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.

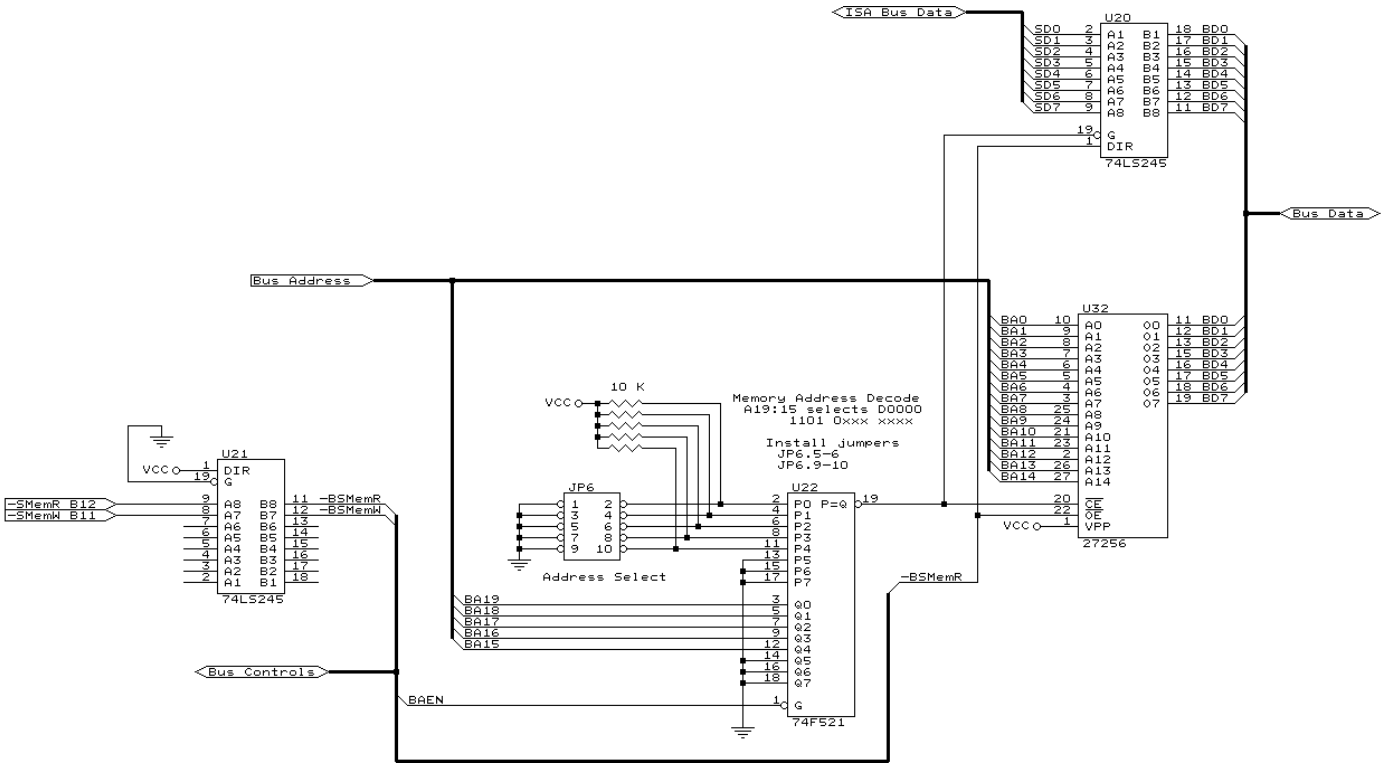
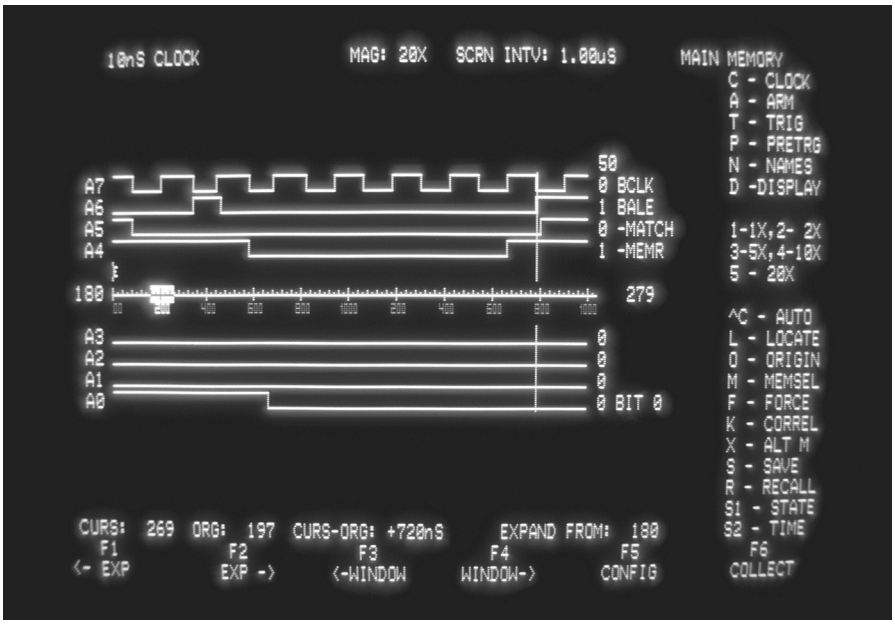


