

The High-Roller — PIC-based Electronic Dice



Stack the odds in your favor with this MCU-based project

This Month's Projects

- High-Roller Dice .. 42
- Brake Flasher 48
- Small-Wart 52



The Fuzzball Rating System

To find out the level of difficulty for each of these projects, turn to Fuzzball for the answers.

The scale is from 1-4, with four Fuzzballs being the more difficult or advanced projects. Just look for the Fuzzballs in the opening header.

You'll also find information included with each article on any special tools or skills you'll need to complete the project.

Let the soldering begin!

Introduction

Having been professionally involved in circuit design many years ago, I recently found myself longing for the unique odor of smoking flux and the burnt fingertips gained while soldering a small component into a circuit. A quick scout around the web showed that many people are now using microcontrollers (MCUs) to implement logic, and that the development tools needed are often free or relatively cheap. Since I also enjoy low-level programming, this was too good to pass up, and I resolved to dip my toe back in the water with a small MCU-based project.

After further research, I decided to use a Microchip (www.microchip.com) PIC MCU. These are very popular, and the company provides lots of information and tools such as the

MPLAB assembler/simulator environment, all available at no cost from their website. If you've not used a Microchip MCU before, writing code for them is very easy. Devices such as the PIC16F84A are used in countless projects that can be found all over the web. A user forum called the PICList (www.piclist.com) is another useful resource if you need some guidance or help.

Not wanting to create yet another electronic thermometer, I decided to design an electronic dice roller, which I call the High-Roller. Nostalgia played a part in this decision, since a 7400-series-based die was one of the projects I built when first getting involved in electronics. While choosing an MCU, I was particularly attracted to the PIC12C508A, because it packs a lot in a very small chip.

The '508A is an eight-pin MCU with 512 words of program memory, seven control registers, and 25 registers for program use. Its instruction set comprises just 33 different op-codes. Six of the '508A's pins can be configured as I/O lines if your design is able to use the chip's internal 4MHz oscillator instead of a more stable external crystal or resonator. This allowed me to keep the circuitry very simple (see Figure 1), and focus on becoming familiar with the programming of the MCU.

One drawback of the '508A is that it is a one-time programmable (OTP) chip, rather than being flashable

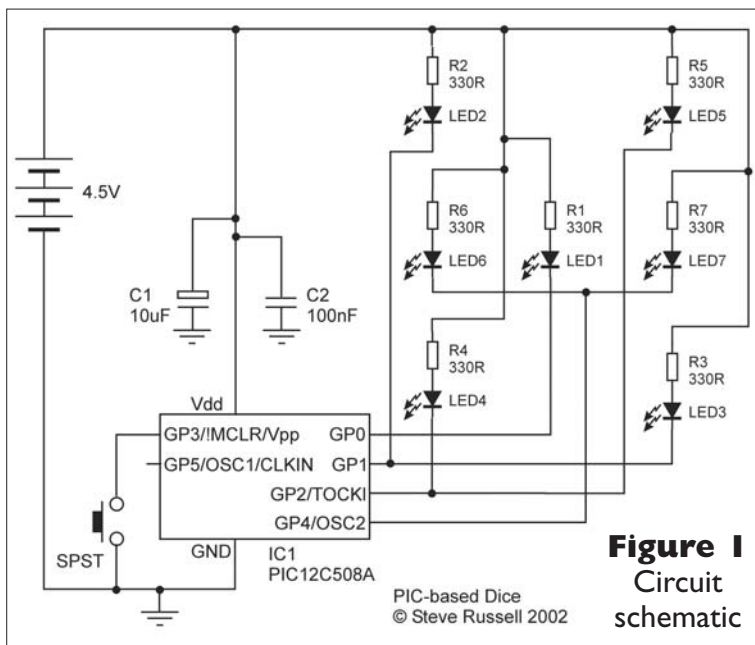
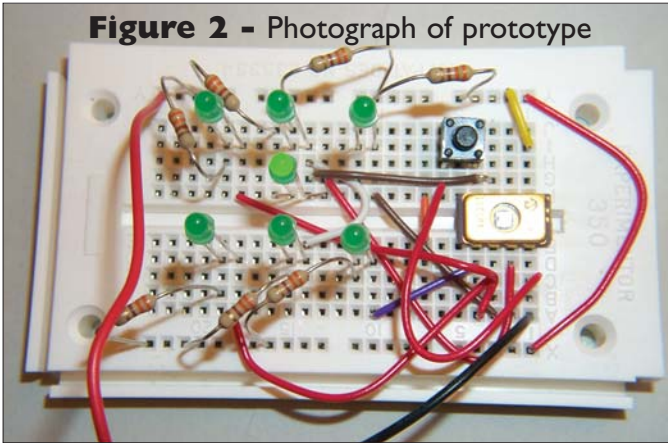


