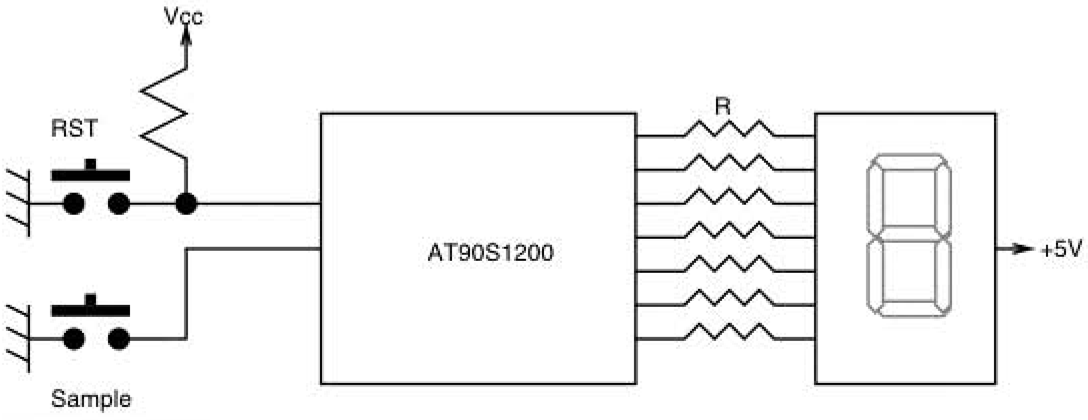

FIGURE 10.6 **Block diagram for an alternative electronic dice circuit. R is a current-limiting resistor for each of the segment LEDs of the seven-segment display.**

Figure 10.7 shows another possible alternative, not much different from our actual implementation, except that each LED is driven independently by an output pin of the processor. That only puts more demands on the processor resources. For this simple project using an AT90S1200, that may not be a big deal, as the extra required output pins are available, but if you want to port it to a different processor with fewer pins, that may not be possible. Also, this scheme, like the last one, requires extra resistors.

10.6 Code Development

How does one go about developing the whole system to be even as simple as the present one? Does one put together the hardware and then write code for it, or develop code first and then build the hardware? This is a tough question and there is no unique answer to this. It will depend on a particular application.

Generally, it is a good idea to write and test as much code as possible using a simulator or a prototype board. For this application, I used the Atmel's evaluation board, MCU00100 (this has now been superseded by the more advanced STK200 and STK300 boards provided by Atmel) and wrote and tested all of the code. Once the code worked as I wanted, I went ahead and built the circuit on a general-purpose PCB as described in the next section.

The code itself evolved. I present the three versions of code, each with some improvement over the previous version. These three versions of code are put in assembler files named ugly_dice.asm, bad_dice.asm, and good_dice.asm.

The program in ugly_dice.asm was the result of the first attempt at programming the dice. It differs from the code in bad_dice.asm in the way the random number is output on the LEDs. I ended up writing complex code to essentially implement an if-then-else structure. Later I realized that for our simple case, such a complex coding scheme is not required. The code in ugly_dice.asm was then simplified and the resulting code is presented in bad_dice.asm.