Photo 1
This logic analyzer trace shows what happens when the CPU reads a single EPROM byte. The access begins with the rising edge of BALE and ends six BCLK cycles later when -SMemR (mislabelled -MEMR here) goes high.



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 *x*86 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.

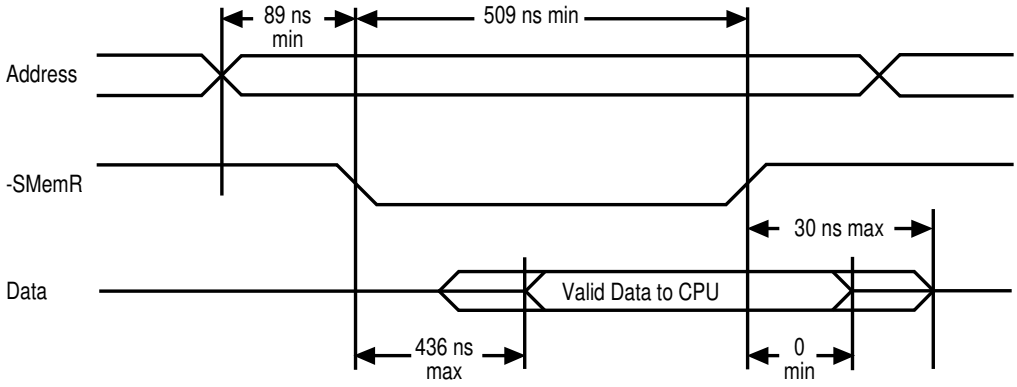
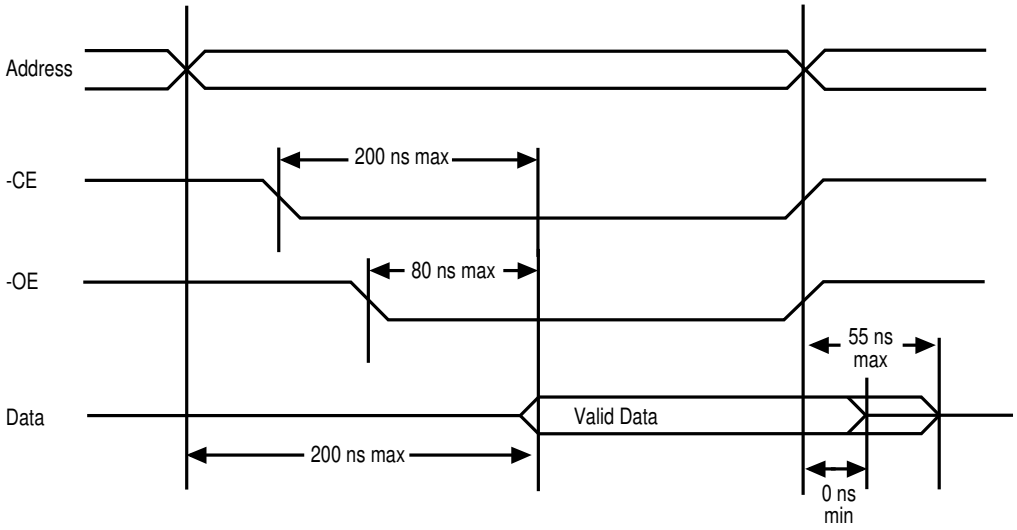


