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.

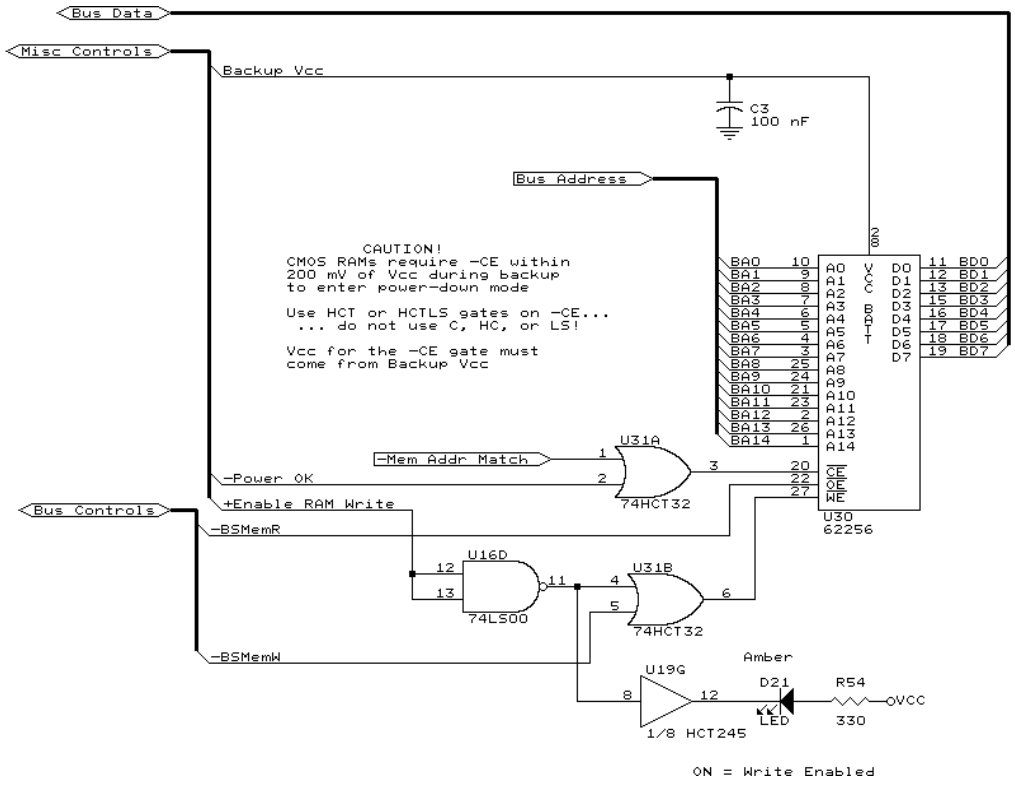


Figure 1
Although 8 KB and 32 KB RAMs, EPROMs, and EEPROMs all come in a 28-pin DIP package, they have some crucial differences. This table gives the connections for the oddball pins. My Firmware Development Board sprouted a cluster of jumpers around the memory socket to cope with all the choices.

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.