Figure 1
Circuit schematic

PIC-based Dice
© Steve Russell 2002

Figure 2 - Photograph of prototype



like the 'F84A. If there's a bug in your program, you need a new chip! For development purposes, therefore, Microchip offers a UV-erasable part, which I used to debug the High-Roller, switching to an OTP part for final construction.

Two other important limitations of the '508A are limited stack size (two words), and no support for interrupts, but I saw these as a challenge!

User interface

For output, I decided to use the time-honored seven-LED way of representing numbers on a die, shown in the prototype in Figure 2. By switching on the appropriate LEDs, the dot pattern for each value from one to six is displayed. My project from all those years ago rolled a single die, but a programmable design can easily be made to roll multiple dice. The High-Roller allows up to seven dice to be rolled simultaneously.

You press the single button to roll the dice, indicated by flashing the display between "3" and "reverse 3." The device has to be responsive, that is, the dice should roll as soon as you press the button, and this is covered in some detail in the code description. After rolling, the results are displayed by showing the values for individual dice sequentially. Note that two consecutive dice with the same value have to be easy to discern. This is achieved by fading the LEDs up from off to full brightness

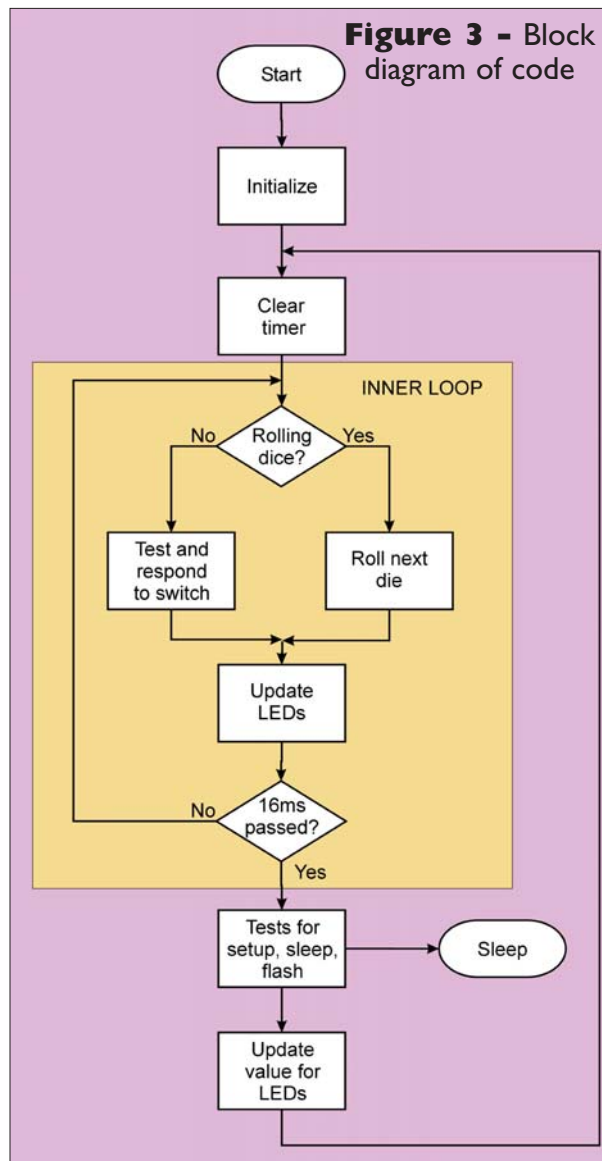
for each value to be displayed.

