers and Compatibles

Phoenix Technologies, Ltd

Addison Wesley Longman, 0-201-51806-6

The straight dope on one vendor's BIOS implementation: good descriptions of what all the functions and data areas actually do. Although it certainly will not match up precisely with the BIOS in your machine, it should give you a running start on the standard functions. It is considerably more verbose than the corresponding Official IBM BIOS documentation, which is not a bad thing.

Undocumented DOS, Second Edition

Schulman, Brown, Maxey, Michaels, Kyle

Addison Wesley Longman, 0-201-63287-X

How It All Works, outward from the most ugly and hidden innards. Lots of explanations, sample code, tables, and a few diagrams tell you pretty nearly everything you need to know about the versions of DOS leading up to Windows 95. If you plan to work down near the bare metal, you'll need this information to prevent you from messing up the details.

The SCSI Bus and IDE Interface

Friedhelm Schmidt

Addison Wesley Longman, 0-201-42284-0

If you must write DOS-less drivers for hard disks and other devices, this book will give you the information you need. Because it's mostly a book about SCSI by a SCSI expert, you won't find the EIDE and ATAPI variations on the IDE theme.

The Mindshare, Inc. PC System Architecture Series:

PCMCIA System Architecture 0-201-40991-7

Pentium System Architecture 0-201-40992-5

PCI System Architecture 0-201-40993-3

80486 System Architecture 0-201-40994-1

EISA System Architecture 0-201-40995-X

ISA System Architecture 0-201-40996-8

CardBus System Architecture 0-201-40997-6

Plug-and-Play System Architecture 0-201-41013-3

Protected Mode Software Architecture 0-201-55447-X

Tom Shanley and Don Anderson

Addison Wesley Longman

This series of books contains excellent x86 information drawn from Mindshare's engineer training courses and give a good overview of the entire subsystem. Use these books in addition to your hardware data books, not as replacements.

ISA System Architecture includes a thorough discussion of ISA bus mechanics. You must also have Solari's book if you're building hardware that must work on many systems, but this book is far more readable. The *80486* and *Pentium System Architecture* books cover CPU details, rather than bus level issues.

The illustrations resemble the overhead projector block diagrams you find in seminars, rather than crisp schematics, and you'll find nary a line of sample code. Although some background information tells you *why* a feature might be useful, the books do not provide system level design explanations.

AT Bus Design

Edward Solari

Annabooks, 0-929392-08-6

ISA & EISA Theory and Operation

Edward Solari

Annabooks, 0-929392-15-9

All the gory details required to get a board running in a PC using the ISA or EISA bus, documented with timing diagrams, signal descriptions, and cautionary notes.

Information that you just cannot find anywhere else, written by somebody who knows whereof he speaks. *ISA & EISA Theory and Operation* replaces *AT Bus Design*, with a better presentation and more explanatory notes.

Acronyms and Abbreviations

Sometimes, it seems that conversations about electronics and computers contain more acronyms than real words. This appendix lists some of the acronyms I've come across or invented, many of which appear in the rest of this book.

The current practice seems to use all caps for acronyms with four or fewer letters, as in CRT. Longer acronyms, particularly well-established ones that *everyone* knows, have just an initial cap or all lowercase letters: Radar or radar. In this book, I've used all caps for nearly every acronym because they seem to be more familiar that way: EEPROM rather than Eeprom.

aka	Also Known As (sometimes, a.k.a.)	IBF	Input Buffer Full
ALU	Arithmetic-Logic Unit (the guts of your PC's CPU chip, often the smallest part)	IDT	Interrupt Descriptor Table
ANSI	American National Standards Institute	IDTR	Interrupt Descriptor Table Register
ASM	ASeMbler (refers to either source code file or the program that processes ASM files)	IF	Interrupt Flag
BIOS	Basic Input-Output System	INK	<i>Circuit Cellar INK</i> magazine
C	Carry Flag (sometimes CF)	INT	INTerrupt (usually: INT signal or INT pin)
CCFT	Cold Cathode Fluorescent Tube (a high-performance LCD backlight)	Int	Interrupt instruction (followed by the interrupt number in hex or decimal)
CCBBS	Circuit Cellar Bulletin Board System	INTA	INTerrupt Acknowledge (CPU output pin)
CMOS	Complementary Metal Oxide Semiconductor	INTR	INTerrupt Request (CPU input pin)
COM	COMmand file (executable binary program)	IOPL	I/O Privilege Level
CPL	Current Privilege Level	IRET	Interrupt Return (CPU instruction)
CPU	Central Processing Unit (the guts of your PC)	IRED	Infra Red LED (also IRLED)
CRC	Cyclic Redundancy Check (or Code)	IRQ	Interrupt ReQuest (signal on ISA bus)
CRT	Cathode Ray Tube (your PC's monitor)	ISA	Industry Standard Architecture (bus)
DBR	Disk(ette) Boot Record	ISR	In-Service Register (8259 controller only)
DOS	Disk Operating System	ISR	Interrupt Service Routine
DPL	Descriptor Privilege Level	k	1000 decimal, as in 1 k Ω
EEPROM	Electrically Erasable PROM	Kb	kilobit = 1024 bits
EISA	Enhanced ISA (bus)	KB	kilobyte = 1024 bytes
EOI	End Of Interrupt (command)	kludge	Not an acronym, a truly ugly solution that barely works.
EPROM	Electrically Programmable ROM	LCD	Liquid Crystal Display (thus, the familiar term "LCD Display" is redundant)
EXE	EXEcutable program file	LDT	Local Descriptor Table
FDB	Firmware Development Board	LDTR	LDT Register
FFTS	Firmware Furnace Task Switcher	LED	Light Emitting Diode
G bit	Granularity bit (in a PM descriptor)	LPT	Line PrinTer (usually suffixed with a digit specifying which one: LPT1)
GDT	Global Descriptor Table	LSB	Least Significant Bit or Byte
GDTT	GDT Register	LSI	Large Scale Integration
GPF	General Protection Fault	MASM	(Microsoft) Macro Assembler
		MB	megabyte = 1048576 bytes
		MCA	Micro Channel Architecture

The Embedded PC's ISA Bus

MDA	Monochrome Display Adapter
MIPS	Million Instructions Per Second (sometimes written as MIP with the S understood)
MSB	Most Significant Bit (or Byte)
NAND	Not-AND boolean function
NOR	Not-OR boolean function
NT	Nested Task
NVRAM	Nonvolatile RAM
OBF	Output Buffer Full
OEM	Original Equipment Manufacturer
P bit	Present bit (in a PM descriptor)
PCI	Peripheral Component Interconnect bus
PM	Protected Mode
POST	Power-On Self Test (typically used for the BIOS POST)
ppm	parts per million
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RF	Resume Flag
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RPL	Requestor Privilege Level
RTC	Real-Time Clock
RTFM	Read The <i>Fine</i> Manual
SSI	Small Scale Integration (old chips!)
SVGA	Super VGA (nearly anything above 640×480 dots)
TASM	(Borland) Turbo ASseMbler
TF	Trap Flag
TR	Task Register
TSS	Task State Segment
TTY	TeleTYpe
UART	Universal Asynchronous Receiver Transmitter (serial I/O chip)
V86	Virtual-86 protected mode
VESA	VGA Extensions Standards Association
VGA	Video Graphics Array
VLB	Video (or VESA) Local Bus
VM	Virtual Machine (in EFLAGS)
WIBNI	Wouldn't It Be Nice If...
XOR	eXclusive-OR boolean function

Schematics

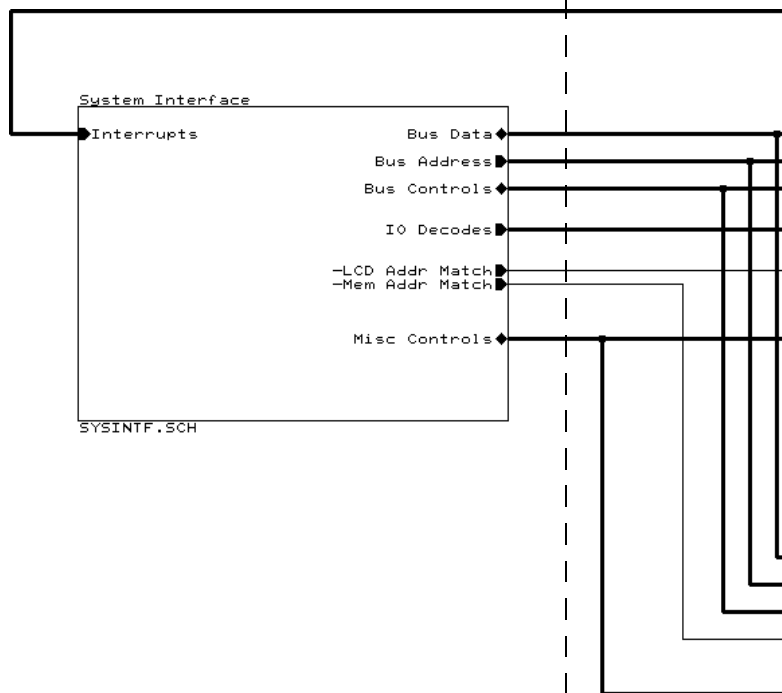
This appendix has all of the original schematic drawings, in their proper order, for your perusal. The schematics accompanying the text in each chapter are slightly different, because I omitted some connections that weren't relevant to the subject at hand. Here, you'll get the full story.

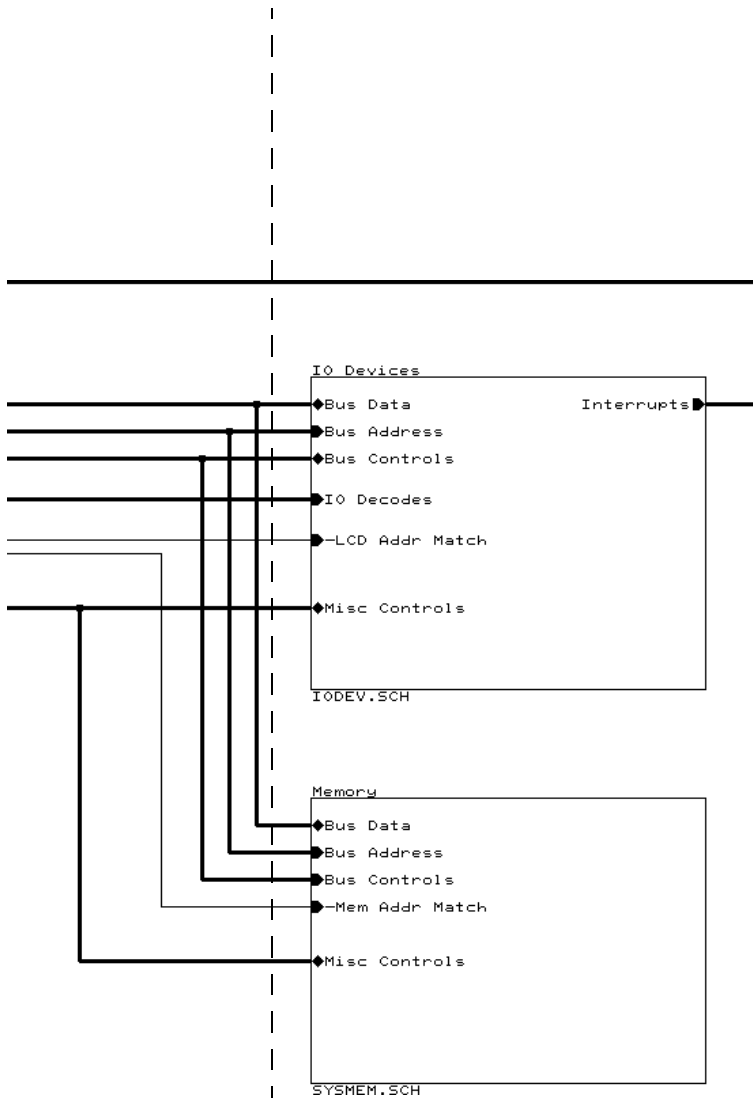
Each pair of facing pages holds a single schematic page with about an inch of overlap. The two dotted lines near the gutter in the middle mark the overlap boundaries, so you can rejoin the pieces fairly easily: squash the book down on a copier, blip the Copy button, trim one half to the line, overlap it with the other, and paste 'em together.

The source code diskette includes these schematics as **TIFF** files that you can load into nearly any graphics program and print on your own printer. After you decompress them, they'll take up about 3.5 MB on your disk and even more when you import them as graphic files. Check the **ReadMe** file for more information!

You'll find tables listing all the I/O ports, control and status bits, and memory addresses, as well as a Bill of Material, after the schematics. You can use those tables as a ready reference as you trace through the schematics.

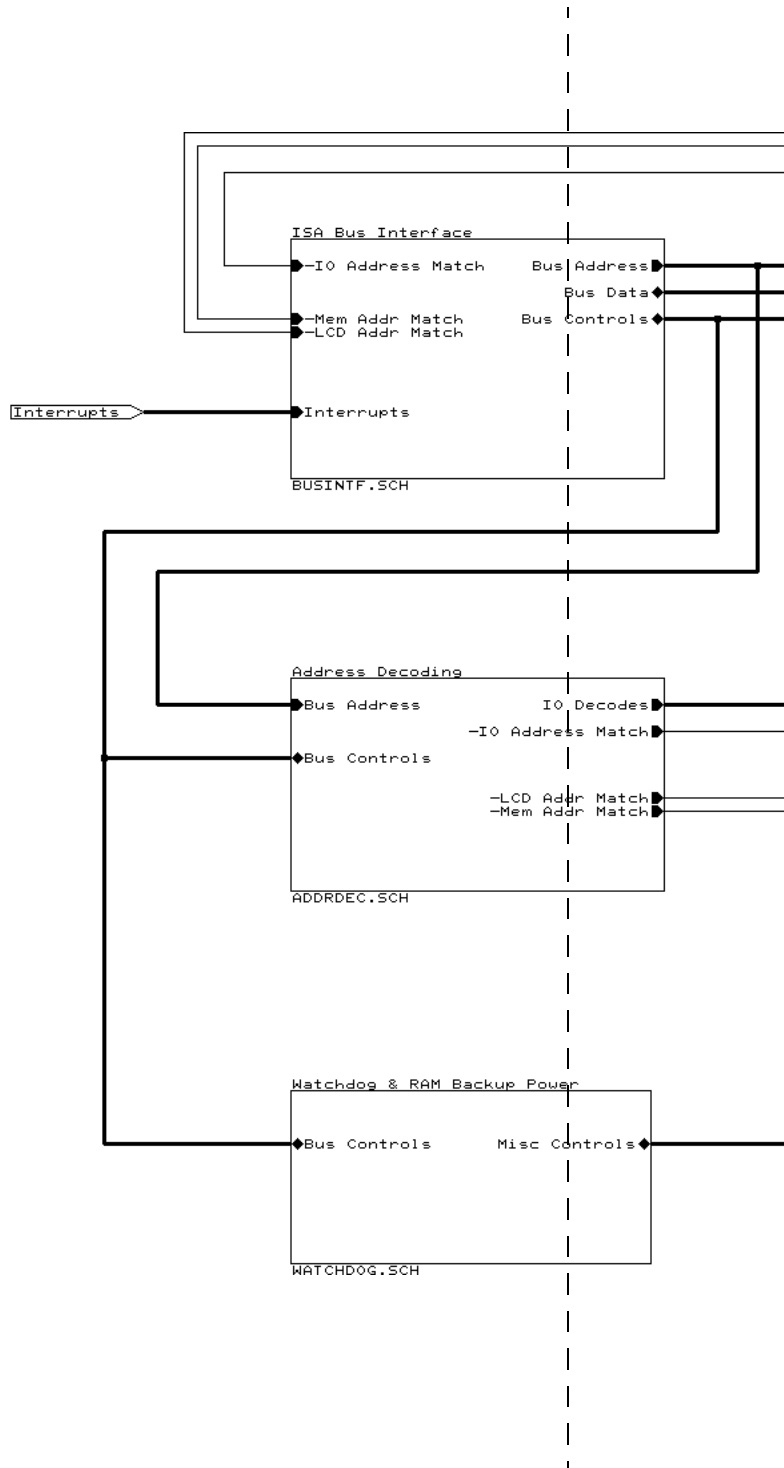
Enjoy!

