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.

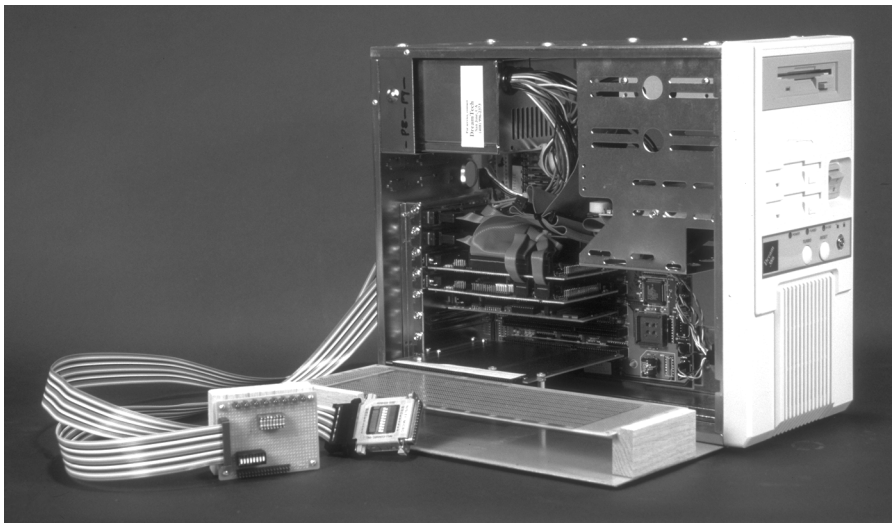
Essential Hardware

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

              DB        'Firmware'          ; 03 OEM name and ver (8 chars)
SectorSize    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 secor
             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

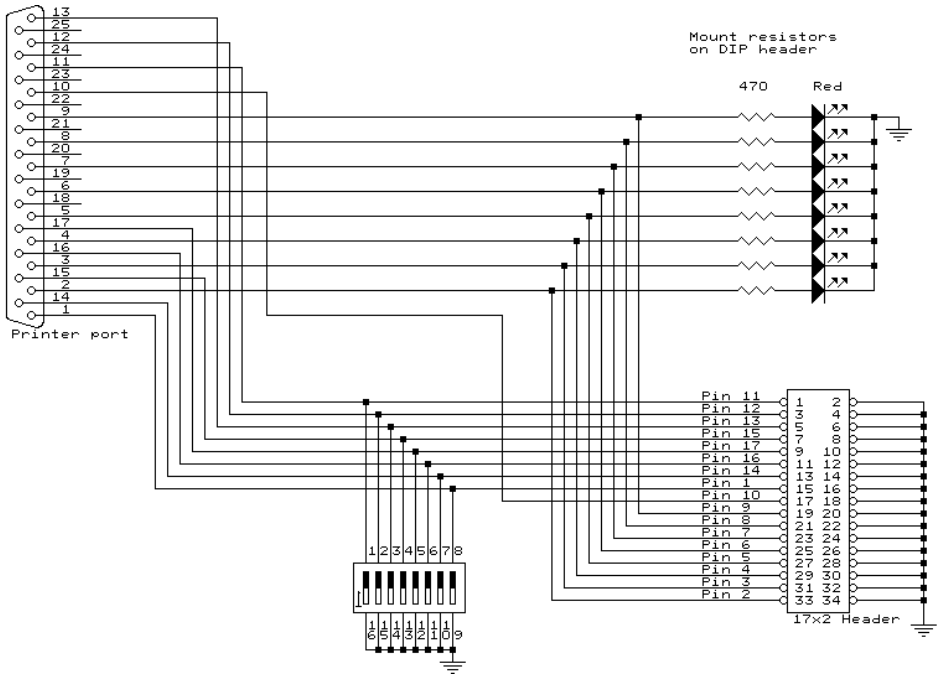
The canonical First Program on any system displays a trivial message like C's "Hello, world" routine, but our target PC doesn't have a display yet. Think small: how about changing a single I/O bit? An LED connected to that pin will make the result visible. Unfortunately, our target system lacks even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