After displaying all of the dice, the LEDs are flashed between "2" and "reverse 2" and the dice are displayed again. Unless the button is pressed again, the High-Roller switches off after five such display sequences. This puts the '508A into its sleep mode, with all LEDs extinguished. Almost no current is drawn while asleep, so no power switch is required. You wake the High-Roller up by pressing the button, which displays the results of

the last roll again. This means that there should be no arguments about what was rolled!

Set-up mode – The final interface

Figure 3 - Block diagram of code



Tools for Construction

Construction is straightforward, just make sure you get the polarized components the right way round. Putting the LEDs in backwards will not cause a component failure, they just won't light up. Getting the tantalum capacitor reversed may cause it to fail, and putting the PIC in backwards will almost certainly blow it, and render it useless.

Note that you will need a device programmer to program the PIC12C508A. A quick Google search found many sites from which you can purchase relatively cheap programmers (<\$100.00, some much less). My own programmer is a Galep-4 from www.conitec.com, which is not cheap (at about \$300.00), but it programs a whole variety of other devices in addition to the smaller PIC MCUs. Its software is included in the purchase price, and is regularly updated to add new devices (currently about 2,000 are supported). I have no connection with Conitec other than as a satisfied customer.

Tools for Debug

No special tools are needed for debug. A multimeter is useful for checking voltages, but an oscilloscope is not needed (it is fun to watch the pulse width modulation used to control the brightness of the LEDs, though!). If the device doesn't work after you have built it, there are three main possible causes:

1. Power not applied.

When you first apply power, you should see the initial "double-1" roll. If not, check that the PIC is receiving the correct voltage on its power and ground pins. (You did insert it the correct way round, didn't you?)

2. A fault in construction or a faulty component.

Check that all connections are correct and look out for short-circuits and open-circuits. Check component polarity (polarized capacitors, LEDs, PIC), and that the switch pulls the PIC input pin to ground when depressed.

3. A faulty PIC12C508A.

Use the verify function of your programmer to make sure the device is correctly programmed. Make sure you double-check the polarity of the power supply before inserting the chip in the circuit. You may want to use a socket to make this easy to do. If you bought your PIC from me, it was verified and checked for correct operation before being shipped to you. Use proper anti-static handling techniques before touching the device.

```

=====
Example 1
;*****
; INNER LOOP
;
; This main loop is repeated over and over for about 16ms. Loop execution
; time creates latency in detecting the end of the 16ms period, so keep
; it as short as possible. TMR0 is clocked each 128us, so the path MUST
; be shorter than this to avoid missing timer comparisons.
;
; Cycle counts:
;
; 1. Longest path through loop code is 88us.
; 2. Shortest path through loop code is 24us.
;
; Worst case timing error in exiting loop is therefore 88/16384,
; or about 0.5%.
;*****
loop
    btfsc        dicestat,S_ROLL ; [1/2] Test if we're rolling the dice
    gotorolldie ; [79a max] YES: Roll die (skip switch testing)
    callswitch  ; [50b max] NO: Poll the switch

;*****
; Test if it's time to turn the LEDs on
display
    movfontime,w ; [+1] Get the time the LEDs should go on
    subwf        TMR0,w ; [+2] Subtract w from timer
    movlw        LEDOFF ; [+3] Assume we're switching the LEDs off
    btfsc        STATUS,C ; [+4/+5] Carry flag=1 if ontime<timer
    movfledval,w ; [+5] ...so time to switch LEDs on, get value
    movwf        GPIO ; [+6] Switch LEDs on/off as reqd.

;*****
; End of inner loop
loopend
    btfss        TMR0,7 ; [+7/+8] bit7=1 after 128 counts
    gotoloop ; [88a/59b] 128 * 128us => 16ms approx

```

decision was to allow the user to select how many dice are rolled. Holding the button pressed for about two seconds puts the High-Roller into set-up mode. In this mode, the number of dice is indicated by flashing that number of LEDs. Subsequent presses of the button let you select from one to seven dice. When the button remains unpressed for about two seconds, set-up mode is exited and the new number of dice are rolled and displayed.