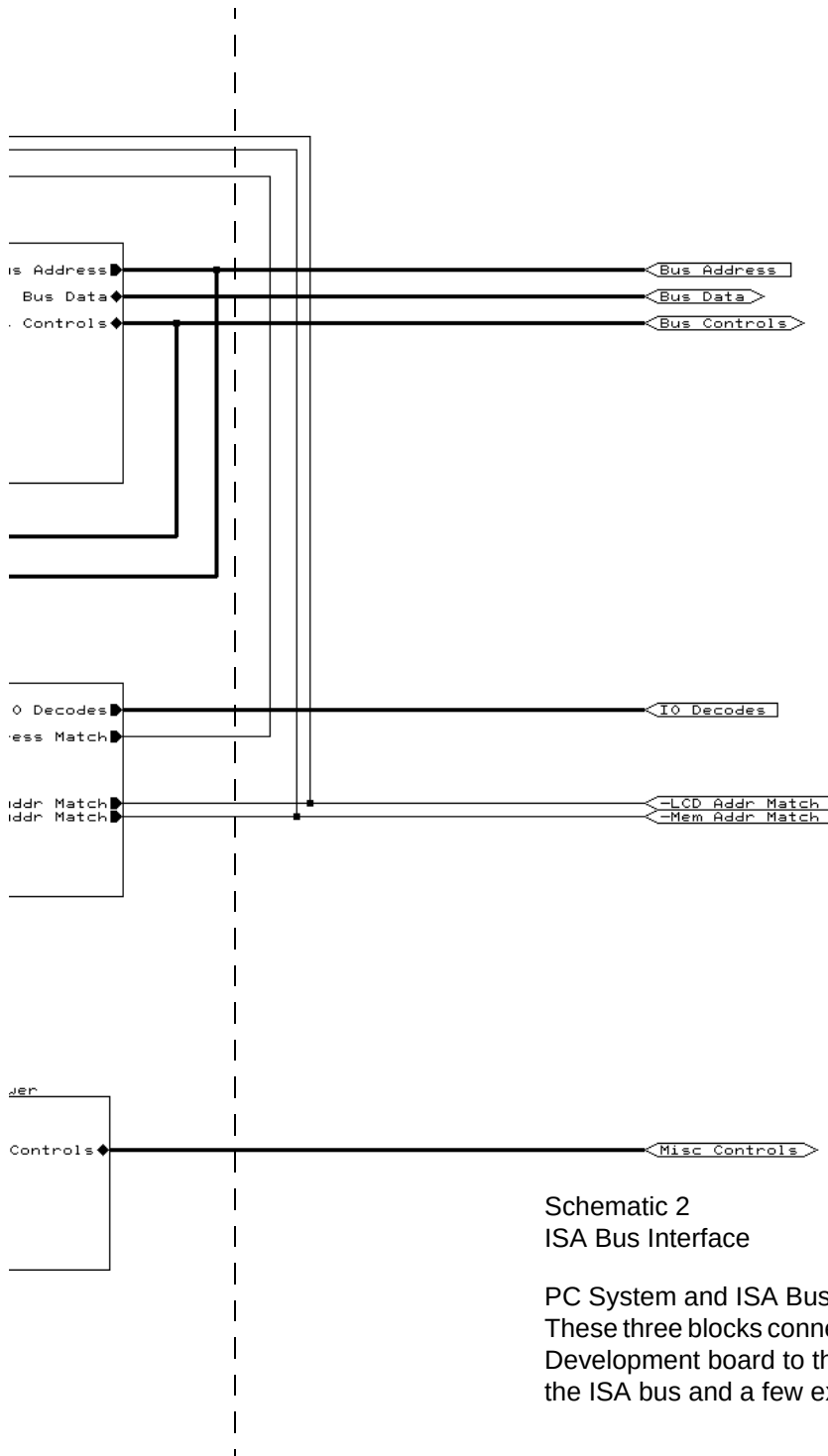


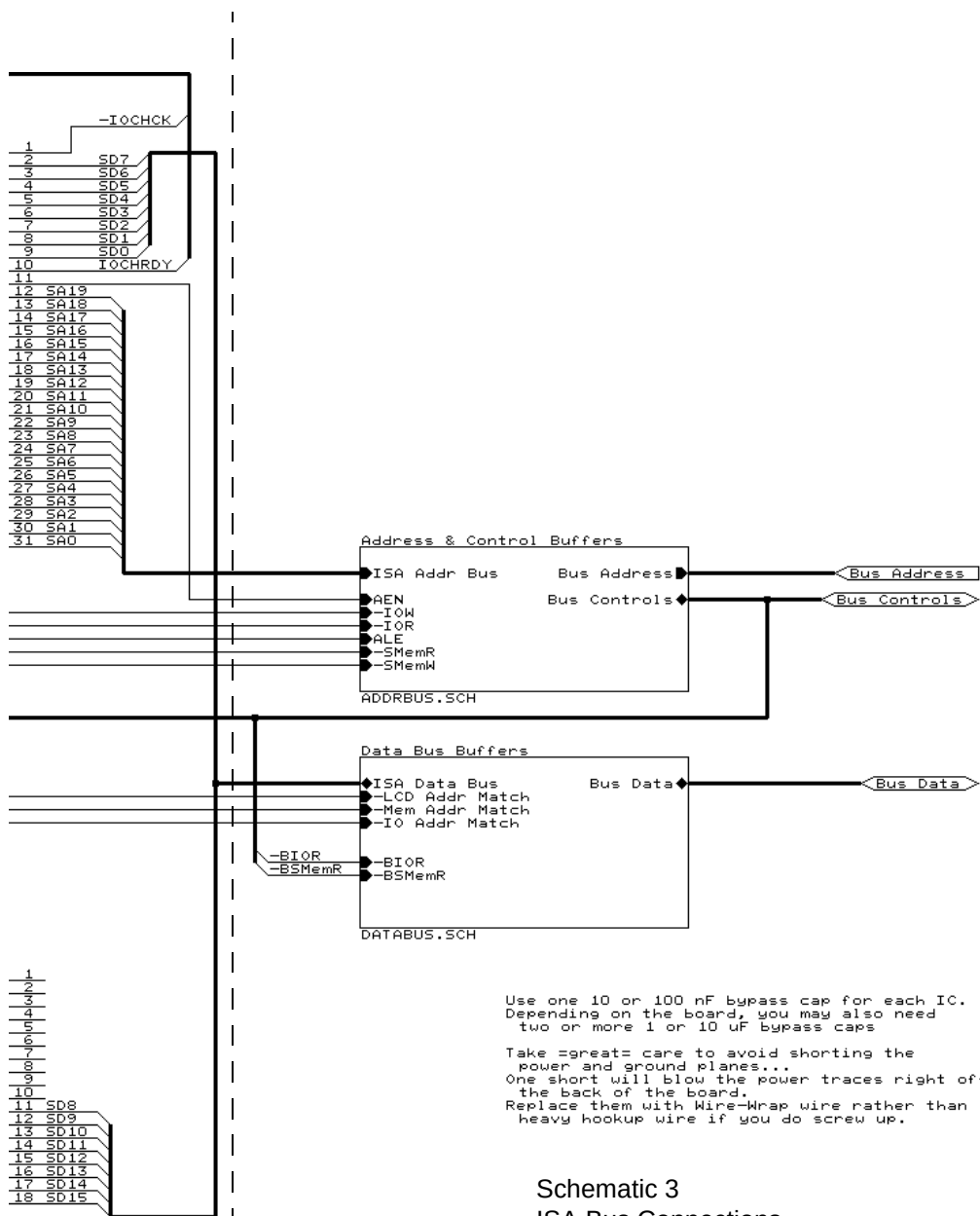


Schematic 1
Board Overview

The Firmware Development Board has three major functional blocks. The next pages explore each block in detail.

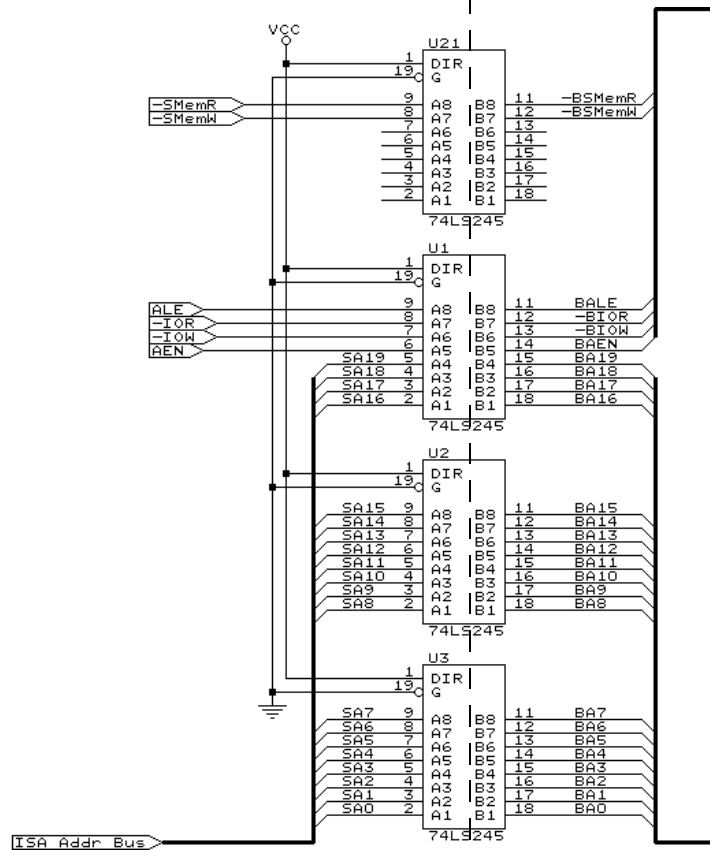


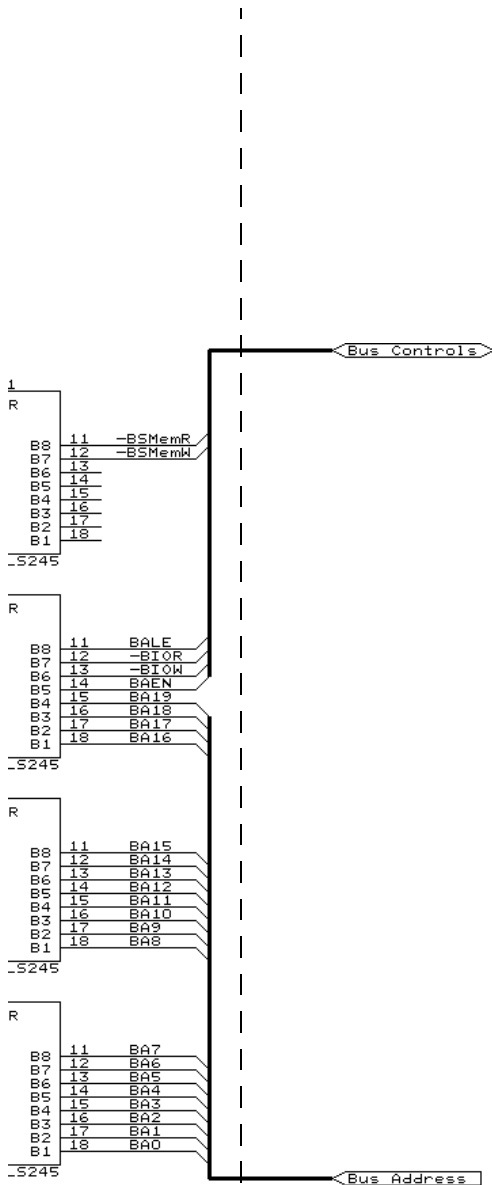




Schematic 3
ISA Bus Connections

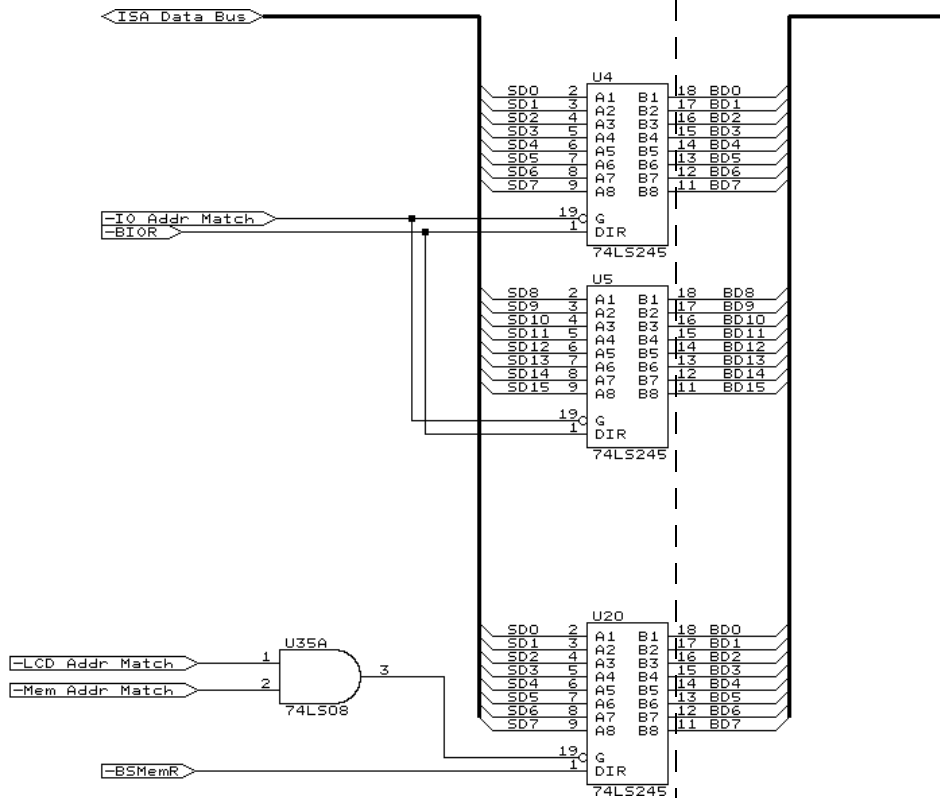
The two blocks on the left represent the ISA bus connectors. The A & C pins are on the component side of the board, the B & D pins are on the solder side.

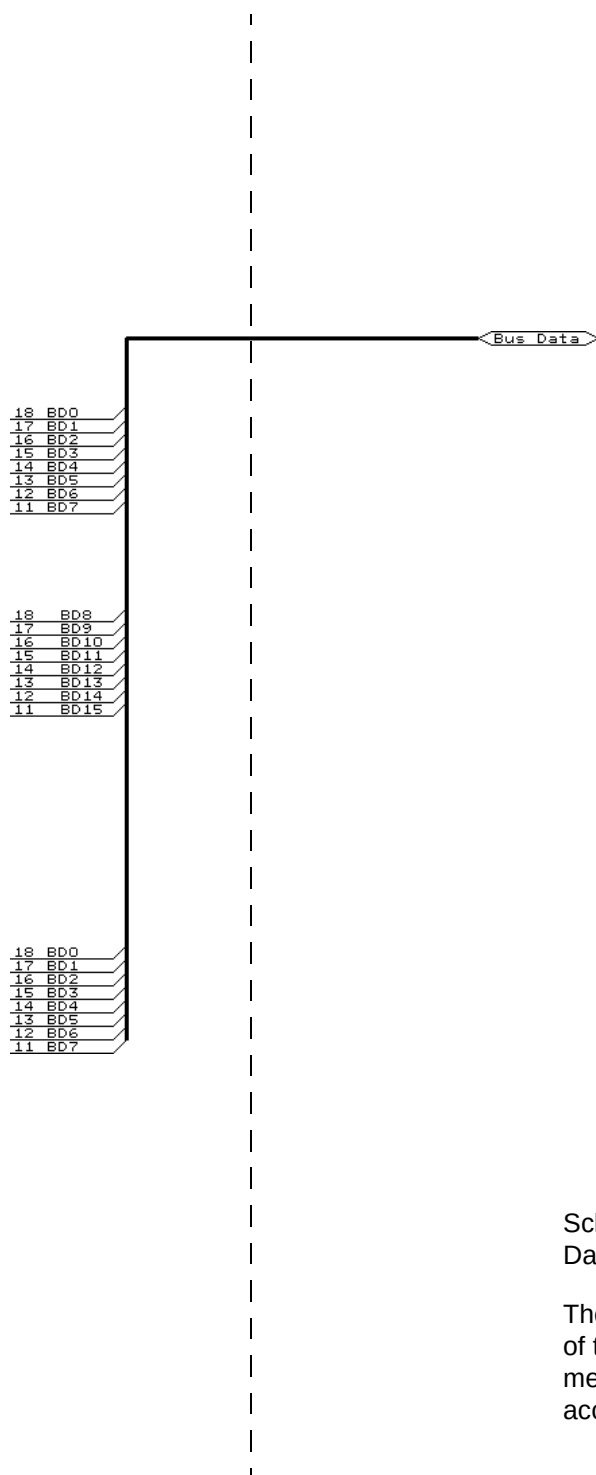




Schematic 4
Address & Control Buffers

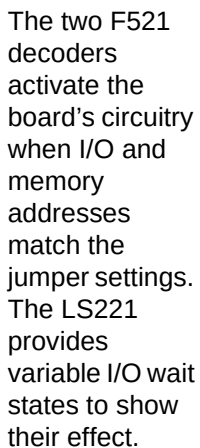
These chips provide the proper isolation and loading on the ISA bus. The buffers are always active, so their outputs always track the current bus signals.