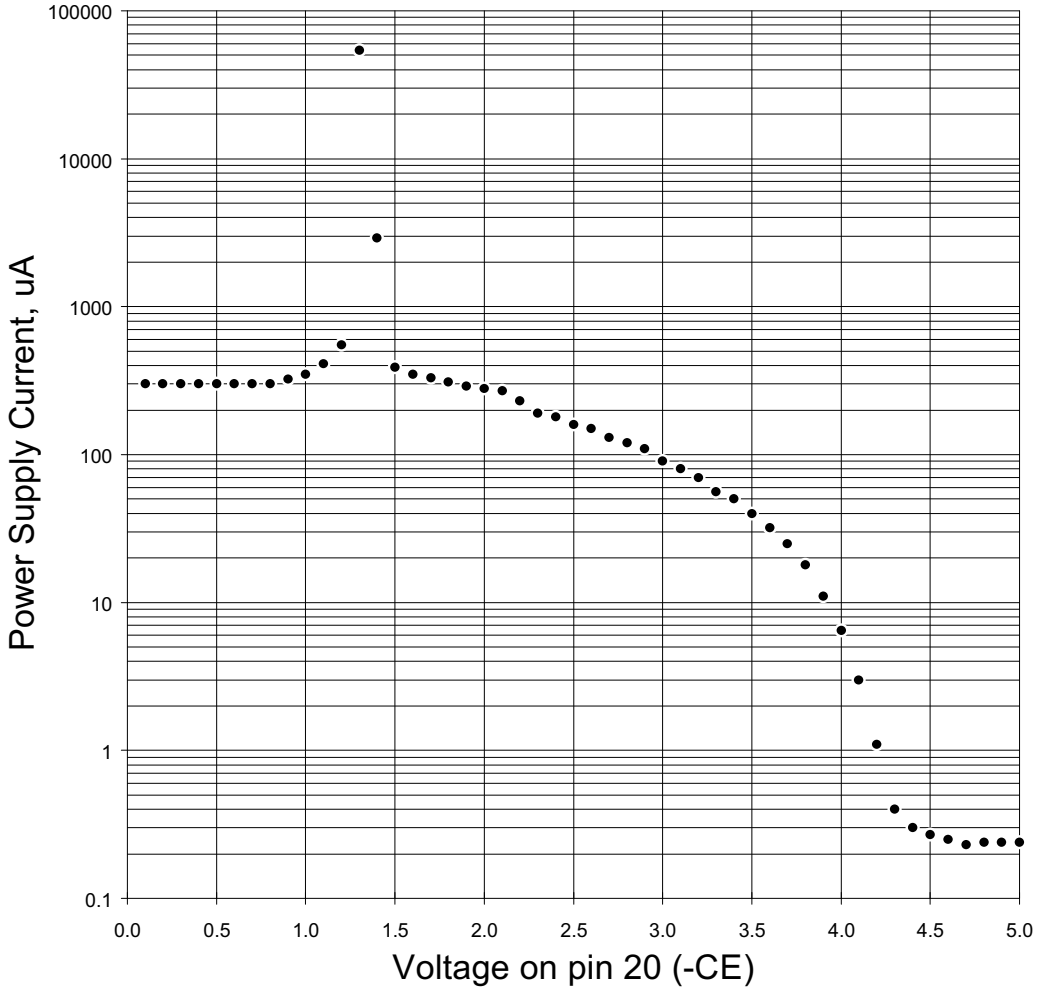
Figure 2 shows the result of a simple experiment measuring supply current as a function of **-CE** voltage. The vertical axis uses a logarithmic scale to compress the current range, making it easy to see when standby mode kicks in at about 4.5 V. I ran the RAM at +5 V, but the curve has a similar shape 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 -CE input voltage affects the current drawn by a static RAM at V_{CC} = 5 volts, but a similar curve applies for 3 V battery backup operation. The -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.

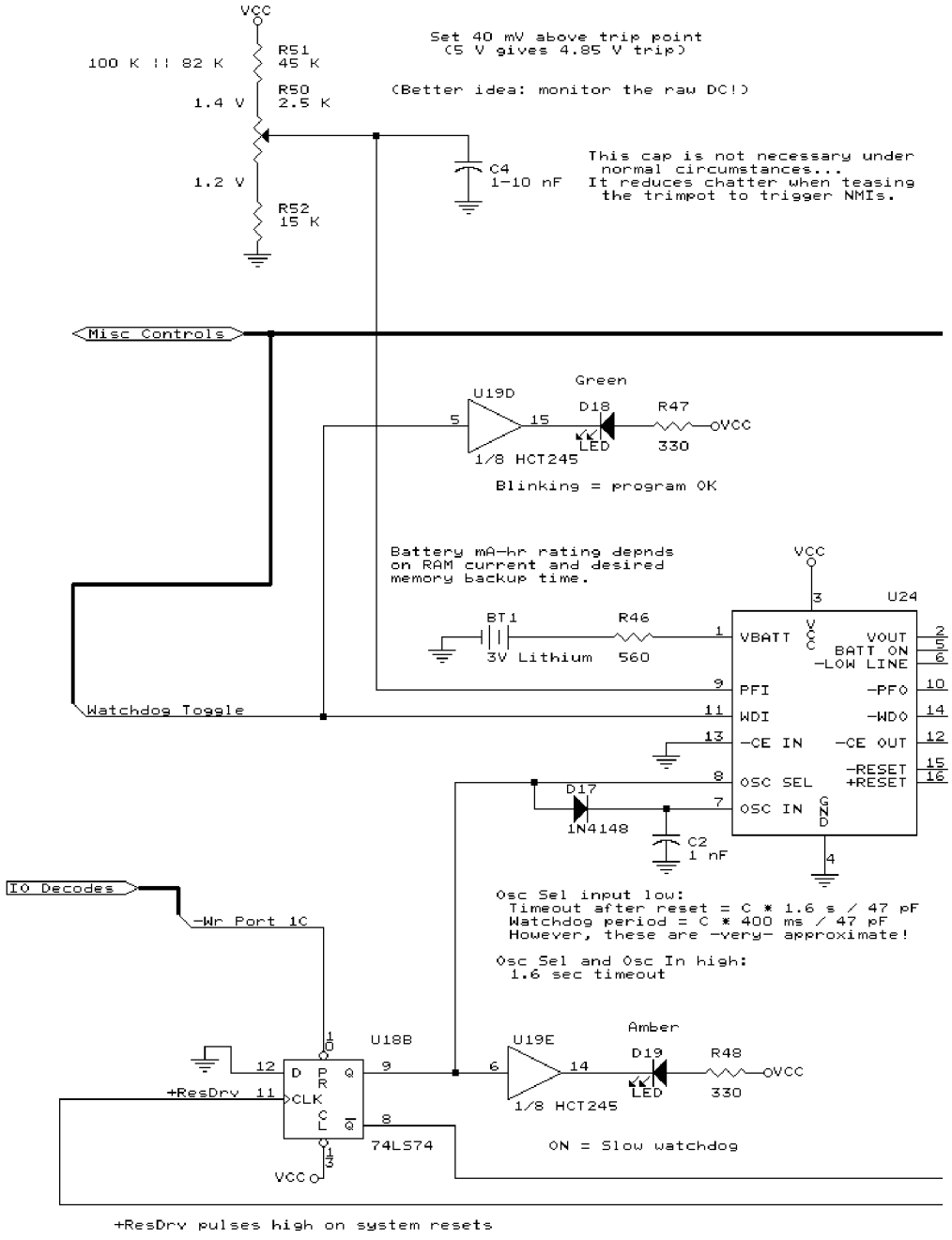
A NEC Static RAM Application Note I reviewed for this project mentioned several UL requirements for lithium cell backup circuits. Even if your product specs don't *require* UL approval, the guidelines make sense. Bear in mind that I have not read the UL regulations themselves, so don't depend on *my*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