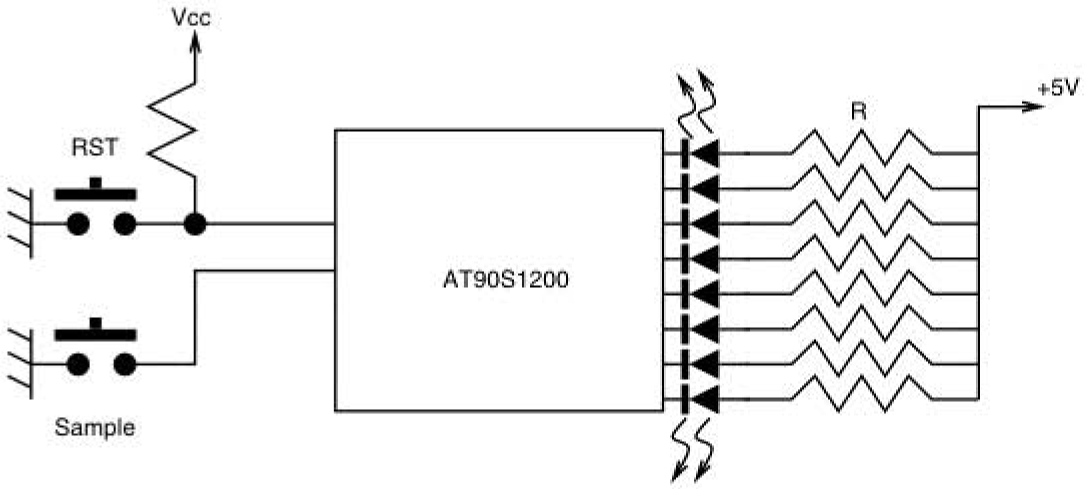

FIGURE 10.7 **Block diagram for another alternative electronic dice circuit. R is a current-limiting resistor for each of the LEDs. The seven LEDs are again arranged as in Figure 10.1.**

Finally, power-saving features were added to bad_dice.asm, resulting in the final code in good_dice.asm.

Anyway, let us consider the code presented below.

```
;ugly_dice.asm
;
;In the program I just initialize the Timer0 to count up
;using the CLK/1024 as a clocking source. The Timer0 merrily
;counts up, resets to 0 and starts all over again ad infinitum
;PortB is configured as all output and PortD bit 0 (PD0)
;as an input. The program waits for a key press and then
;takes a snap shot of the Timer0 (register TCNT0), and then
;is further processed before put on the display. read on..
;Dhananjay V. Gadre
;19th February 1999
.include "1200def.inc"
.cseg
.org 0
       rjmp    RESET           ;Reset Handle
RESET:  ldi r16, 0b00000101 ;DIV1024 selected for timer0
       out TCCR0, r16        ;timer0 counts up now
       ldi r16, 0b11111111   ;configure PORT B for all outputs
       out DDRB, r16
       cbi DDRD, 0           ;configure pin PD0 as input
       sbi PORTD,0           ;set the PD0 so that it can act as i/p
                             ;with internal pullup
get_t:  rcall sw_input       ;wait for the switch to be pressed
        ldi r18, 0b11111111  ;blank off the previous display
        out PORTB, r18
        rcall delay          ;wait for some time
        rcall delay
        rcall delay
        rcall delay
        rcall delay
        rcall delay
        in r17, TCNT0        ;read the Timer0 value
more:   mov r18, r17         ;copy r17 to r18
```

```
;now check if the number is less than $06
compa:  cpi r18, $06
       brlo enough          ;yes, then nothing more
                            ;prepare to output it
                            ;else
       clc                  ;clear carry for sub
       sbci r18, $06        ;and subtract $06
       rjmp compa           ;do it till the remainder is
                            ;less than $06
enough:   cpi r18, $00      ; OK, now the number in r18 is
                            ; between 0 and 5
;The following piece of code makes this program ugly
;The identification of the number and display can be handled
;more cleverly as in bad_dice.asm
      brne not_0            ;number is not zero
      ldi r18, 0b11111110   ;number is 0, so display 1 on the
      out PORTB, r18        ;LEDs, i.e. light up LED B2
      rjmp get_t
not_0:  cpi r18, $01        ;check if the number in r18 is 1
      brne not_1
      ldi r18, 0b11111101   ;yes, so light up B1 and B3
      out PORTB, r18
      rjmp get_t
not_1:  cpi r18, $02        ;check if it is 2
      brne not_2            ;no it is not
      ldi r18, 0b11111100   ;yes, it is. so light up B1 and B3
      out PORTB, r18
      rjmp get_t
not_2:  cpi r18, $03
      brne not_3
      ldi r18, 0b11111011
      out PORTB, r18
      rjmp get_t
not_3:  cpi r18, $04
      brne not_4
      ldi r18, 0b11111010
      out PORTB, r18
      rjmp get_t
not_4:  ldi r18, 0b11111001
      out PORTB, r18
      rjmp get_t
;Delay subroutine
;uses registers r16 and r18
;values are set arbitrarily
delay:    ldi r16, 0b11111100
loophere:  ldi r18, 0b10001110
decrement: dec r18
        brne decrement
        dec r16
        brne loophere
        ret
;returns when a key on PD0 is pressed and released
;till then it loops waiting for the key press
sw_input:  in r16, PIND       ;input PORTD pin value
        andi r16, 0b00000001 ;isolate PD0 state
        brne sw_input        ;if switch is not pressed, loop back
        rcall delay          ;if pressed, then wait some time to
                             ;ward off the switch bounces
pin_0:    in r16, PIND       ;now check if the switch is released
        andi r16, 0b00000001
        breq pin_0
        rcall delay
        ret                    ;switch is now released..go back
```