Figure 2b
These timings show the read cycle for a 200 ns 28C64A EEPROM and are typical of EPROMs as well. The bus buffers shown in the schematic truncate the rather long maximum data hold time.



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 µs and the maximum wait-state delay shouldn't exceed 1 µs.

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.

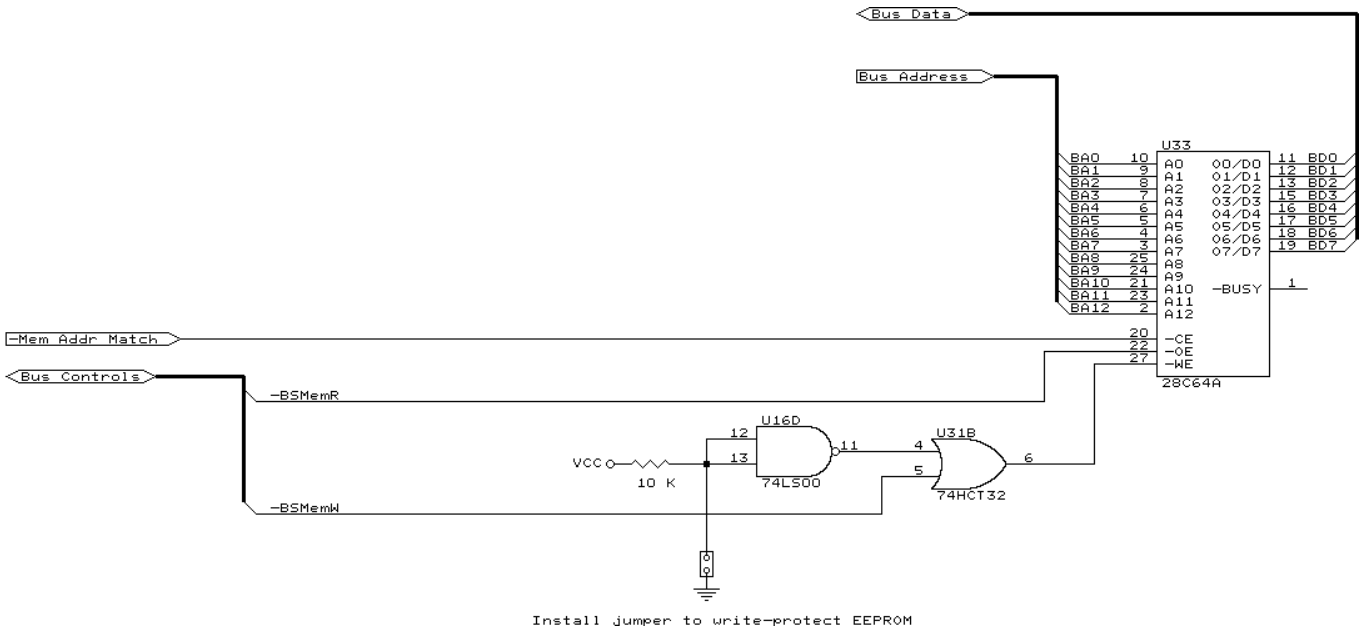


Figure 3a
ISA bus memory write accesses are similar to reads. Note that the data may not be valid for quite a while after the leading edge of -SMemW, so the destination must rely on the trailing edge for precise timing.