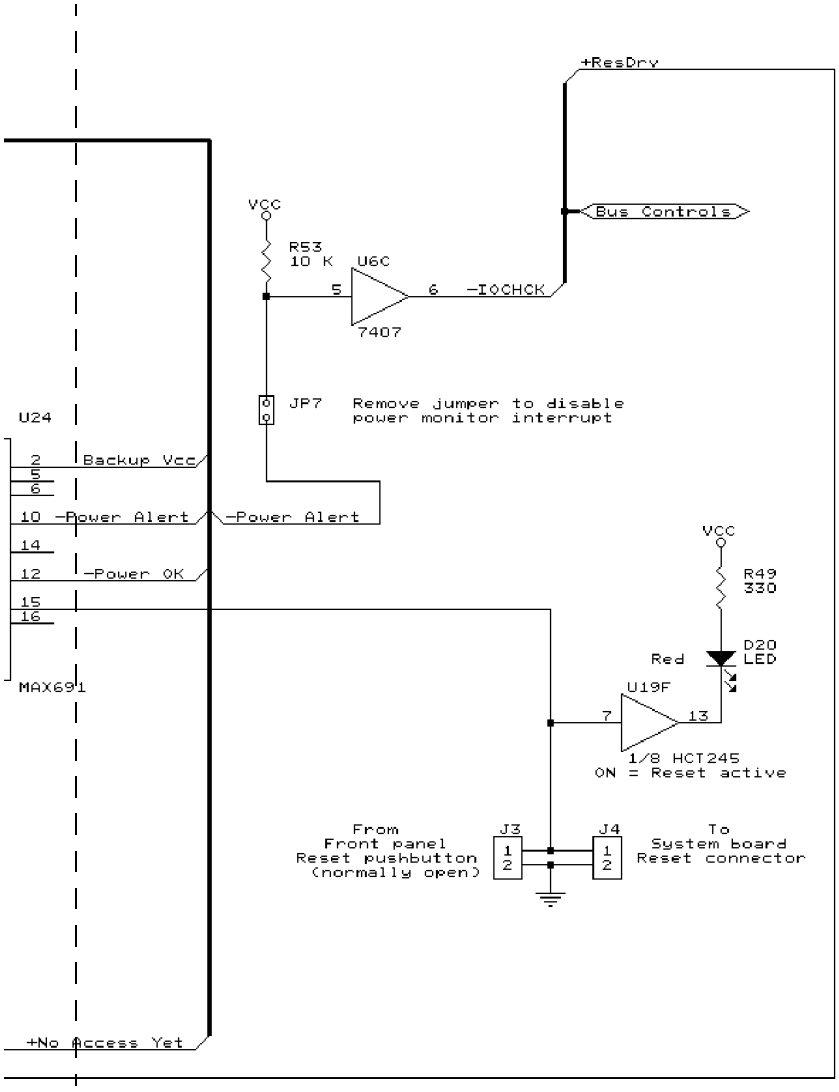
$$(0.01 \times 250\ \text{mAh}) \div (8\ \text{hours/day} \times 365\ \text{days/year} \times 10\ \text{years})$$

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.


