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*`** **$\overline{\texttt{SMemW}}$**

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: *italic* 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