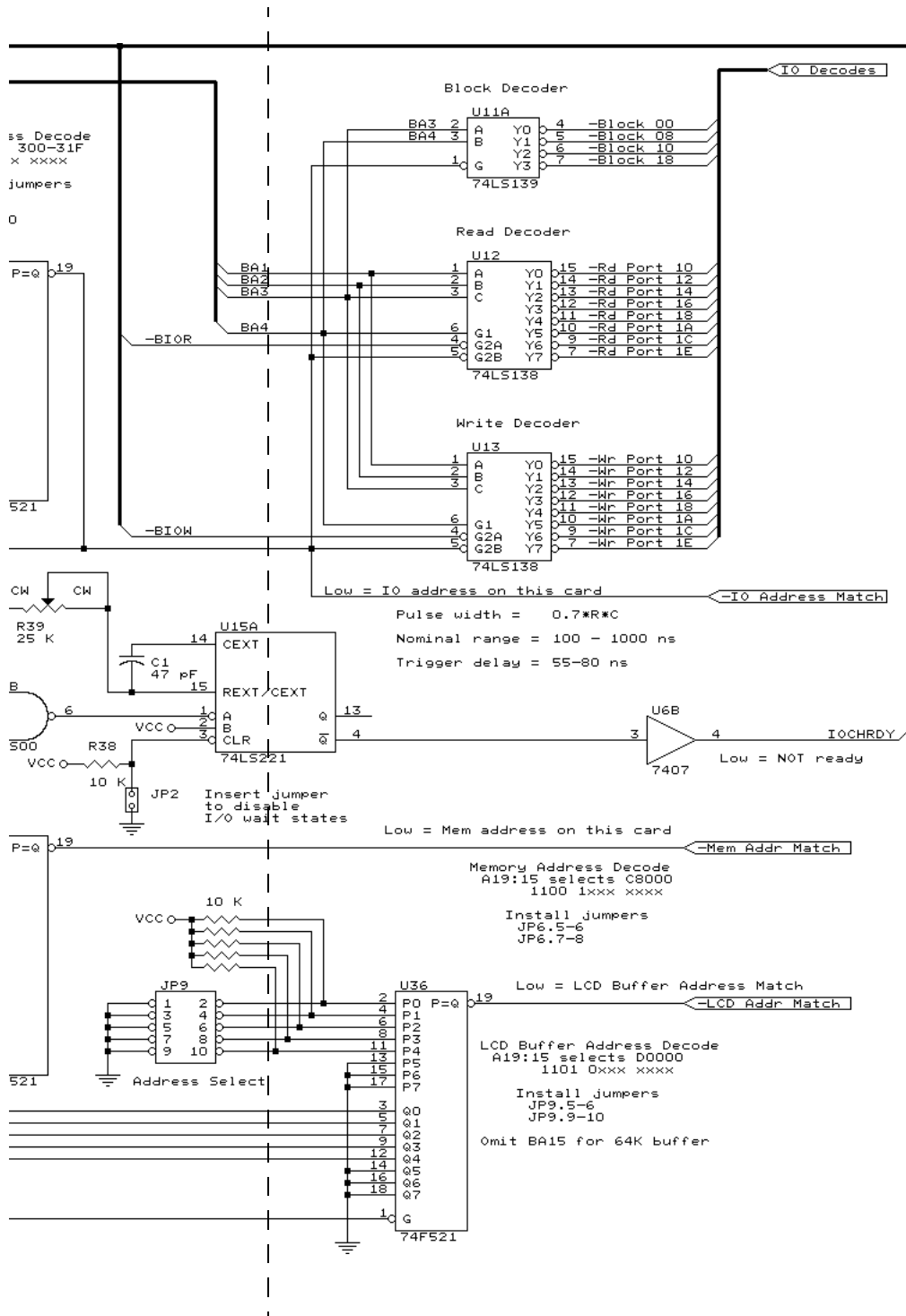




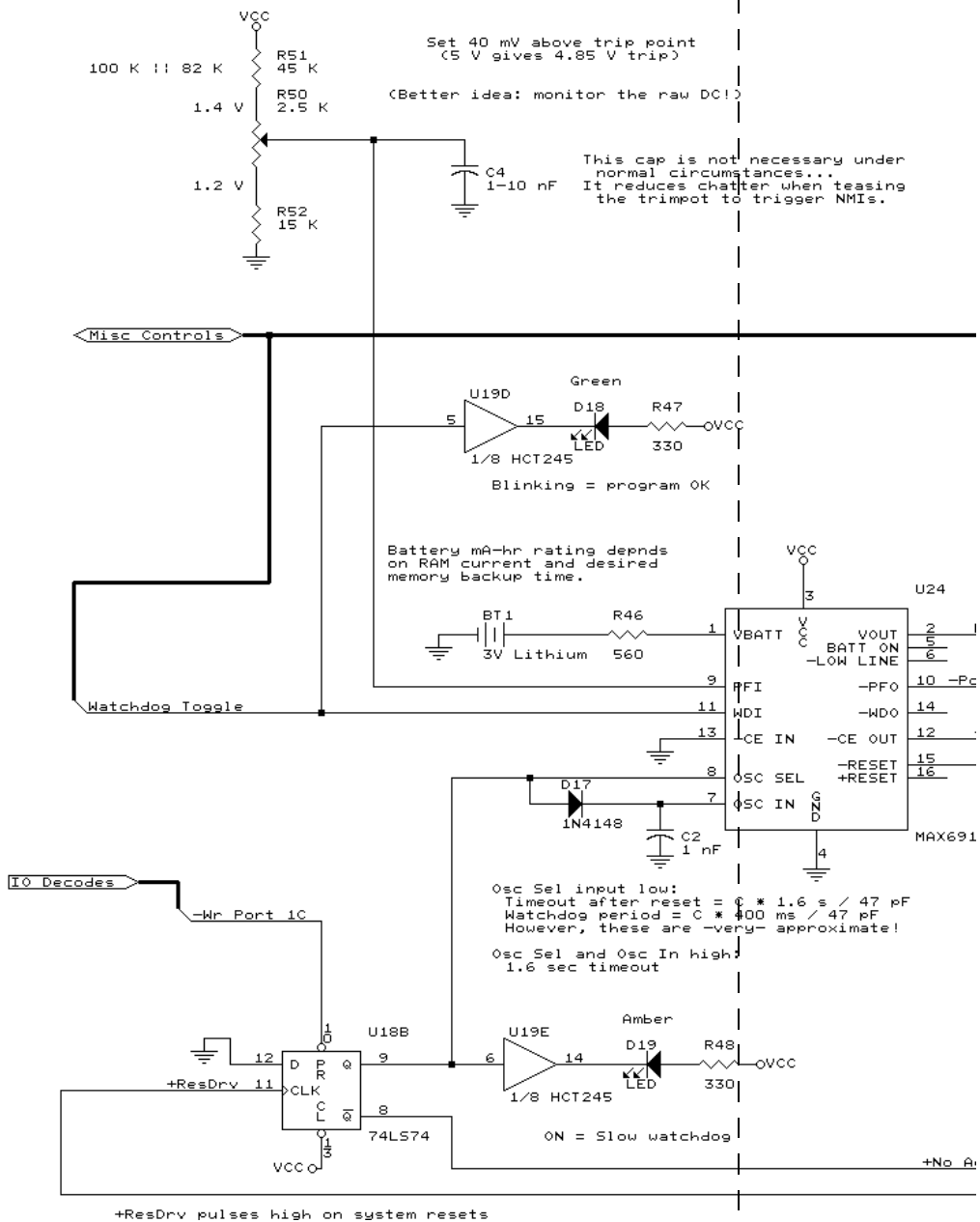
Schematic 5
Data Bus Buffers

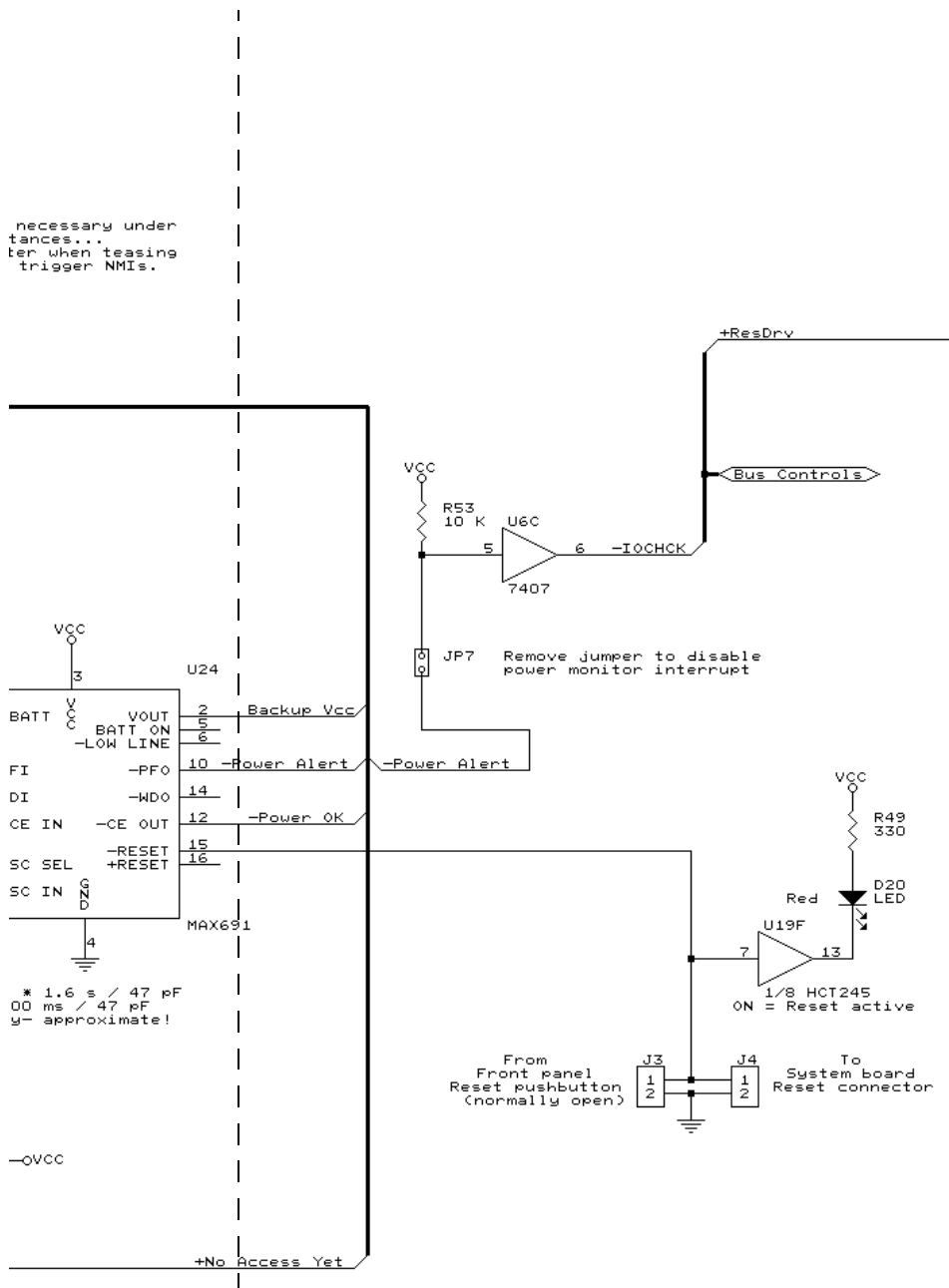
These chips provide bidirectional control of the data bus signals for I/O and memory accesses. Note that memory accesses use 8-bit transfers, not 16-bit.