VCC
100 K || 82 K
R51
45 K
1.4 V
R50
2.5 K
1.2 V
R52
15 K
Set 40 mV above trip point
(5 V gives 4.85 V trip)
(Better idea: monitor the raw DC!)
C4
1-10 nF
This cap is not necessary under
normal circumstances...
It reduces chatter when teasing
the trimpot to trigger NMIs.
Misc Controls
U19D
Green
D18
R47
VCC
LED
330
1/8 HCT245
Blinking = program OK
Battery mA-hr rating depends
on RAM current and desired
memory backup time.
BT1
3V Lithium
R46
560
U24
VBATT
VOUT
BATT ON
-LOW LINE
PFI
-PFO
WDI
-WDO
-CE IN
-CE OUT
-RESET
+RESET
OSC SEL
OSC IN
GND
Watchdog Toggle
D17
1N4148
C2
1 nF
IO Decodes
-Wr Port 1C
U18B
+ResDrv
CLK
74LS74
U19E
Amber
D19
R48
1/8 HCT245
Osc Sel input low:
Timeout after reset = C * 1.6 s / 47 pF
Watchdog period = C * 400 ms / 47 pF
However, these are -very- approximate!
Osc Sel and Osc In high:
1.6 sec timeout
ON = Slow watchdog
+ResDrv pulses high on system resets

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 μA 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 V - 3 V) ÷ 5 mA = 500 Ω

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 CheckK) 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.

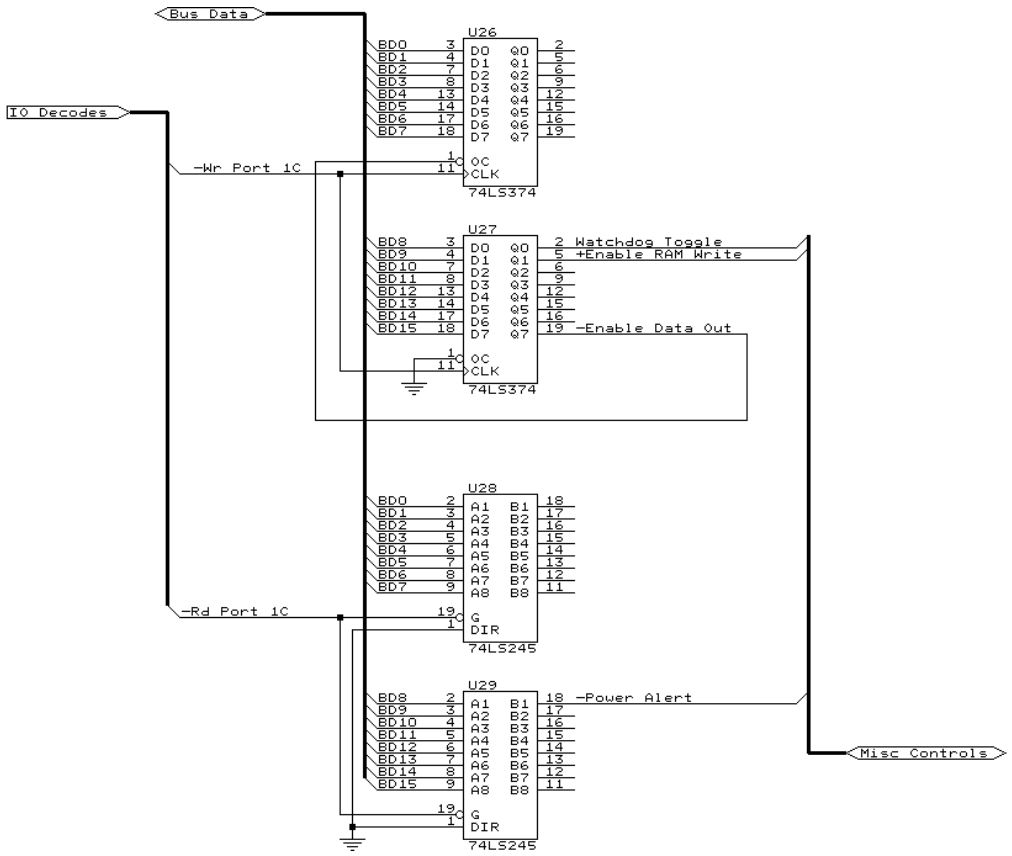
Although the MAX691 data sheet has equations giving the external capacitor value for a given timeout, I've found that they provide only rough starting points. You should probably perform some experimentation to find the right value for your application and verify the results. Remember that a slow watchdog 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,CtlsCopy         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       CtlsCopy,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 µs 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:>Int02Off      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.