The code in bad_dice.asm is the same as in ugly_dice.asm, except for the part that displays the resulting number (between 0 and 5). This code is as follows:

```
;code segment from bad_dice.asm
enough: ldi r16, $01          ;since the number is between 0 and 5
                              ;add '1' to make it between 1 and 6
     add r18, r16
     com r18                  ;the output display LEDs are arranged
                              ;as active low, so complement the result-
                              ing
                              ;number
      ori r18, 0b11111000     ;set the other unused port pins
                              ;to an inactive state
      out PORTB, r18          ;display it!
      rjmp get_t              ;that it! go get more
```

The program uses free-running Timer0 to get the randomness. However, it is possible to use other methods of random number generation; a prominent one is to use Linear Feedback Shift Register (LFSR) as outlined in a previous chapter. The advantage of the LFSR method is that it only requires a seed to arm the LFSR, and then it will churn out random numbers.

For this simple case, where the Timer0 is not going to be used for any other task and is always available, we could use it full time. The Timer0 is clocked at 1/1024 the clock frequency, and since the state of the timer is not known to the user, reading it gives a certain randomness suitable for our purpose.

The program in good_dice.asm builds upon the code in bad_dice.asm by adding an interrupt subroutine that occurs every time TCNT0 overflows. With a clock input to the counter of clk/1024, this occurs every 128 ms for a 2.000-MHz clock. The initialization code in good_dice.asm initializes three registers as low_timer, med_timer, and high_timer to zero. The interrupt subroutine increments these registers each time the timer overflow occurs.

By looking at the values in med_timer (you can modify the Timer0 ISR so that it powers down after a longer time), the Timer0 ISR determines whether to blank off the display with power-down sleep or not.

The Timer0 interrupt subroutine from good_dice.asm is illustrated below.

```
;Timer0 ISR from good_dice.asm
;low_timer, med_timer and high_timer are three registers
;that hold elapsed time. These registers are cleared to zero
;by main program every time a key is pressed, else their value
;builds up and when it exceeds certain value, this ISR powers
;down and puts the processor to sleep
Timer_int: in save_status, SREG  ;save status in reg save_status
        cpi med_timer, $01        ;compare med_timer to 01
        brne skip_it              ;is equal then prepare to power
                                  ;down
        ldi temp, 255             ;tri-state all outputs
        out PORTB, temp           ;configure all ports as O/P
        out PORTD, temp           ;and set all values to '1'
        out DDRD, temp
        out DDRB, temp
        in temp, MCUCR          ;now set SE and SM bits in MCUCR
        ori temp, $30           ;register and make them '1' to
        out MCUCR, temp         ;select power down mode of sleep
        sleep                   ;now sleep off
        out SREG, save_status   ;restore status
        reti                    ;return.. well it doesn't matter
skip_it:  inc low_timer         ;if not, increment low_timer
        cp low_timer, r22       ;if it overflows to 0, then incr
        breq inc_med            ;med_timer too.
        out SREG, save_status   ;else restore status and return
        reti
inc_med:  inc med_timer         ;incr med_timer and check if it
        cp med_timer, r22       ;overflows
        breq inc_high           ;if yes, then incr high_timer
        out SREG, save_status   ;else restore status and return
        reti
inc_high:  inc high_timer       ;incr high_timer
        out SREG, save_status   ;restore status and return
        reti
```

10.7 Fabrication

Figure 10.8 shows the photograph of the dice circuit assembled on a general-purpose PCB. The circuit was assembled using the same general fabrication techniques presented in a previous chapter. For the AT90S1200 controller, we have used a 20-pin socket, which is a