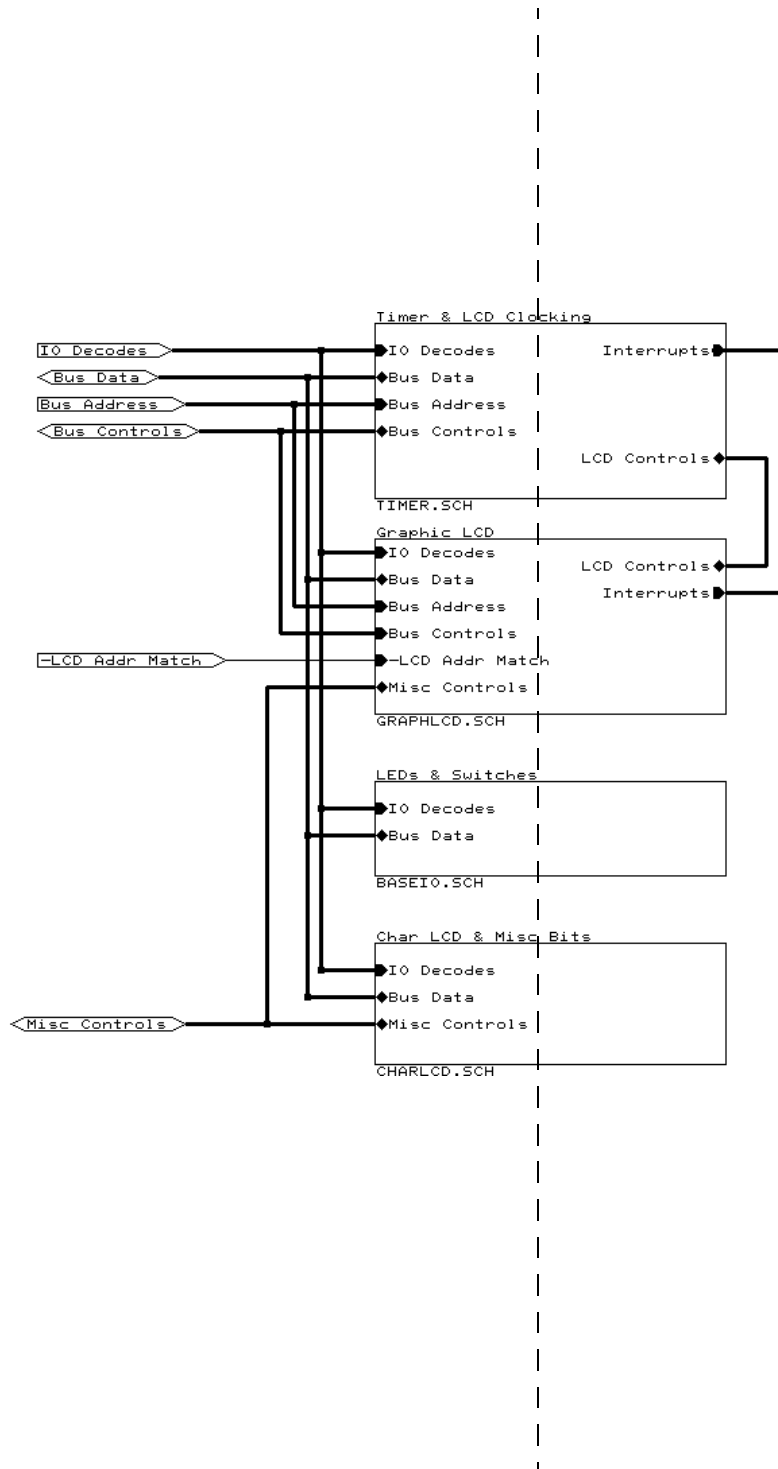
The Embedded PC's ISA Bus

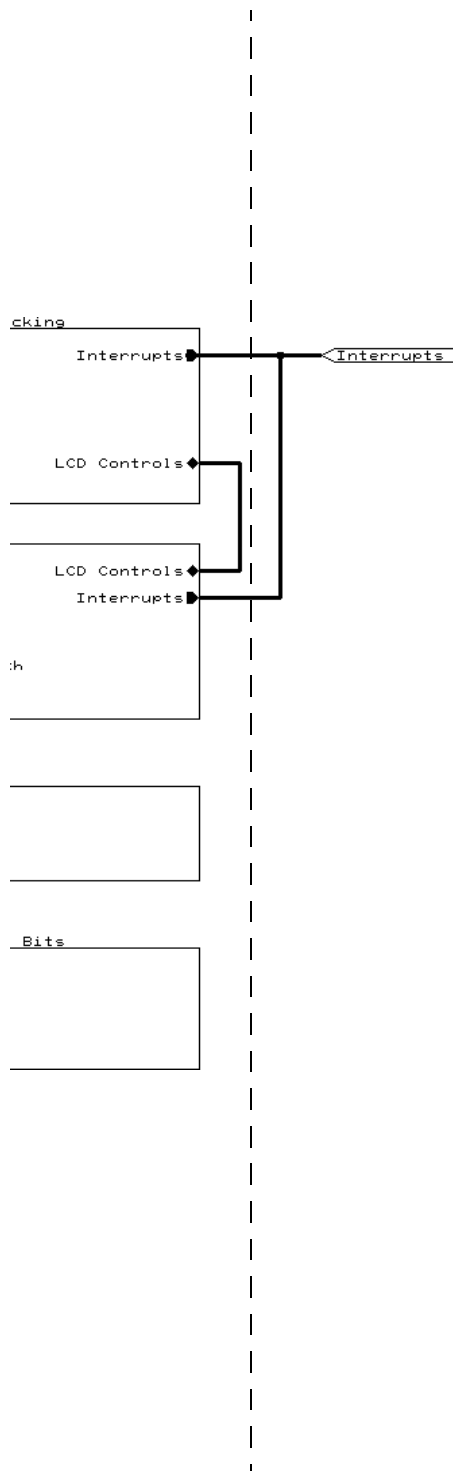




Schematic 7: Watchdog and Battery Backup

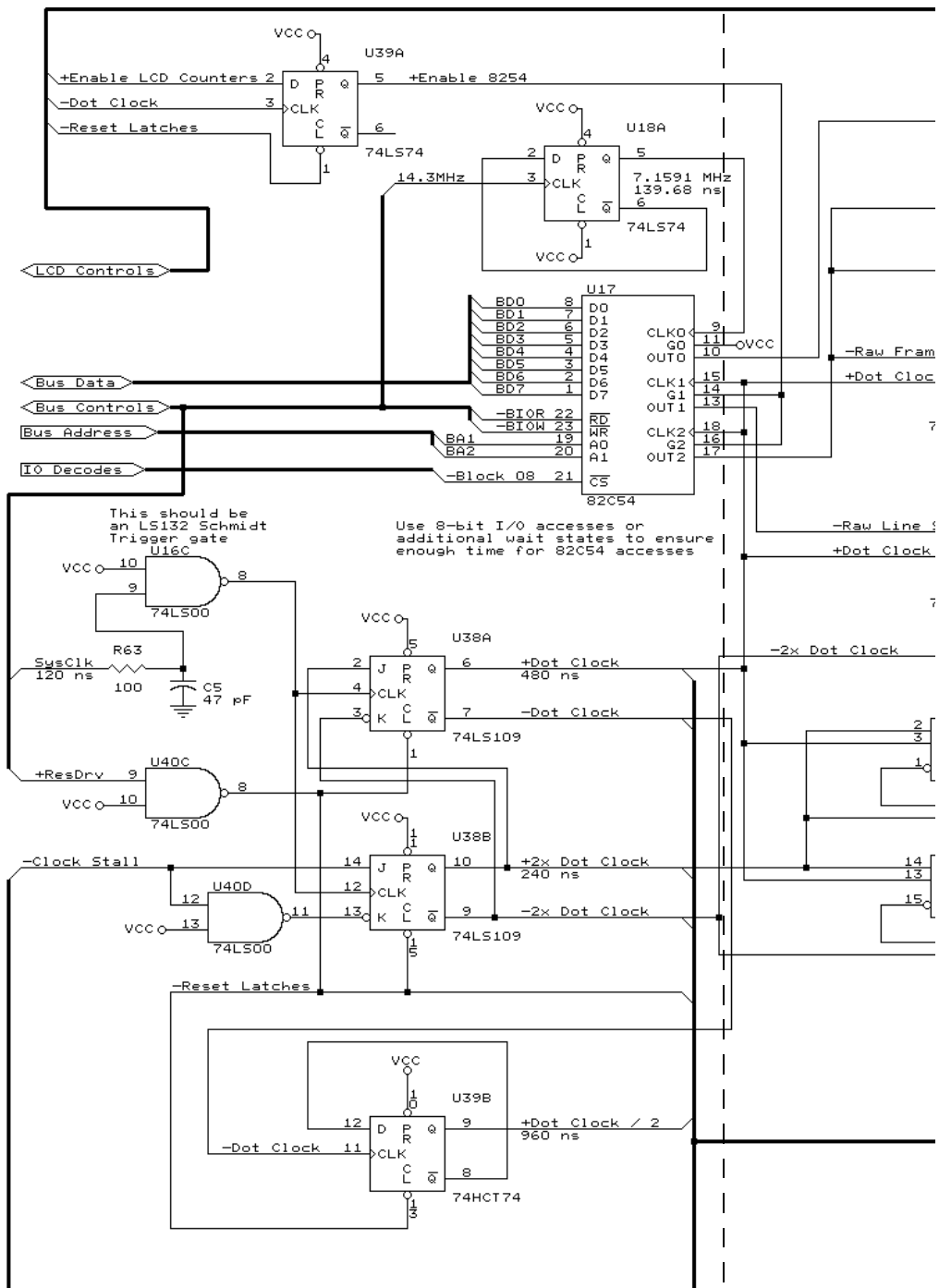
The MAX691 monitors the +5 V supply and triggers an NMI through the -IOCHCK bus line. Its watchdog timer output triggers a system reset using a wire to the Reset switch.

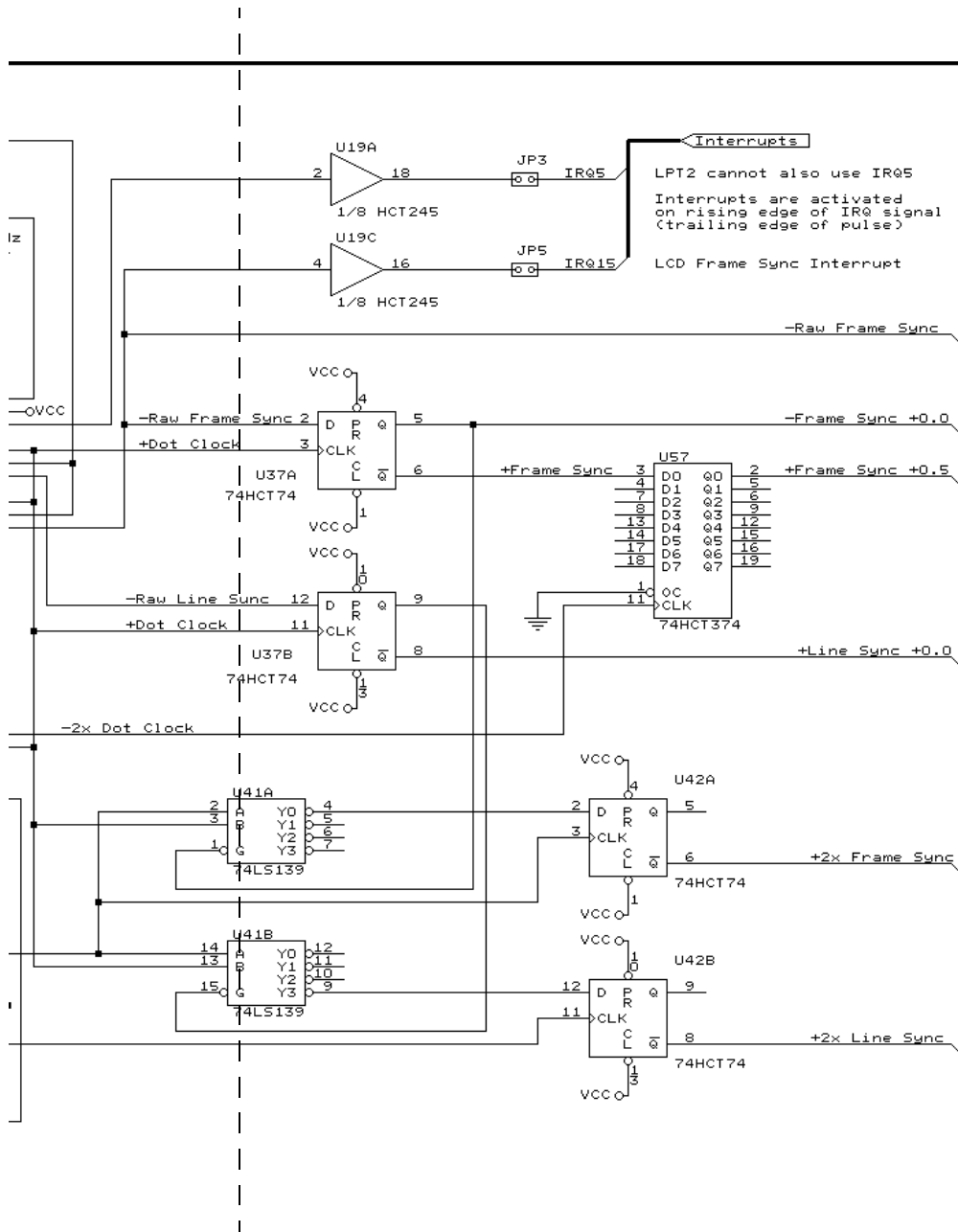




Schematic 8
I/O Devices

The Firmware Development Board includes three main I/O devices. Details of each gadget appear in successive pages.

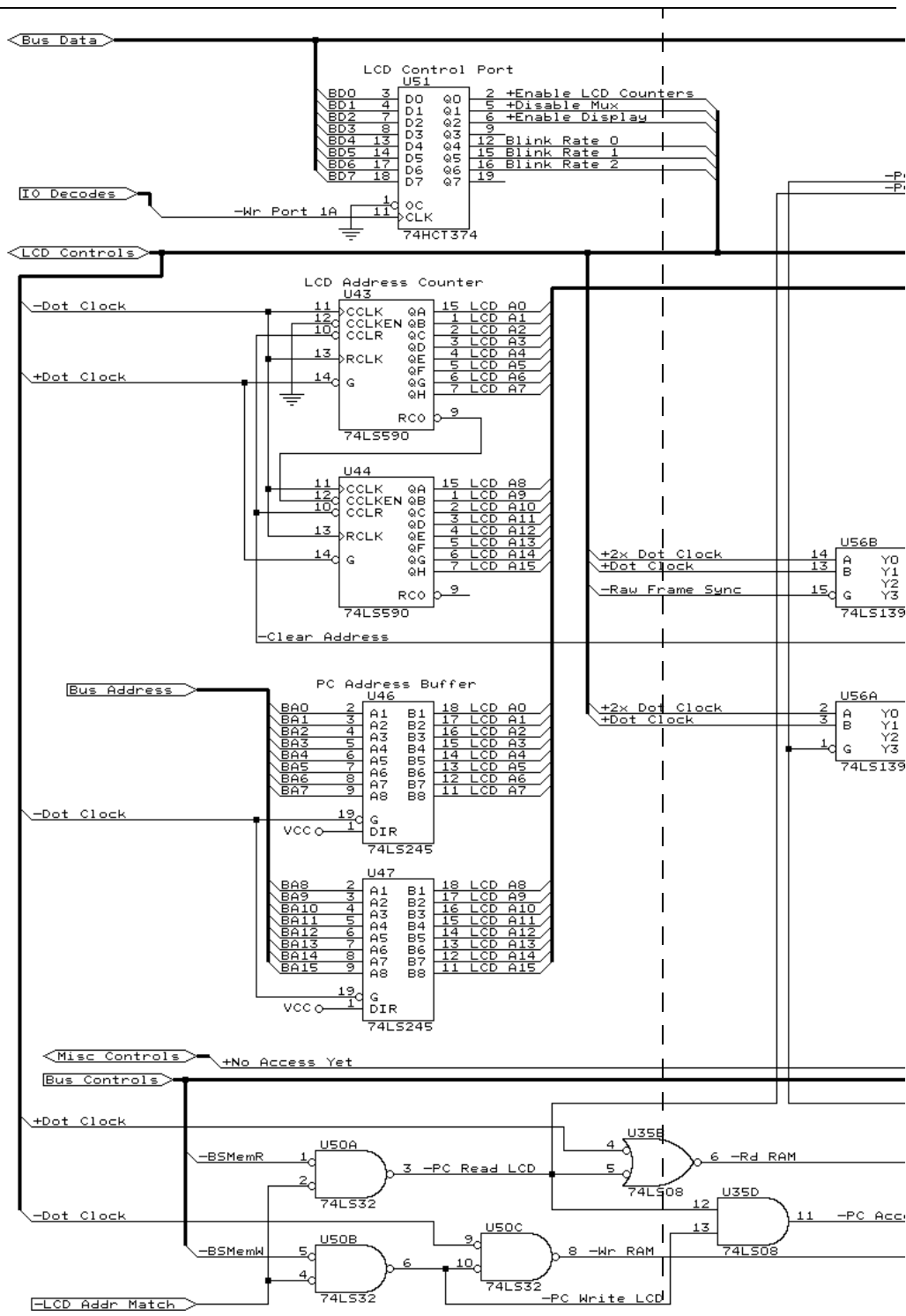


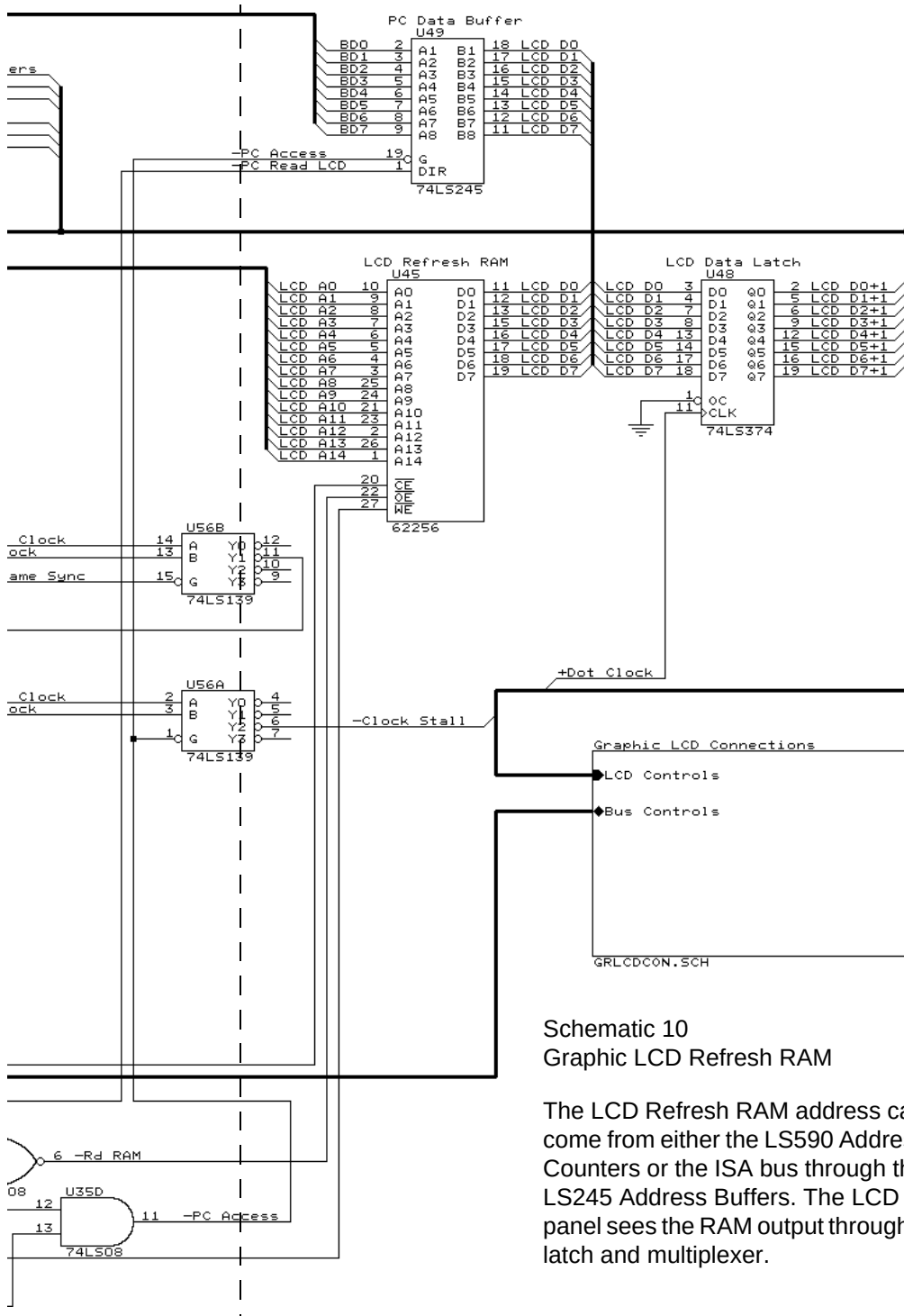


Schematic 9: Timer and LCD Clocking

The 82C54 timer can supply generally useful interrupts or control the Graphic LCD interface, the latter of which accounts for most of the support circuitry in this schematic.