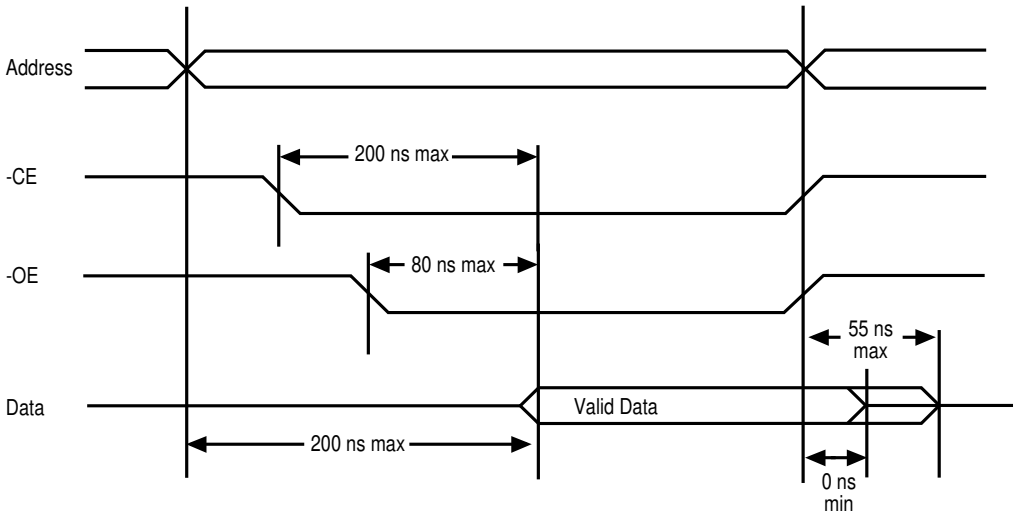
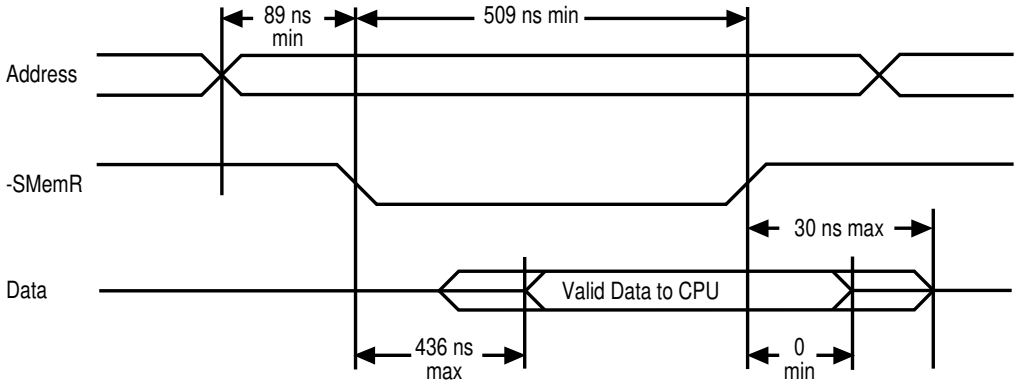


Figure 3b
The write cycle for a 28C64A EEPROM seems essentially identical to a standard RAM, but the EEPROM cannot accept more data for about a millisecond after beginning a write. This timing diagram shows the signals for the first part of the process; the text in this chapter describes how the firmware 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("Timout 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 µs. 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 {
         putstr("--\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!

                RETF                        ; and return to BIOS!
Onward:

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

                MOV     CX,0
Wait1:          LOOP    Wait1

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

                MOV     CX,0
Wait2:          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.