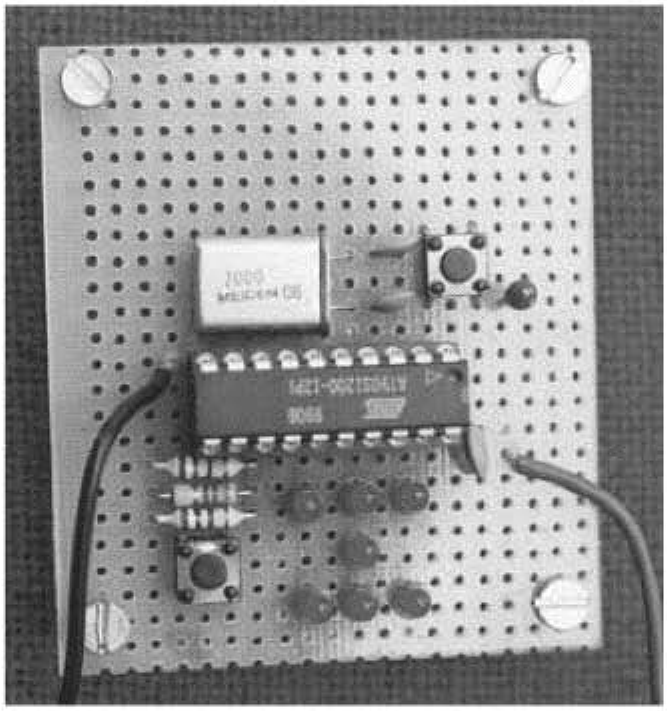

FIGURE 10.8 Photograph of the completed dice circuit board.

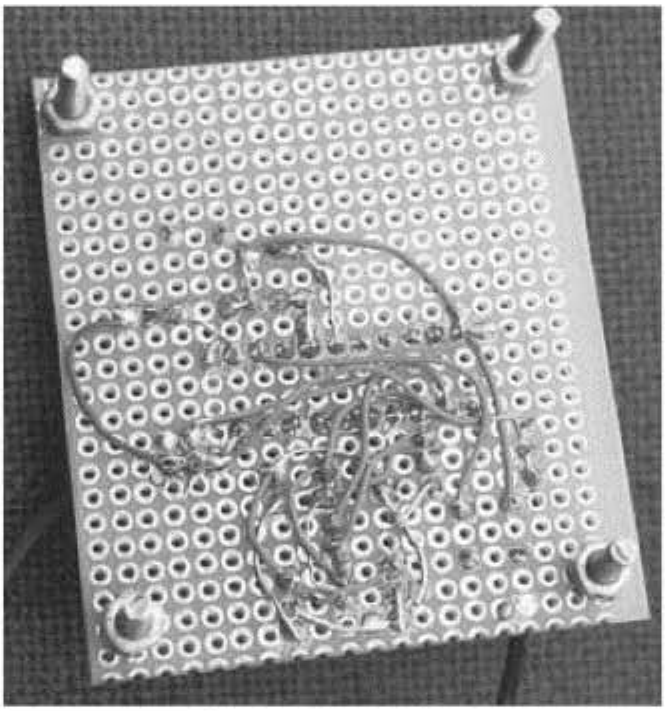

FIGURE 10.9 **Photograph of the solder side of the dice circuit board.**

good idea for prototypes. If the processor is to be dedicated to this circuit and it is felt that the code works satisfactorily and would not need further revision, in subsequent boards the processor could be soldered directly onto the PCB to save the cost of the socket.

It is a good idea to solder the resistors, the socket, and the tiny switches right at the beginning. Later, the capacitors and the crystal are soldered and in the end, the LEDs are put in place. After all the components are soldered, proceed to wiring the components together. Figure 10.9 illustrates the solder side of the dice circuit board.

10.8 Testing

After the circuit is assembled on the general-purpose board, it is a good idea to inspect for possible shorts and open or unwanted electrical connections. A multimeter comes very handy to check for open connections and shorts in the circuit board. Check if the supply voltages are connected at the right place and the LEDs and the processor are mounted with the right polarity. Connecting the processor the other way around will lead to a lot of grief.