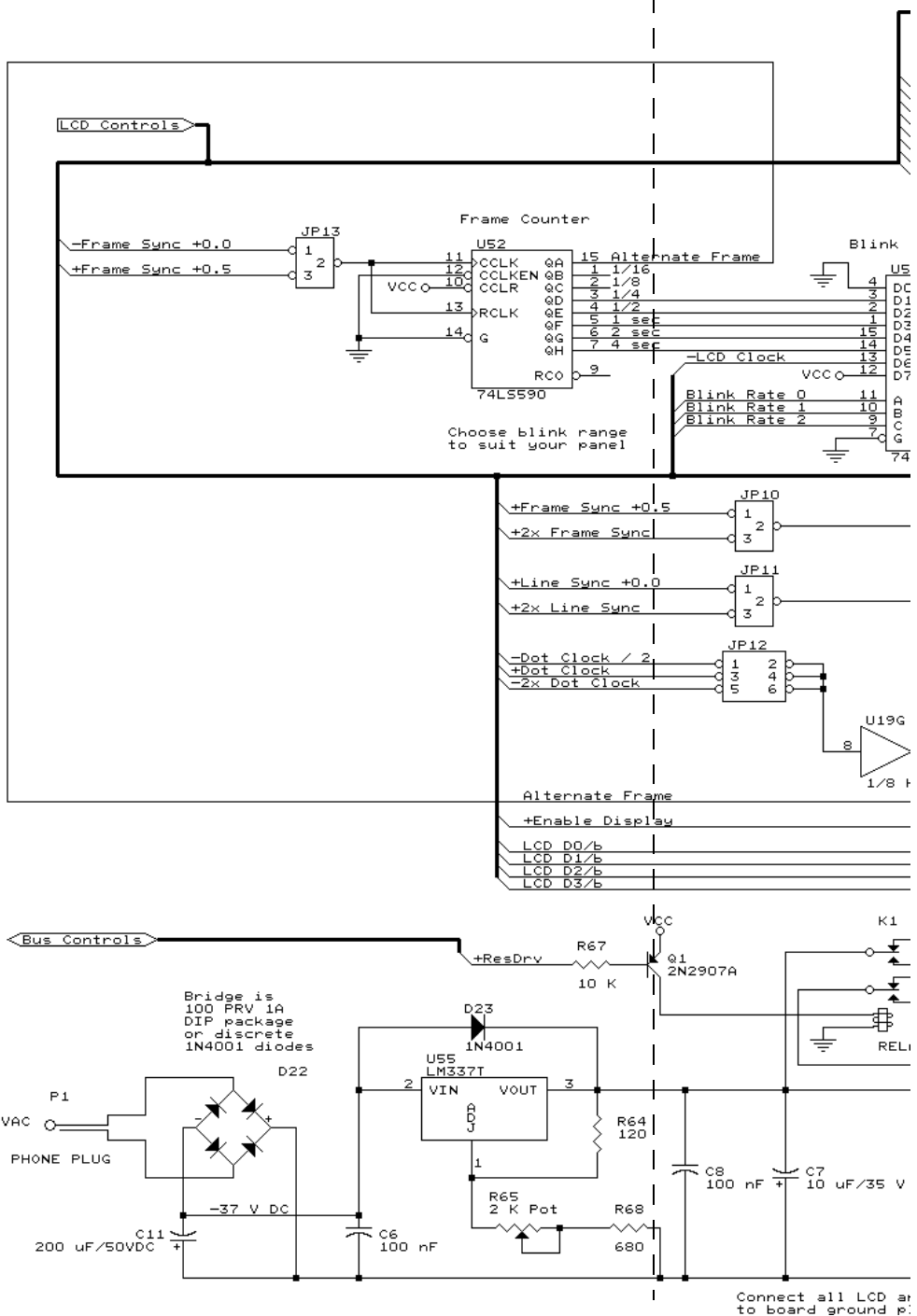
314

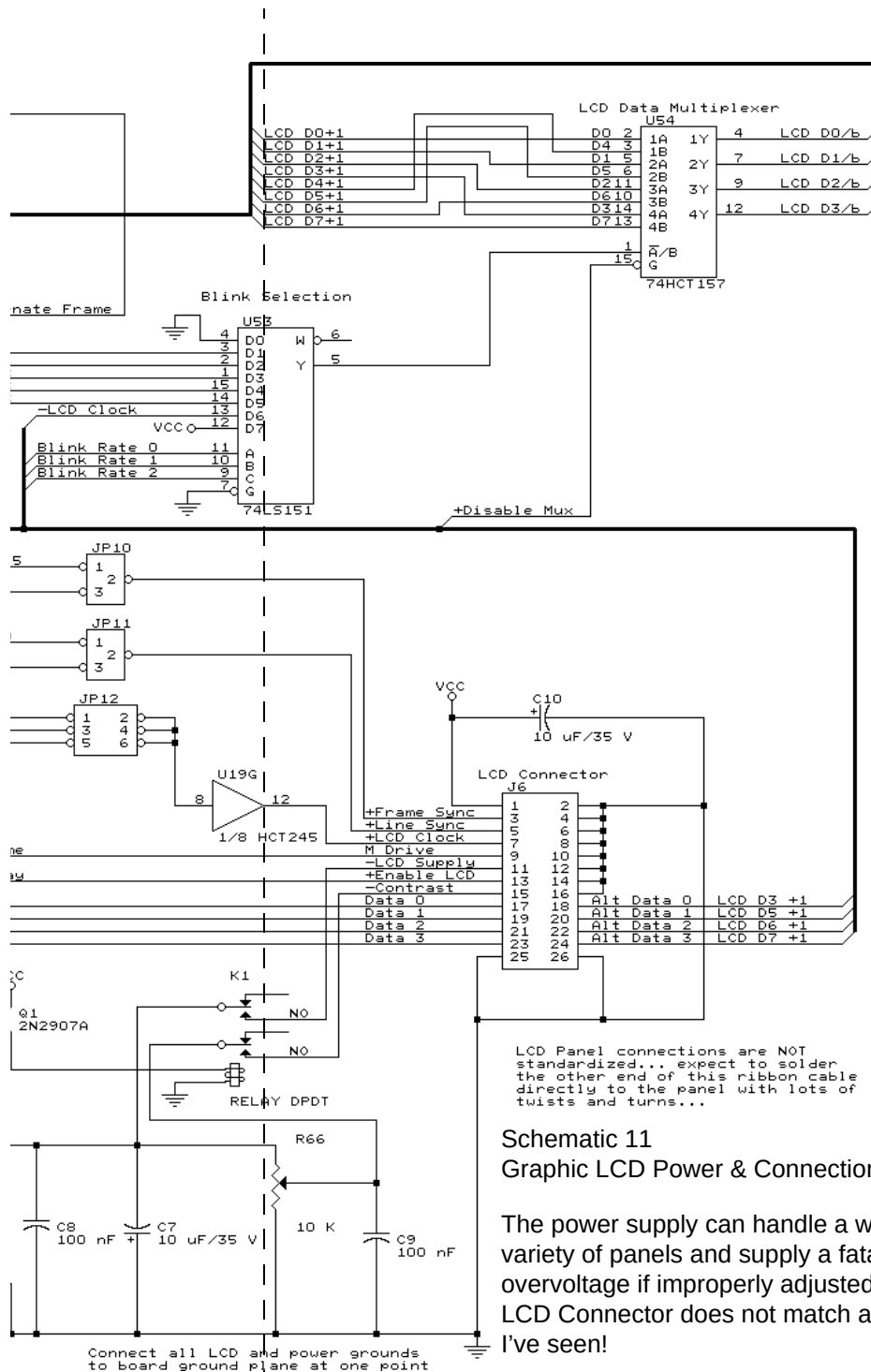




Schematic 10
Graphic LCD Refresh RAM

The LCD Refresh RAM address can come from either the LS590 Address Counters or the ISA bus through the LS245 Address Buffers. The LCD panel sees the RAM output through a latch and multiplexer.



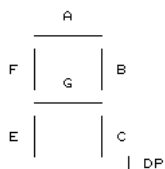


Schematic 11
Graphic LCD Power & Connections

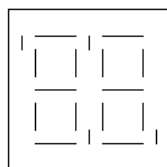
The power supply can handle a wide variety of panels and supply a fatal overvoltage if improperly adjusted. The LCD Connector does not match any panel I've seen!

LED byte values...

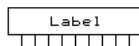
Char	Before NOT (inv)	After NOT (direct)
0	7E	81
1	30	CF
2	6D	92
3	79	86
4	33	CC
5	5B	A4
6	5F	A0
7	70	8F
8	7F	80
9	73	8C
A	77	88
B	1F	E0
C	4E	B1
D	3D	C2
E	4F	B0
F	47	B8
	05	FA
	0F	F0
	67	98
	08	F7



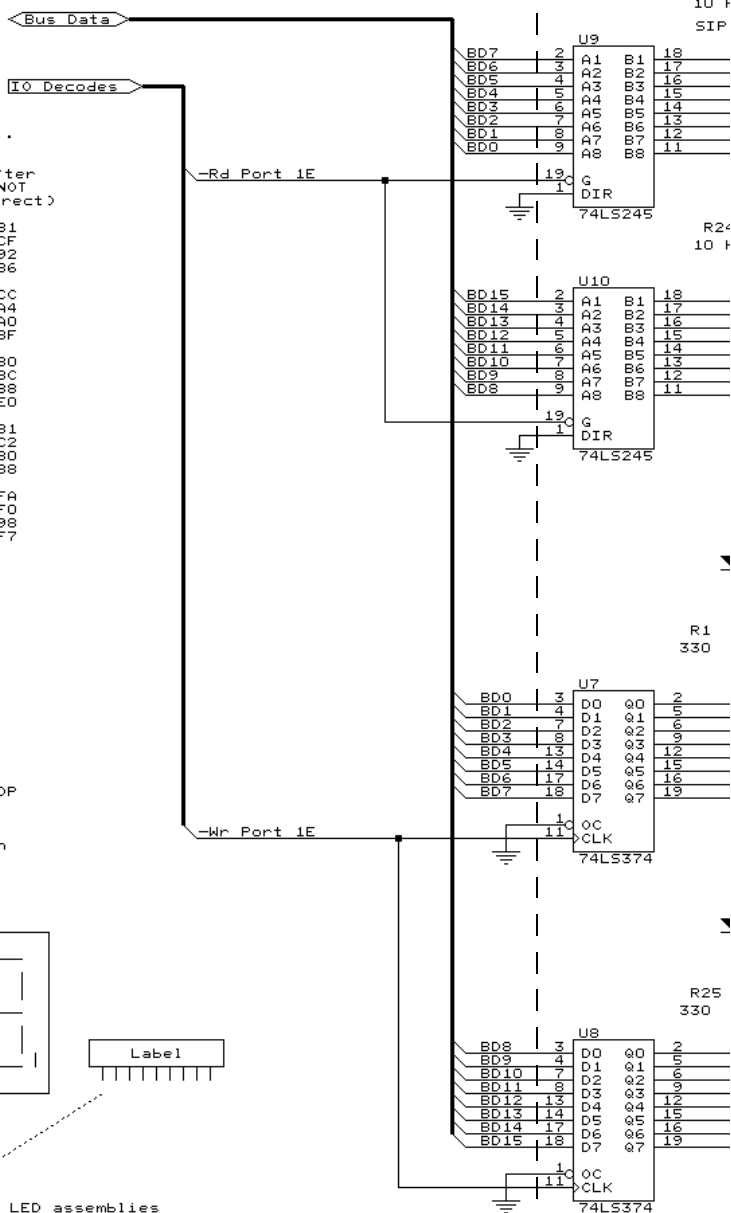
Segment Location

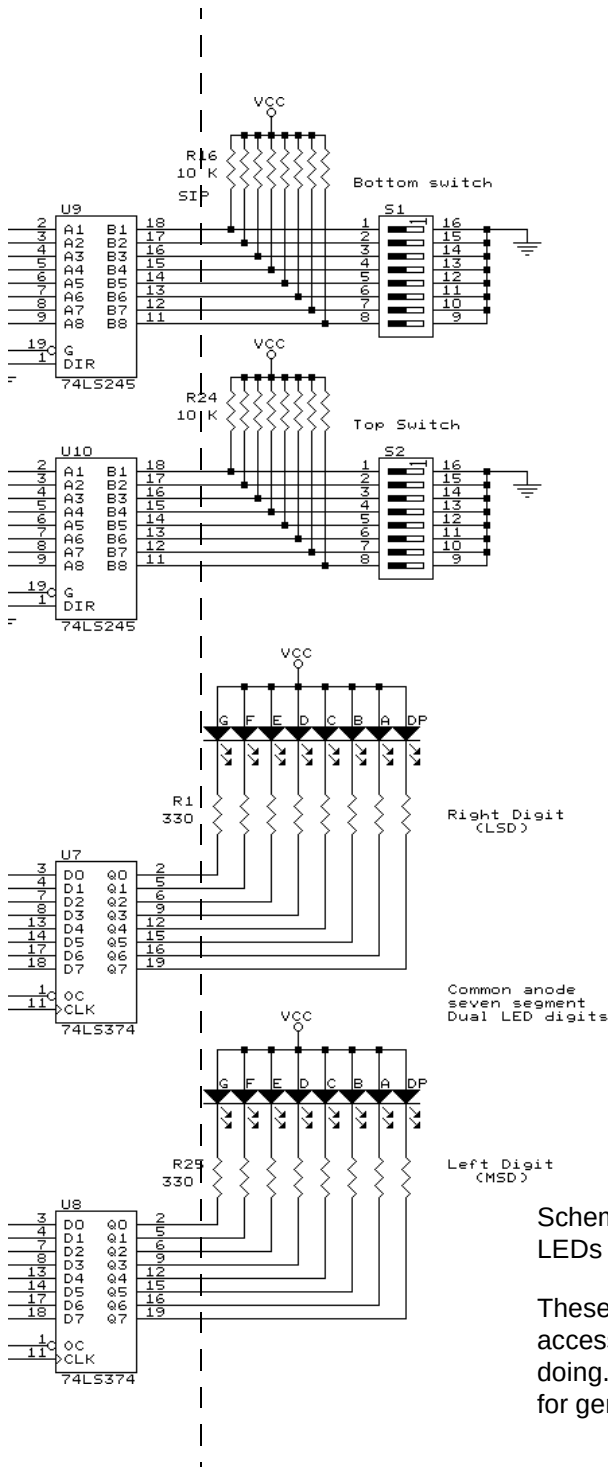


Label text



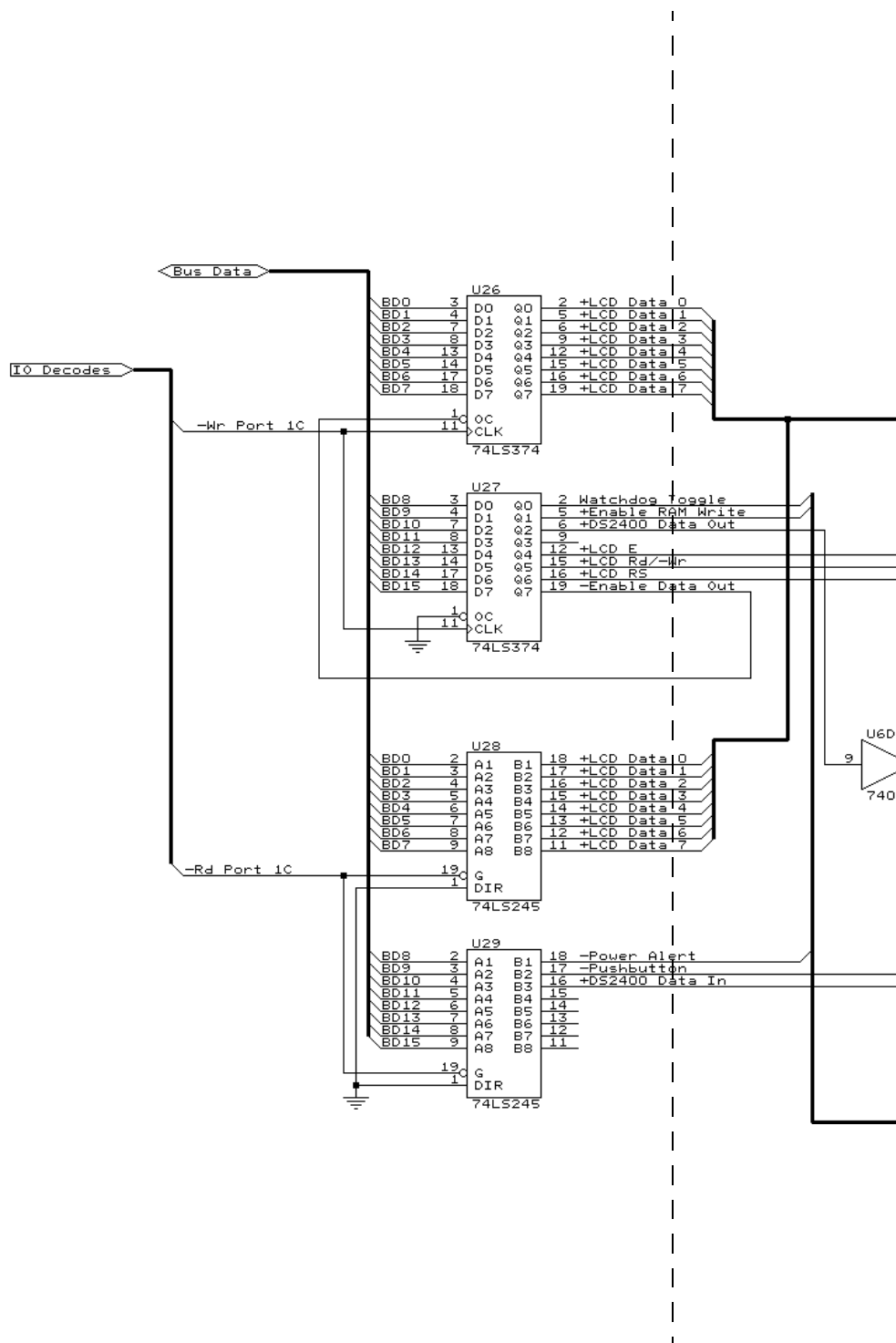
Most two-digit LED assemblies have pin 1 located here... but a little probing will make sure if yours is unmarked





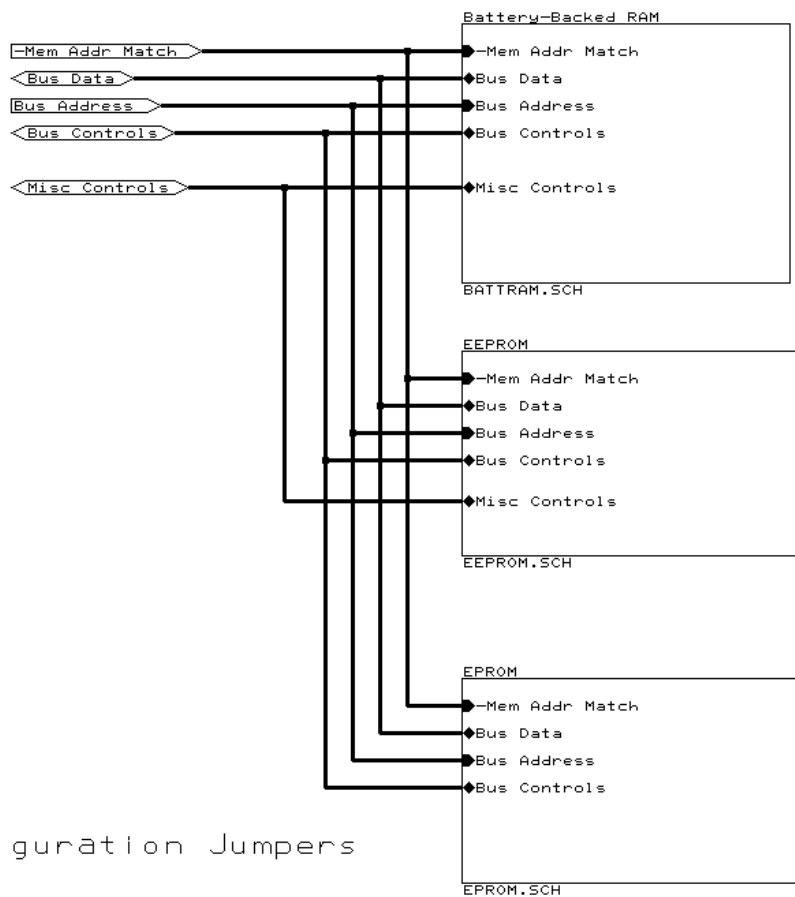
Schematic 12
LEDs and DIP Switches

These simple circuits provide convenient access to the firmware and show what it's doing. You can easily modify this circuitry for general bit input and output.

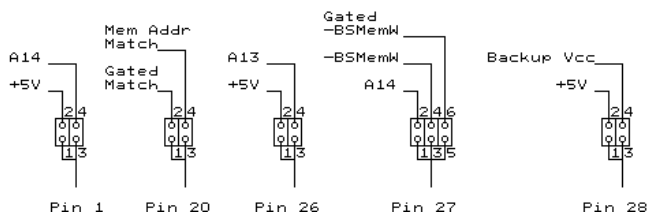


A small text LCD panel can be a useful output device for many systems. This I/O port design eliminates the need for a direct ISA bus connection with all its timing complexities.

Choose =one= memory circuit!



Configuration Jumpers



Socket Configuration

32K Static RAM (62256)

Pin 1 - BA14
Pin 20 - (Power OK) AND (Mem Addr Match)
Pin 26 - BA13
Pin 27 - (Enable RAM Write) AND (BSMemW)
Pin 28 - Backup Vcc

8K EEPROM (28C64A)

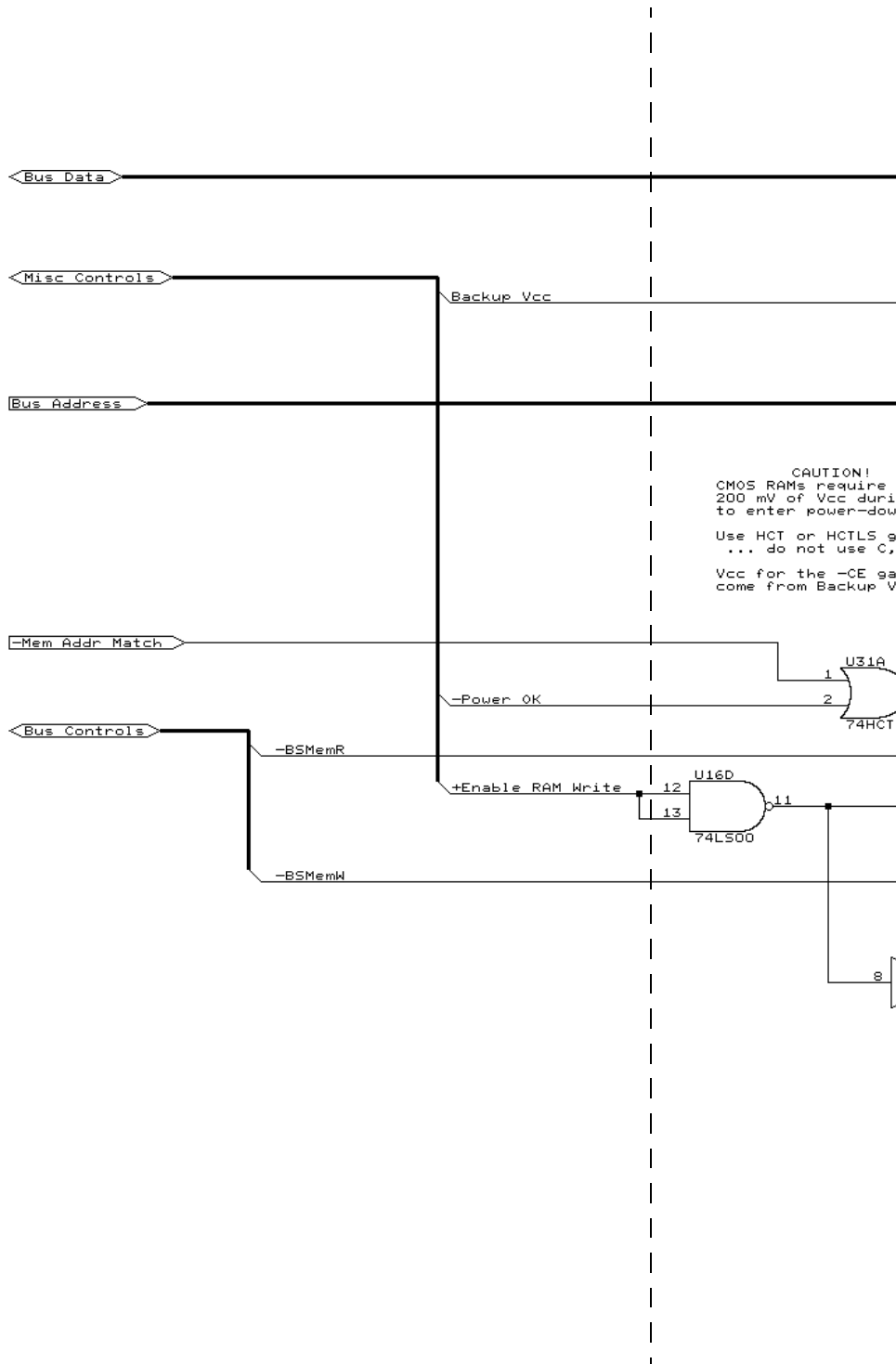
Pin 1 - no connection (-Busy out)
Pin 20 - Mem Addr Match
Pin 26 - no connection (tie high for 8K RAM)
Pin 27 - (Enable RAM Write) AND (BSMemW)
Pin 28 - +5 V

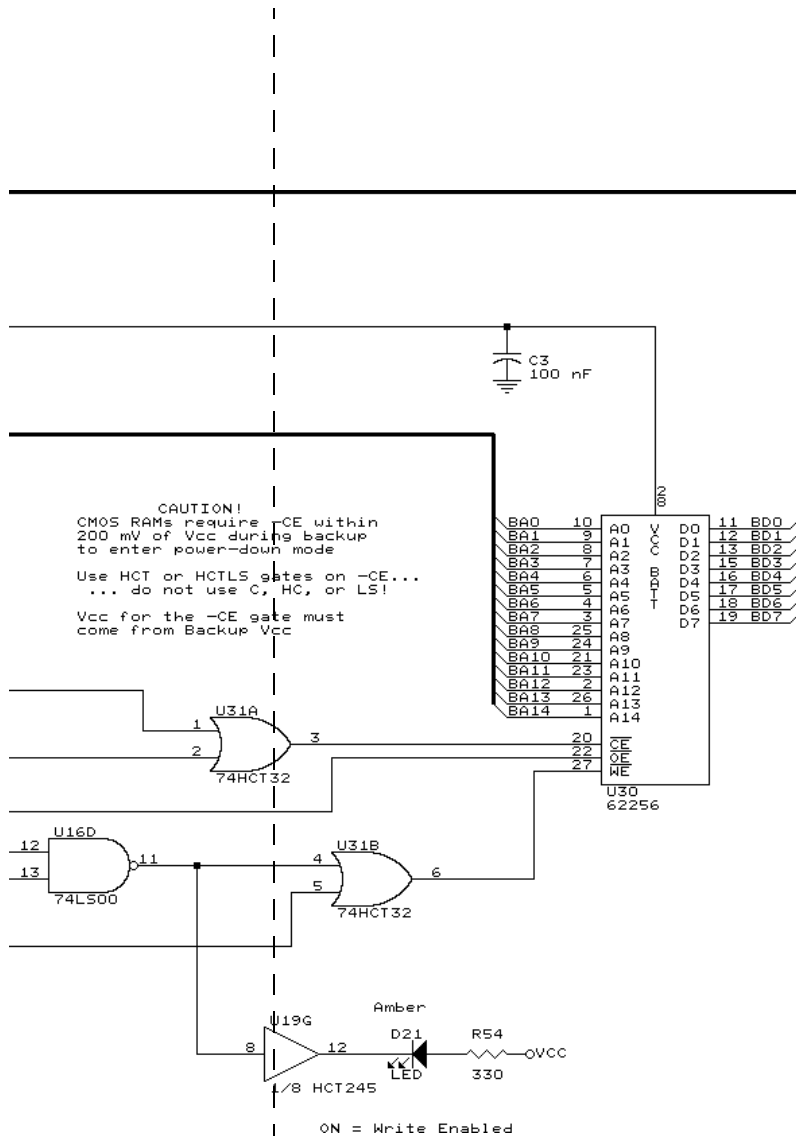
32K EPROM (27256)

Pin 1 - +5 V
Pin 20 - Mem Addr Match
Pin 26 - BA13
Pin 27 - BA14
Pin 28 - +5 V

Schematic 14 Memory Options

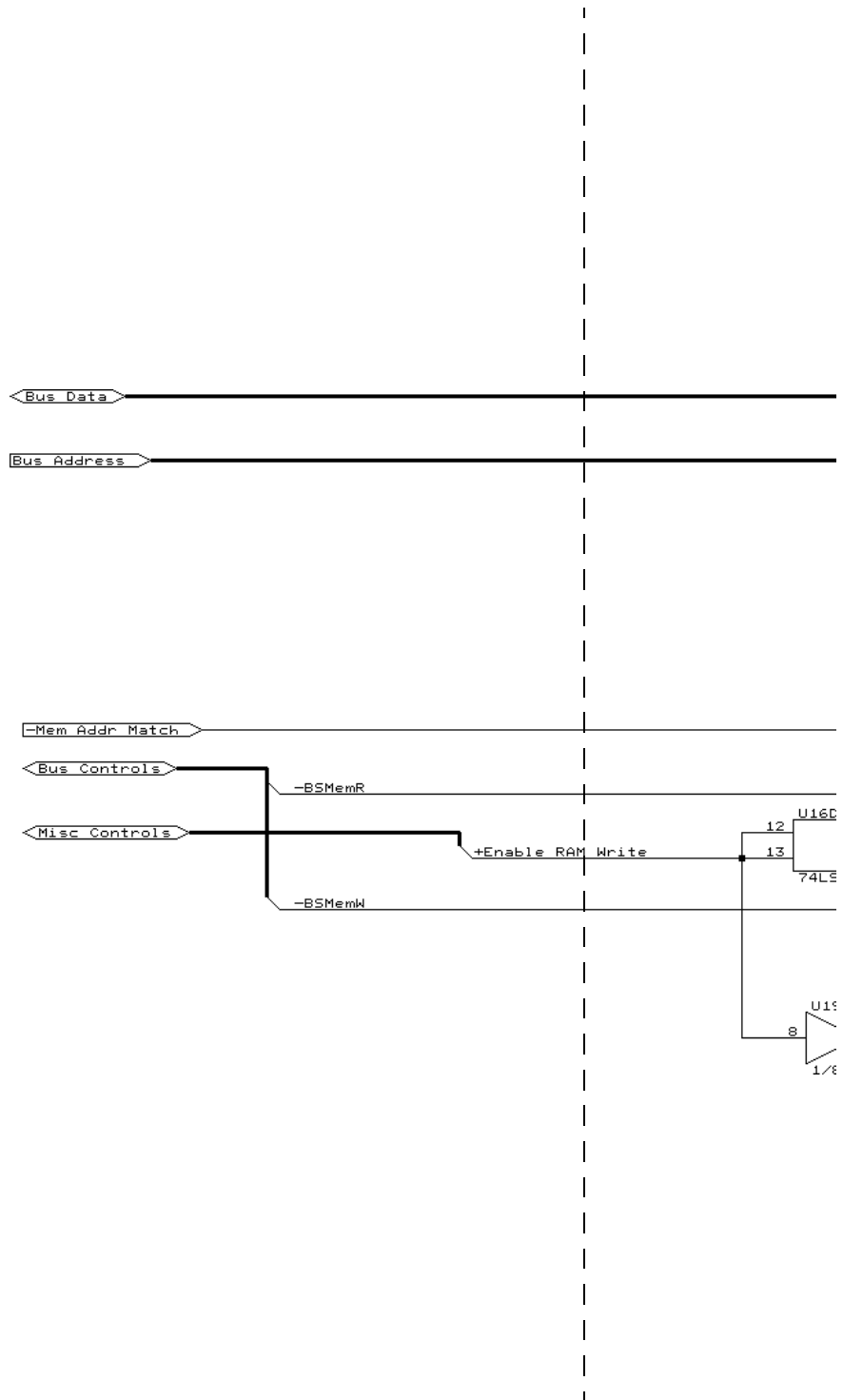
The Firmware Development Board can support nonvolatile memory with EPROM, EEPROM, or battery backed RAM. This schematic shows the various connections and jumper options.

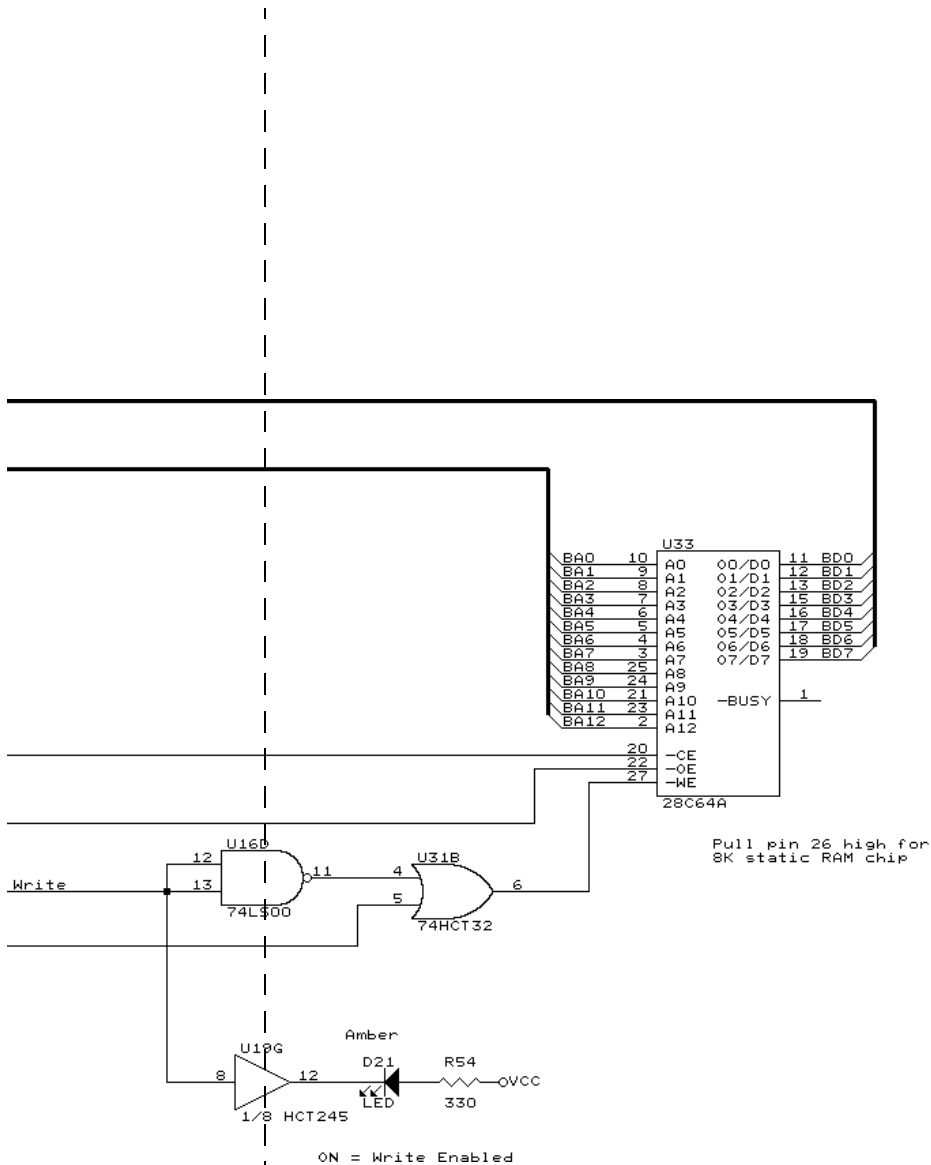




Schematic 15
Battery backed RAM

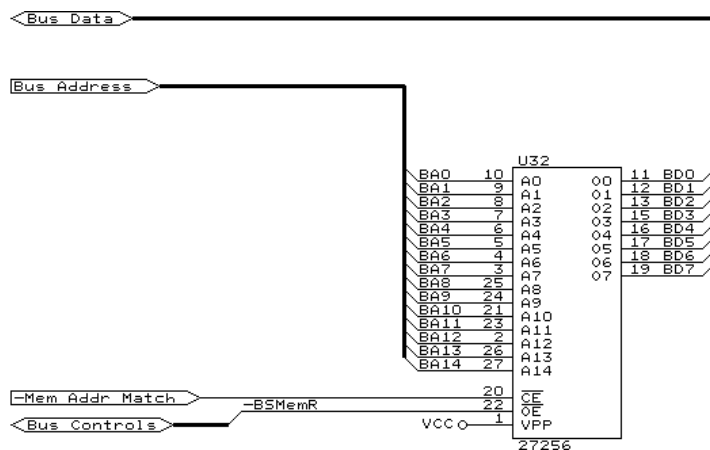
The MAX691 you saw earlier controls the Backup V_{CC} that drives this RAM and HCT32 gate. The RAM is normally write protected to prevent accidental data changes.