Once you are satisfied that all the connections are proper, insert the processor in its socket and apply power to the circuit. The applied voltage should be +5 V DC on the processor supply input pins. For this project, all the LEDs should light up immediately. Press and release the sample switch, and a random number between 1 and 6 should appear on the LEDs. Press and release this switch many times and check that the sequence in fact appears to be random. Also check that after pressing and releasing the switch, the display blanks briefly before the new number is displayed. If the circuit seems to function as described, congratulate yourself. If not, get ready for some detective work. First and foremost, check with a multimeter (in the DC voltmeter mode) that the supply voltage appears at points that it should, such as between pin 20 and ground of the processor. Next check if the reset switch is not sticking. This can be checked by monitoring the voltage on the reset pin (pin 1) of the processor. Put one of the multimeter probes at pin 1 and the other at ground (pin 10). When the RST switch is not pressed, the voltage at pin 1 should be around the supply voltage of +5 V. Now press the RST switch; the pin 1 voltage should be 0 V now. Release the RST switch and watch this voltage go up to +5 V again. If this is not happening, suspect your reset circuit composed of the resistance R4, capacitor C4, and the RST switch J1.

If the circuit seems to reset properly and is still not functioning as expected, time to look for the oscillator circuit. For the components illustrated, the circuit works at 2.000 MHz, and monitoring X2 (pin 4) should show sinusoidal oscillations at 2 MHz on an oscilloscope. If this is not happening, the culprit could be the crystal or the two capacitors C2 and C3.

Check the operation of the sample switch to see that the logic at pin PD0 changes when the switch is pressed and when released. Another possibility could be that the LEDs are arranged the other way around; check that, too, and correct that if needed. This covers the possible ways in which things could go wrong in this small project.

10.9 Usage

Well, using the dice is as simple as saying cheese. Just connect the dice to a suitable +5-V source capable of supplying a few milliamperes (25 mA or so) and you are ready to go.

To adapt the circuit to run off a battery, there are two options; either to use a +9-V box battery or use four 1.5-V cells. If you want to use a +9-V box battery you could use a 78L05 voltage regulator to get the +5-V supply voltage. The 78L05 voltage regulator comes in a small TO-92 package and can be easily put on the same PCB as the dice. However, since it requires a minimum voltage of +6.7 V to provide the +5-V output, it cannot be used if you choose to use four 1.5-V cells. Using four cells of 1.5 V each will give off +6 volts, and to get +5 V out of it you could use the LP2940 voltage regulator as described in the earlier chapter on system design. You could also choose to drop the voltage to something close to +5 V with the help of two series diodes.

Once you put the required power supply in place, just punch away at the sample switch and watch the dice roll off.

10.10 Power Consumption

As a portable utility, it will most probably be battery powered. The main concern with battery-powered devices is the power consumption, both while operating the device and when the device is not in use. The idle and power-down features of the AVR controllers come in handy in minimizing the power usage, especially when the device is not in use (Figure 10.10).

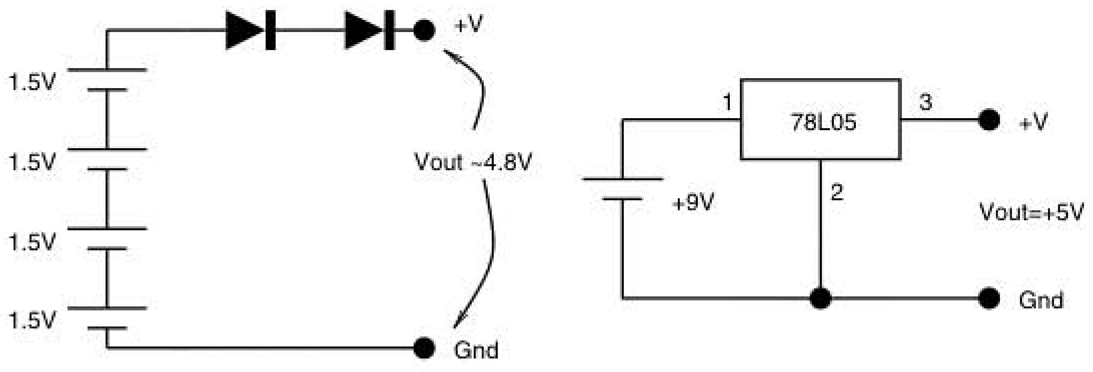

FIGURE 10.10 **Possible sources of supply voltage for the dice circuit.**

I measured the current consumption of the device when in use and when the device was put in powered-down sleep mode.