Schematic 16
EEPROM

Firmware polling determines the write cycle duration for this EEPROM. This chip must also have write protection to prevent accidental changes.



Schematic 17
EPROM

This is about as simple as a memory can get, even if the address and data buffers did appear on previous pages. You obviously don't need any write protection for an EPROM!

Port Addresses

You will find all of the Firmware Development Board I/O port addresses and bits defined in `\ISAbus\Code\Micro-C\FirmDev.h`.

U14, the 74F521 address comparator, decodes all I/O addresses from 0300 through 031F, even if the board doesn't actually have any hardware at a particular address. The Firmware Development Board hardware uses these I/O addresses.

Port	Read	Write
0308	Timer 0 Count/Status	Timer 0 Count
030A	Timer 1 Count/Status	Timer 1 Count
030C	Timer 2 Count/Status	Timer 2 Count
030E	unused	Timer Control Register
031A	unused	Graphic LCD Controls
031C	Board Status Bits	Board + Text LCD Controls
031E	DIP Switches	LED Digits (7 seg + dec pt)

See Schematic 12 for the bit layout in the two seven-segment LED digits at 031E.

Control and Status Bits

These bits at I/O address 031A control the Graphic LCD Interface. Because the interface is write only, there are no corresponding Status bits:

Bit	Write Function
7	unused
6 - 4	Blinking MUX selection
3	unused
2	Enable LCD Panel
1	Disable Blinking MUX
0	Enable Address Counters

These bits at I/O address 031C control various functions and report status conditions throughout the board:

Bit	Read Status	Write Function
15	unused	-Enable Data to Char LCD
14	unused	+RS to Character LCD
13	unused	+Rd/-Wr to Character LCD
12	unused	+E to Character LCD
11	unused	unused
10	+DS2400 Data In	+DS2400 Data Out
9	-Pushbutton	+Enable FDB RAM Write
8	-Power Alert from MAX691	Watchdog Toggle to MAX691
7 - 0	Data from Character LCD	Data to Character LCD

Memory Addresses

The Firmware Development Board occupies two distinct blocks of the CPU's memory address space, as defined by the jumper settings for U22 and U36. The settings used throughout the book are defined in `\ISAbus\Code\Micro-C\FirmDev.h`:

Address	Function
C800:0000 - C800:7FFF	(E)EPROM or RAM
D000:0000 - D000:7FFF	Graphic LCD Refresh RAM

Bill of Materials

This list includes the parts required for the Firmware Development Board, with the exception of the prototyping board itself and sundries such as sockets, jumpers, bypass caps, and so forth, all of which depend on how you plan to construct the hardware.

Remember those bypass capacitors! Although the ISA bus circuitry doesn't run at warp speed, you must still pay attention to your construction techniques. A prototype board with solid power planes will work much better than daisy chained Wire-Wrap links.

The EEPROM, EPROM, and battery backed RAM share common circuits, so you won't need quite as many parts as you see on those pages. Decide which versions you'll build and plan accordingly.

Most of the LEDs appear in the two seven-segment digits in Schematic 12, which doesn't show their reference numbers. The remainder, all discrete LEDs, are scattered over the board to indicate key logic states.

The Embedded PC's ISA Bus

Item	Quan	Ref	Description	Item	Quan	Ref	Description
1	1	BT1	3V Lithium			R3	330
2	2	C1	47 pF			R4	330
		C5	47 pF			R5	330
3	1	C2	1 nF			R6	330
4	4	C3	100 nF			R7	330
		C6	100 nF			R8	330
		C8	100 nF			R9	330
		C9	100 nF			R10	330
5	1	C4	1-10 nF			R11	330
6	2	C7	10 uF/35 V			R12	330
		C10	10 uF/35 V			R13	330
7	1	C11	200 uF/50VDC			R14	330
8	21	D1	LED			R15	330
		D2	LED			R17	330
		D3	LED			R18	330
		D4	LED			R19	330
		D5	LED			R20	330
		D6	LED			R21	330
		D7	LED			R22	330
		D8	LED			R23	330
		D9	LED			R25	330
		D10	LED			R26	330
		D11	LED			R27	330
		D12	LED			R28	330
		D13	LED			R29	330
		D14	LED			R30	330
		D15	LED			R31	330
		D16	LED			R32	330
		D18	LED			R47	330
		D19	LED			R48	330
		D20	LED			R49	330
		D21	LED			R54	330
9	1	D17	1N4148	28	7	R16	10 K
10	1	D22	BRIDGE			R24	10 K
11	1	D23	1N4001			R38	10 K
12	4	JP_1	JMP 2X2			R53	10 K
		JP_20	JMP 2X2			R55	10 K
		JP_26	JMP 2X2			R66	10 K
		JP_28	JMP 2X2			R67	10 K
13	1	JP_27	JMP 2X3	29	6	R33	4.7 K
14	3	JP1	HEADER 5X2			R34	4.7 K
		JP6	HEADER 5X2			R35	4.7 K
		JP9	HEADER 5X2			R36	4.7 K
15	5	JP2	JMP			R37	4.7 K
		JP3	JMP			R56	4.7 K
		JP5	JMP	30	1	R39	25 K
		JP7	JMP	31	1	R40	2.7 K
		JP8	JMP	32	10	R41	R
16	3	JP10	3 HEADER			R42	R
		JP11	3 HEADER			R43	R
		JP13	3 HEADER			R44	R
17	1	JP12	HEADER 3X2			R45	R
18	1	J1	CON AT62B			R58	R
19	1	J2	CON AT36B			R59	R
20	2	J3	CON2			R60	R
		J4	CON2			R61	R
21	1	J5	CON14AP			R62	R
22	1	J6	CON26A	33	1	R46	560
23	1	K1	RELAY DPDT	34	2	R50	2.5 K
24	1	LCD1	DMC-20434			R57	2.5 K
25	1	P1	PHONE PLUG	35	1	R51	45 K
26	1	Q1	2N2907A	36	1	R52	15 K
27	35	R1	330	37	1	R63	100
		R2	330	38	1	R64	120

Item	Quan	Ref	Description
39	1	R65	2 K Pot
40	1	R68	680
41	2	S1	SW DIP-8
		S2	SW DIP-8
42	1	S3	SW PUSHBUTTON
43	14	U1	74LS245
		U2	74LS245
		U3	74LS245
		U4	74LS245
		U5	74LS245
		U9	74LS245
		U10	74LS245
		U20	74LS245
		U21	74LS245
		U28	74LS245
		U29	74LS245
		U46	74LS245
		U47	74LS245
		U49	74LS245
44	1	U6	7407
45	5	U7	74LS374
		U8	74LS374
		U26	74LS374
		U27	74LS374
		U48	74LS374
46	3	U11	74LS139
		U41	74LS139
		U56	74LS139
47	2	U12	74LS138
		U13	74LS138
48	3	U14	74F521
		U22	74F521
		U36	74F521
49	1	U15	74LS221
50	2	U16	74LS00
		U40	74LS00
51	1	U17	82C54
52	2	U18	74LS74
		U39	74LS74
53	1	U19	1/8 HCT245
54	1	U24	MAX691
55	2	U30	62256
		U45	62256
56	1	U31	74HCT32
57	1	U32	27256
58	1	U33	28C64A
59	1	U34	DS2400
60	1	U35	74LS08
61	3	U37	74HCT74
		U39	74HCT74
		U42	74HCT74
62	1	U38	74LS109
63	3	U43	74LS590
		U44	74LS590
		U52	74LS590
64	1	U50	74LS32
65	2	U57	74HCT374
		U51	74HCT374
66	1	U53	74LS151
67	1	U54	74HCT157
68	1	U55	LM337T

Index

Symbols

?heap 172
?temp 82, 171

Numerics

16450 91
16550 91
27256 95, 100
28C256 104
28C64A 100, 103–104
2x Dot Clock 211, 221, 224, 227, 238
6800 150
74xx logic, see also LSxx, Fxx, HCxx ... 43
7407 43, 149, 151
74F521 44, 329
74HCT32 105, 121
74LS688 44
80286 26, 56, 138, 142–143, 190, 251, 253–
254, 277
8031 89, 93, 122, 179, 196
80386 26, 36, 138
80386SX 10, 19, 35, 50, 90, 214, 244, 252–
253, 255
8042 142
80486 26, 79
8051 62, 151, 153, 157
8052-BASIC 90
8080 135
80826 142
8086 26, 34, 73, 79, 155, 191
8086RLPS.ASM 178
8086RLPT.ASM 26
8086RLXS.ASM 178
8088 38, 43, 54–55, 79, 93, 138, 167, 191,
251, 253
8250 90–92
8253 55
8254, see 82C54 125
8259 74, 76, 130
control lines 77

INT output 76
IRQ inputs 76
master 76, 79, 81
slave 76, 78
SP/–EN pin 78
8284 54–55
82C54 53–54, 56–57, 59, 62, 76, 147, 153,
155–156, 214, 216–217, 229–230,
233, 235, 239
Clock 62, 214, 216
Clock glitches 216
Control Word 64
Gate 62, 214, 233
Mode 2 81
Mode 3 64
modes 62, 64
Output 62, 68
readback 85
setup 64
Status Byte 67–68
8514/A 205

A

abort 73
address bits 44, 100, 252
above 1 MB 278
address bus 57
address decoding 43, 57, 167
address space 187, 205
Alternate Frame 199, 221, 227
ANSI 269
cursor controls 161, 193, 268–269
decoding 269
ASCII 261
character set 265
character zero 267
control characters 265
assembly
386 instructions 139
conditional 141

assumption, unjustified 89, 158

B

Back Space 265

backlight inverter 231

Bad Idea 139

Bad News 93, 259

Bad Thing 224

BALE
(Bus Address Latch Enable) ... 37, 48

bare metal 26

barnacles
see also PC Compatibility Barnacles 35

battery-backed RAM .. 180, 190–192, 205
see also MAX691 167

BCLK (Bus Clock) 37, 39, 48

Big Red Switch 120

BIOR
(Buffered I/O Read) 43–44

BIOS
bitmap fonts 259
character-drawing 258
font 265
power-on memory check 105
power-on reset timing 124
RAM size 170
redirecting 177
timer 48, 138
timer tick 126

BIOS alarm clock 68

BIOS extension 190, 192
addressing 110
booting 164
checksum 107
entry conventions 167
identification 106
length code 106
memory models 164
Original PC 133
scan 134
signature 106
stack use 171
system initialization 174

system state 163

BIOS kits 26

BIOS services 53

BIOS time-of-day 53

BIOSExt.LIB 178

BIOW (Buffered I/O Write) 44

blown stack, see stack, blown 71

boot loader 27, 167
BootSect.ASM 15, 186

booting from diskette 110

Borland
C compiler 34, 179
C startup code 183

BOUND 72

BP register 81, 261

Bradley, David 79

buffers 41

bus
address bits 43
buffers 41
cycle 48, 205
loading 41
BusLabel file 51

C

C strings 267

Carriage Return 265

Carriage Return character 265

Cassette BASIC 111, 164, 177

CAUTION 3–4

CC86 178

CE 113–114

CE2 104

CGA 54, 93, 170, 257, 259, 261

Clear Counter 217

clothes dryer 71

CMOS 207, 221
TTL interface 149

CMOS configuration 11

COBOL 53, 131

code segment 166
see also CS 27

-
- COM file 27, 135–136, 180
 conventions 185
 segments 180
 COM port 22, 33, 189, 192
 COM program 165
 see also COM file 165
 segment registers 137
 Console.C 256
 constants 167
 Consulting Engineer's First Principle .. 143
 CP/M 135
 CPU
 performance 50
 sufficiently fast 91
 CRC in DS2400 157–158
 Creeping Featuritis 179, 188–189
 critical regions 162
 CS 137–138, 170, 186
 CS (Chip Select) 59
 CS:IP 26, 137, 143, 170, 185–186
 cuts and adds 45
- D**
- DAck 279
 data segment
 reserving RAM for 170
 see also DS 27
 DBR (Diskette Boot Record) 12
 Debug 21
 Debug (DOS program) 15
 debugger 189
 debugging, burn-and-crash 188
 delay line 51
 delay loops
 see also timing loops 160
 Delete 265, 267
 DIP switches ... 39, 45, 57, 124, 128, 256
 diskette boot loader 136, 167, 185, 187
 diskette controller 278
 DISKSIZE macro 12
 DMA (Direct Memory Access) .. 278–279
- DMF651 195–196, 199, 204, 220, 231–232,
 234–235, 238, 250, 262–263, 265,
 272, 275
 DogTest 126, 128, 130
 DOS environment 183
 Dot Clock 202, 204–209, 211, 214, 216–217,
 220–221, 224–225, 227, 230–232,
 235, 237–239, 263
 pausing 224
 Dot Clock/2 211, 227, 239
 Dot Clocks 209, 214, 220
 Double Fault 191
 DRQ (DMA ReQuest) 279
 DS 137–138, 172, 182
 DS2400 147, 149–151, 153, 156, 158
 command word 156
 CRC 151, 157
 data 158
 interface 153
 laser trimming 150
 mask error 158
 power 151
 reset glitch 151
 Type ID 157
 dual-ported RAM
 see also LCD Refresh RAM 204
- E**
- EEPROM
 Busy output 104
 polling 103
 programming 104
 RAM use 110
 read access 100
 schematic 100
 slaughter 104
 valid data 104
 write access 100
 write protection 104
 EGA 261
 EISA 35
 EMM386 205, 252
-

EMS	
see also expanded memory	205
see also LIM EMS	205
Enable LCD Counters	214
End of Medium	265
end-of-line wrapping	267
ENDXFR	
see also Synchronous Ready	39
EOI (End Of Interrupt)	78, 82
for NMI	130
nonspecific	78
EPROM schematic	95
Epson EG7004	209
ES	137
ES:BP	261
ES:DI	247
ES:SI	254
escape hatch	134
Exceedingly Bad Idea	10, 129, 277
exception interrupts	72–73
exceptions	
fix up	73
EXE file	180
header	180
loader operations	182
Relocation Table	181
expanded memory	94, 189, 205, 252
extended memory	189, 252, 256
external interrupts	72, 191
Extra credit	161, 265
ExtTest	141, 145

F

F521	47–48, 95, 100
FARHEAP	256
fault	73
FDBExt	145
FDBExt.ASM	134
FIFO	278–279
FirmDev	92, 111
First Program	16, 21
FLAGS (register)	72–73
flat memory model	182

font bitmaps	258
Form Feed character	265
Frame Sync	197–199, 204, 214, 216–217, 220, 227, 230, 234, 245
Frame Sync +0.0	221
Frame Sync +0.5	220–221, 227
FS segment register	138

G

GAL	41
Game of Life	243, 252
dot mapping	245
Optimization Challenge	245
rules	244
slide show	252
GDT (Global Descriptor Table)	254
getch()	27
Glass Teletype	257, 265, 269
glitch	160
Good Thing	56, 179
GraphLCD	229, 238, 242
ground plane	45
GS (segment register)	138

H

H61104	199
H61105	199
hardware interrupt handler	
see also interrupt handler	155
hardware reset	26, 191
HCT74	211
HD44780	150, 159
HDM86	22
heap	165–166, 183
heartbeat	126
Hello, world	16, 174
Hello.C	32
HEXFMT	25
HM62256LP	
see also battery-backed RAM	113
see also RAM	113
Hobson's Choice	79
Hopper, Adm. Grace	53

-
- hyperthyroid 143, 191
 - I**
 - I/O addresses 43
 - see also I/O port 43
 - I/O operations 57
 - I/O port 43, 129
 - IBM PC 93, 120, 164, 257–258
 - IBM PC AT 9, 39, 53, 56, 90, 133, 142, 191, 254
 - IBM PC/XT 257
 - IDT (Interrupt Descriptor Table) 73
 - IEEE P996 121
 - IF (Interrupt Flag) 72
 - IIR 91–92
 - interrupt summary bit 91
 - pending interrupt number 91
 - IIR (Interrupt ID Register) 92
 - impossible, doing the 153
 - IMR (Interrupt Mask Register) 76
 - Industry Standard Architecture 35
 - inline assembly code 81
 - Insert 267
 - In-Service Register, see also ISR 76
 - Instruction Pointer
 - see also IP 23
 - instruction queue 50
 - instruction timing 153
 - INT 00h 73
 - INT 02h 72
 - INT 08h 73, 79, 191
 - INT 0Bh 191
 - INT 0Fh 79
 - INT 10h 72, 177, 259, 261
 - INT 15h 253–254, 256
 - Int 15h 68
 - INT 18h 110–111, 164, 176–177
 - INT 19h 110, 163–164, 173–174, 176–177, 189–190
 - INT 1Ch 126
 - INT 1Fh 259
 - INT 21h 26, 183
 - INT 50h 81
 - INT 60h 171
 - INT 61h 173
 - INT 67h 171
 - INT 70h 84
 - INT pin 78
 - int86x() 261
 - INTA (INTerrupt Acknowledge) 71, 76–77
 - Interrupt
 - ID 72, 77–78
 - interrupt
 - conflicts 79
 - Double Fault 191
 - exception 79
 - hardware handler 82
 - hardware processing 78
 - nested 83, 175
 - number 72
 - pending 90
 - processing 73–74
 - reserved 79
 - Segment Not Present 191
 - serial 90
 - software 72
 - interrupt handler 72, 82, 129, 162, 175
 - chaining 173
 - hardware 176
 - IRET 175
 - latency 87
 - macro wrapper 175
 - reading port bits 162
 - return 74
 - return from 73
 - software 177
 - stack 176
 - INTERRUPT macro 81
 - interrupt priority rules 76
 - Interrupt Service Routine, see also ISR 72, 76
 - interrupt type 72
 - interrupt vector 73, 137
 - hooking 163
 - user functions 171
 - Interrupt Vector Table 73
-

INTERRUPT, C macro 82
INTO 72
INTR (Interrupt Request) .. 71–72, 76–78
intr() 261
IO ADDRESS MATCH 44
IO.SYS 13
IOCHCK (IO CHannel Check) 122, 129,
140
IOCHRDY 48
 (I/O CHannel ReaDY) . 43, 48, 50–51
IOCS16 (I/O Chip Select 16-bit) 43–44, 57
IOR (I/O Read) 38, 48, 51, 59, 61, 77
IOW (I/O Write) 37, 48, 51, 61, 77
IP 137
IP (Instruction Pointer) .23, 137, 170, 186,
188
IRET 74
IRQ
 see also interrupts 78
IRQ0 62, 76
IRQ15 76
IRQ7 76
IRQ8 53, 76
IRQ2 78, 81
IRQ5 81, 83, 87
IRQ8 84
IRQ10 81, 83
IRQ15 81, 83, 87
IRR (Interrupt Request Register) .. 76, 78
ISA bus memory access 95
ISR
 see also In-Service Register 78
 see also Interrupt Service Routine .. 92
ISR (Interrupt Service Routine) 76
IVB (Interrupt Vector Byte) 78–79
IVT, see also Interrupt Vector Table ... 73

J

jumper 44, 50

K

keyboard controller 191
keyboard lock switch 133, 170

kludge 92, 121, 143

L

laying-on of hands 243

LCD

backlight 209
blinking 204, 233, 235, 263
block diagram 202
busmastering 226
CAUTION 226
character cursor 265
connector 200
DC to DC converter 200
DMA interface 226
duty cycle 197
external controller 196
frame memory 196
Line Sync 196
logic levels 200
LSI driver chips 199
optical response 225
polarization 199
power supply 226
proportional characters 263
refresh rate 196, 205
resolution 205
row selection 197
signal names 200
sync 198
voltage 200
LCD A15 227
LCD Address Counter 217, 230
LCD Address Latch 217
LCD Control Latch 230
LCD Control Port 214, 232, 235, 237
LCD Data Latch 217, 220, 237
LCD Data Multiplexer 221, 227, 238, 252
LCD Refresh RAM 202, 204, 217, 220, 224,
230–232, 234, 238, 247, 249, 251,
256, 258, 261, 263–264, 275, 277,
280

-
- LCD, character 147, 149, 158, 173
 contrast 150
 Enable 159
 Enable glitch 160
 interface 159
 LCDMakeAddr 252
 LCDTest.C 161
 LED
 digits 39, 45, 57, 68, 124, 134, 141, 158,
 173, 184, 190, 192
 drivers 125
 indicators 124
 LG64AA44D 205–207, 237, 246, 250, 263–
 264, 267, 275
 LIDT 73
 LIM
 see also expanded memory Spec ... 94
 LIM EMS (Lotus-Intel-Microsoft) 94, 252
 Line Feed 265
 Line Sync ... 197–198, 214, 216, 220, 227
 Line Sync +0.0 220
 line wrapping 270
 lithium cell 114, 117, 149
 fault current 120
 recharging 117
 LM215 .209, 221, 227, 239–240, 250–251,
 272
 LM337 230
 LM64015T 209, 227, 238, 272
 LM641481 227, 238
 LoadExt 107, 111, 131, 145, 192
 Locate 180, 182, 185, 229, 256, 275
 CFG file 188
 output formats 185
 startup code 183
 statements 187
 logic analyzer 6, 37, 47–48, 85, 90
 logic probe 6
 LS00 216
 LS14 216
 LS74 56, 124
 LS109 211, 216
 LS132 216
 LS138 44
 LS139 44
 LS221 45, 48, 50–51
 LS245 41, 43, 48, 56, 59, 95, 125, 150, 224
 LS374 45, 48, 150
 LS541 41
 LS590 217

M
 magic dust 179
 main() 26, 166, 172, 174, 177, 182
 MakeFile 188, 256
 malloc() 183
 MASM 25
 Master (bus signal) 279
 Matsushita EDM LG64AA44D
 see also LG64AA44D 195
 MAX691 113, 117, 120–121, 123, 126, 129,
 140–141, 149
 Battery On 122
 bypass cap 120
 CE In 121
 CE Out 121
 delay time 121
 Low Line 123
 Osc In 124
 Osc Sel 124
 PFI 122, 139–140
 PFO 122–123, 130, 139–140
 Reset 122, 126
 schematic 117
 VOUT 122
 watchdog 123–124
 MAX691A 120–121, 140
 MC146818A 53, 68
 battery-backed RAM 129
 see also Real-Time Clock 129
 MComp.BAT 178
 MC-Net 89–90
 MDA 257, 259
 memcpy() 250, 256
 MEMCS16 (Memory Card Select, 16-bit)
 97
-

memory

above 1 MB	277
address calculation	137
BIOS scan	94
hole	277
I/O cards	94
no wait state access	99
PC layout	93
system board BIOS	94
video RAM	93
memory model	
ROM	167
Small	27, 166
Tiny	27, 134, 164
MemR	278
MemTest.C	99–100, 104, 125
MemW	278
metastability	216, 243
Microchannel	35
Microsoft C	34, 179
MON86	22–25, 51, 81
moral of the story	72, 79, 92
MSDOS.SYS	13
MSR (Modem Status Register)	91
multitasking	162
multithreaded programs	162

N

NAND	216
nanosecond	53
NMI	122–123, 129, 138–141, 191
disabling	129
glitches	139
handler	130, 139, 141
handler chaining	140
hot	129, 140
IRET	131
sources	129
NMI (Non Maskable Interrupt)	72
nonvolatile RAM	
see also battery-backed RAM	144
nonvolatile storage	167
null modem	23

O

octothorpe	268
Once Upon A Time	89
one hand clapping	220, 234
Optrex DMF651	195
Oriental Pepper Soup	36
OS/2	188, 191
oscilloscope	6
see also scope	47, 244
outport	185
output port	229
outpw()	50, 255
outrage, who's responsible	180

P

PAL	41, 44, 167
parallel port	85, 156
PC Address Buffers	217, 224
PC AT, see also IBM PC AT	133
PC Compatibility Barnacles	56, 74, 79, 92–93, 95, 257
see also barnacles	35
PC serial numbers	147
PC, see also IBM PC	93
PC/XT, see also IBM PC/XT	257
PCI bus	280
PCLK (Processor CLock)	55
PEEL	41
Pentium	26, 35, 79–80
Pentium Pro	35, 56, 79, 105, 253
performance	50
Phar Lap Software	278
PIC microcontroller	74
PM (Protected Mode Enable) bit	142
poke()	104
pop quiz	220
PortTest.C	47–48
power failure	130, 139
Power Good	120–123
Power OK	121
prefetch	50
printer port	24, 36

Programmable Interrupt Controller, see also
 8259 74
 programmable logic devices 41
 protected mode 21, 252, 278
 protection violations 191
 prototype board 45
 PS/2 Model 80 147, 164
 pseudorandom pattern 230
 PSP (Program Segment Prefix) .. 135, 137
 PSR32K.HEX 100
 pullup resistors 43
 pulse widths 153
 pushbutton switch 133, 168, 190
 putch() 27
 PWM 150

R

RAM

 battery backed 113, 165
 data retention 113
 low-power mode 114
 nonvolatile 144
 RAM refresh 50
 RAMTest 125
 rand() 100
 Raw Frame Sync 214, 220
 Raw Line Sync 214, 220
 RD (Read Strobe) 59
 real mode 252
 Real-Time Clock 191, 253
 see also MC146818A 143
 Refresh (bus signal) 279
 relocatable programs 182
 remote debugging 189
 ResDrv (RESet DRiVers) .. 121, 123–124,
 200, 226, 275
 Reset 12, 123–124, 126, 129, 142
 80286 142
 adapter kludge 123
 button 12, 275
 command 143
 shutdown reason code 143
 system board 123

ROM

 performance 99
 shadowing 97
 ROM memory model 167
 see also memory model 167, 170
 ROMScan.C 106
 Rotate.ASM 24
 Row Select register 197
 RS-485 90
 RTFM 92
 rule of thumb 41
 runtime library 183

S

Schmidt Trigger 216
 Schottky diode 117
 scope, see also oscilloscope 48, 85
 SCR latchup 200
 SCSI 124
 Second Program 21
 Second-Worst Hack 190
 segment
 offset 247
 Segment Not Present error 191
 Segmented Memory Monster 180
 serial number 173
 serial port
 see also COM port 23
 SerIO.ASM 27
 SerIOFF.ASM 28
 SerNum.C 158
 SETAE instruction 251
 seven-segment display
 see also LED display 45
 Sharp LM64015T
 see also LM64015T 207
 sharp objects 61
 Sherlock Holmes 91
 shutdown 191
 see also Reset 144
 Sieve 184, 187–188
 SLIB 28, 92
 command line example 28

Small 167, 188
 see also memory model 27
SMemR 113, 224, 230
SMemW 113, 224, 226, 230
software interrupts
 see interrupt 72
software piracy 147
software, simple matter of 166
sound board 56, 278
SP (Stack Pointer) 137, 171, 191
 maximum value 171
SRDY (Synchronous ReaDY) 38, 97
SS 137, 171–172
SS:SP 171–173, 175
stack 165–166
 blown 89
 monitoring 89
startup code 166, 177, 183
 segment setup 186
SVGA 170, 205
Synchronous Ready, see also SRDY ... 38
SysClk 202, 211, 216, 224–226
 noise 216
system board 53

T

Tab character 265
target system 189
TASM 134
TDREM 180, 189–190, 192
three-finger salute 164
TimeTest.C 61, 69
timing analysis 59–60
timing loop 153
Tiny 167, 188
 see also memory model. 27
Tiny.LIB 28, 92
TLY-365-121 209, 235, 250, 272
TO-220 226
TO-92 149
trace outputs 90
traps 73
trimpot 48

triple fault 191
TSR 91
Turbo Debugger 189–190, 192

U

UART

 see also 8250, 16550 91
UL 117, 120
User PC 23, 25

V

variables

 initialized 166, 171
 uninitialized 165–166, 171
vertical scrolling 267
Very Good Idea 126
VGA 93, 170, 193, 205, 259, 261
 font 265

W

wait state 45, 48, 50, 61, 225
 generator 61
walk in the woods 124
watchdog 123, 139, 141, 162, 173
 bit patterns 128
 disable 124, 126
 see also MAX691 123
 variable rate 124
 WatchBits 126, 128
 WatchFlag 126
 WatchPending 126, 128–129
WatchPending 128
WIBNI 122
Windows 95 188
woodworkers 228
Worst Hack 144, 190–191
write protection 113, 165

Firmware & Schematics on Diskette!

The diskette accompanying *The Embedded PC's ISA Bus* by Ed Nisley contains all the C and Assembler firmware from the book, with **complete source listings and executable files**:

- **Boot loaders** for your embedded PC programs
 - Boot programs directly from a DOS diskette, without using DOS
 - Start during power-up from code in EPROM or nonvolatile RAM
- **Sample firmware** programs
 - Use ordinary **COM** files or **EXE** files converted to binary
 - Generate embedded programs with familiar DOS/Windows compilers!*
 - Hardware interrupt handlers that produce timing trace signals
 - Non-maskable interrupt handlers for hardware error detection
 - BIOS extensions written in Assembler or C
 - Build a no-moving-parts embedded PC from a standard system board!*
 - Real-Time Clock interrupts and alarms using BIOS functions
 - Unique hardware serial numbers for positive identification
 - Direct character LCD interfaces for text and graphic LCD panels
 - Using the PC's Worst Hack
 - ANSI cursor control sequence decoder
- **Hardware diagnostics** that check out your construction
 - A parallel port circuit with LEDs, trace outputs, and switch inputs
 - ISA bus data buffers and I/O address decoders
 - 16-bit LED digit display and switch input ports for simple digital I/O
 - 82C54 programmable timers for periodic interrupts and hardware control
 - ISA bus memory circuits:
 - EPROM for low-cost program storage
 - EEPROM for data logging and on-the-fly program updates
 - Battery-backed RAM with firmware write protection
 - A watchdog timer that detects program lockups
 - A power monitor that ensures proper shutdown after a detecting a glitch
 - Dot-drawing and text routines for a big, bitmapped graphic LCD panel
- **Measurement firmware**
 - I/O wait state adjustment to show the effect of slow ISA bus cycles
 - ISA bus memory and I/O access times
 - Hardware interrupt response time measurement
 - Non-maskable interrupt sources and conditions
- **...and much more!**

The diskette also includes **complete schematics** for all the ISA bus hardware projects. You can import the **TIFF** files into your graphics program or word processor, then print them out in one piece for easy reference. See the Schematics appendix for a preview!

The Embedded PC's ISA Bus

"Ed is the only guy I know who can feel completely at home with a soldering iron in one hand and a PC mouse in the other. The consistent wit and wisdom he has applied to documenting an abundance of firmware and hardware applications over the years has truly provided a cornerstone of technical knowledge for an entire generation."

— Steve Ciarcia, Publisher,
Circuit Cellar INK

"There may come a day when Windows CE (or whatever they call it this week) greets you from an LCD on the side of your toaster. But until then, it's you, the bits, and the bus. If there's ever been a better book than this for developing those new skills, or scraping the rust off some very old skills, I've yet to see it."

— Jeff Duntemann, KG7JF,
Editor-in-Chief, *Visual Developer*

"The Embedded PC's ISA Bus is much more than a description of using this ubiquitous bus. It's a course in using the PC as an embedded processor. Read the book—even better, build the projects—and you will have progressed from embedded apprentice to journeyman."

— Jack Ganssle, columnist,
Embedded Systems Programming

Ed Nisley began his career with a decade at IBM, designing and building neat stuff like mainframe CPUs, SPC instrumentation, bit-sliced graphic firmware, and even a digital video disk.

He is now a registered Professional Engineer living in Raleigh, NC, stuffing firmware into industrial controls, prototype consumer electronic devices, and amateur radio widgets.

Real-world success with embedded ISA-bus systems requires mastery of both hardware and software techniques, comprehensive knowledge that can't be found in dry bus specs or simple hobby projects. Ed Nisley has spent 20 years in the trenches of embedded programming and hardware design; in his popular "Firmware Furnace" columns in *Circuit Cellar INK*, and now here in these pages, he maps the territory where code meets solder.

This book is the most advanced and comprehensive compendium of practical techniques for bare metal ISA bus projects, filled with essential information that will help:

Engineers	design better embedded PC hardware
Programmers	write high-performance embedded firmware
Technicians	debug obstinate ISA bus gadgets
Students	relate abstract theory to applications
Enthusiasts	control unique projects with standard PCs

The topics range from pure firmware to detailed hardware, including the BIOS routines and circuitry on the PC's side of the ISA bus that you can use throughout your embedded systems designs:

Firmware

Boot	diskettes without using DOS
Examine	I/O ports, memory, IRQs, and bus cycles
Control	hardware with C and Assembly language
Adapt	working source code and programs

Gadgets

Build	ISA bus memory and I/O circuits
Protect	code with watchdogs and power monitors
Identify	hardware with unique serial numbers
Display	text and graphics on big LCD panels

Practical Tricks

Run	code in EPROM, EEPROM, or nonvolatile RAM
Measure	wait states and interrupt latency
Fix	hardware problems with firmware
Debug	firmware using hardware outputs

Software on the accompanying diskette tackles these issues and more, with complete C and assembler code: check the listing inside the back cover for details!



PEER-TO-PEER
COMMUNICATIONS
www.peer-to-peer.com

ISBN 1-57398-017-X