VCC(V)	ICC(MA)	STATE
+5	28.5	All 7 LEDs lit
+5	1 μA	Power-down sleep with all LEDs off.

However, the key to minimizing the current consumption during the power-down sleep mode is not merely activating the power-down sleep mode in software, as I found out after some efforts. The AVR ports (PORTB and PORTD in our case) if configured in high-Z floating state or as inputs, consume quite some current (about 600 μA for both the ports) even if the device is put in power-down sleep mode. To minimize this current consumption, I had to configure both the ports as outputs and I set the state of all the port bits to "1", which resulted in a 1-μA current consumption in power-down sleep mode, which is really amazing.

10.11 Adapting the Circuit to an AT90S2343

We have used a 20-pin AT90S1200 processor for this project. The project actually requires only four I/O pins—three for LED outputs and one for the sample switch input. This can be easily provided by an 8-pin AT90S2343 controller. However, the controller would need to be programmed with its internal oscillator. Any AT90S2343 could be used by programming it appropriately with the help of a parallel programmer (the serial programming methods cannot change the internal oscillator fuse bit).

The advantage of using an AT90S2343 in the internal RC oscillator enabled mode is that of reduced board space (an 8-pin DIP as opposed to a 20-pin DIP package) and component count, as it would not need the crystal and associated capacitors.

Figure 10.11 illustrates the block diagram of the dice circuit using an AT90S2343. Porting the circuit would need modifications to the software, too. The sample switch is

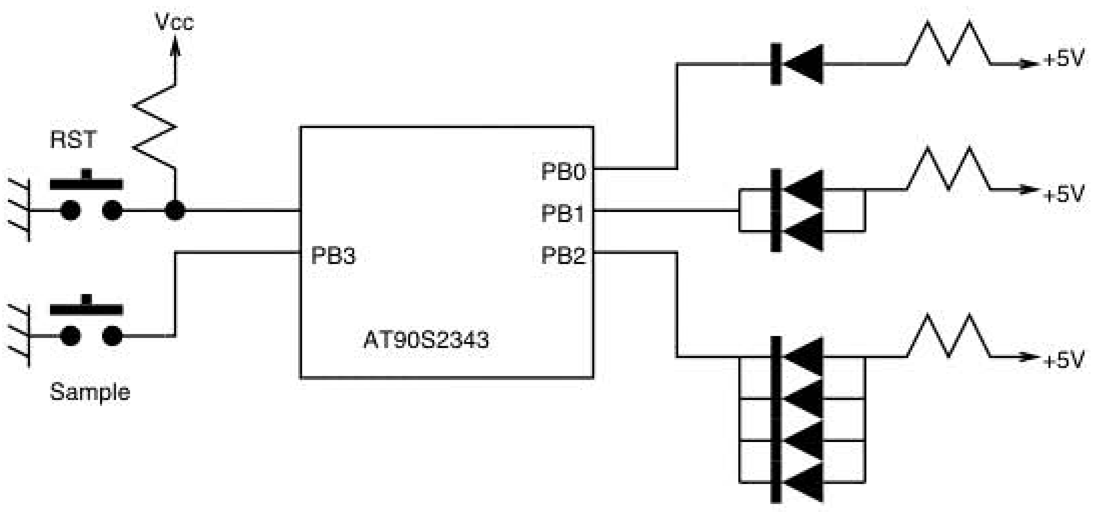

FIGURE 10.11 **Block diagram for the electronic dice using an AT90S2343.**

now connected to the PB3 and not PD0, as the 2343 does not have a PORTD. Secondly, the software would need to initialize the stack pointer appropriately. The AT90S1200 processor has a hardware stack that allows up to three nested subroutines. The processors in the AVR family other than the AT90S1200 use a software stack (in the internal RAM area), and so the stack pointer needs to be initialized. Once these changes are implemented, an AT90S2343 could well be used instead of the AT90S1200.

This page intentionally left blank.