When power is first applied, the High-Roller is initialized to roll two dice, both of which are set to "1."

Things to note about the PIC12C508A and other comments

- An individual oscillator calibration value is stored at address 0x1FF in each '508A during manufacture. If you wish, you can (as in the High-Roller) load this value into the OSCCAL register to ensure reasonable accuracy of the internal clock. For OTP parts, check your programmer information carefully to make sure you don't overwrite this value when programming. For UV-EPROM parts, be sure to read out the calibration value before erasing, as you

will need to program this value back in each time you re-program the device.

- Due to the way subroutine calls are implemented, subroutines must be located in the first 256 words of the 512 word program memory. This is why most programs you will see (including this one), jump to a location following the subroutines almost immediately after the start address of 0x00.

- If you want the '508A to wake from sleep on a pin change, Microchip says you have to read the inputs before going to sleep, otherwise this function may not work as expected. See Example 3.

- While developing the High-Roller, I wanted to be able to adjust the bright-

ness of the flashing LEDs (during a roll and at the end of a display sequence) without having to carefully check all through the code. A simple macro (Example 4) made this possible.

- To take the picture shown in Figure 2, I removed the light-proof cover from the window of the '508A (a thick piece of sticky card) so you can see the UV window. Interestingly, the flashgun caused the device to reset itself — you can see the first "1" being displayed. Microchip recommends that you cover the window, since light can clear internal registers, leading to unpredictable results and great difficulty in debugging your application.

Code description

A high-level block diagram of the code is shown in Figure 3. Let's take a look at the overall flow, examining the main areas in a little more detail as we go. Along with the excerpts included here, the full code is available from the *Nuts & Volts* FTP Library. As timing is crucial, many of the source code comments contain a number in square brackets, indicating cycle counts, which I used for calculating execution times.

When a reset occurs — due to either power being applied or a button press waking the device up — a '508A status bit informs you which of these it was, so you can tailor the code to handle the needs of each situation. Much of the initialization is common, and — most importantly — the code sets some configuration options using the OPTION register of the MCU.

In particular, weak pull-ups are applied to all inputs — this allowed me to leave GP5 floating, and avoids the need for a resistor to pull up the switch connected to GP3. The MCU is also set to wake up when an input pin changes state; its internal 4MHz clock is selected, and a 1:128 prescaler assigned to the timer, TMR0. The 4MHz clock means the MCU runs with a nominal 1us cycle time, since one cycle takes four clocks. A 1:128 prescaler for the timer sets the TMR0 register to increment every 128us,

setting bit 7 of TMR0 after a little more than 16ms, which, as you will see later, is an important part of the design.

The inner loop — After clearing TMR0, the code enters the inner loop, which is the heart of the program (see Example 1).

It is responsible for updating the display and checking for changes in the switch, and has to do so often enough that the user perceives that responses to the button are instantaneous. The '508A does not support interrupts, so the switch has to be polled.

A status byte (dicestat in the code) contains flags that control program operation. One of these flags (S_ROLL) indicates whether or not the dice are being rolled. If they are, execution is transferred to the rolldie subroutine (see Example 2) using a goto to help avoid stack problems and to allow the code to skip the call to switch by using goto display when returning from rolldie. Switch presses can safely be ignored while the dice are being rolled. While using gotos is often frowned upon in programming circles, they are sometimes necessary, and can be efficient when used carefully.

Rolldie rolls just one die each time it is called, making three calls to a linear feedback shift register (also in Example 2) to generate a pseudo-random number between 0 and 7. LFSRs have the advantage of being quicker to execute than many other common random number generators. Values 0 through 5 represent die values of 1 through 6, so values of 6 and 7 are discarded and the same die is rolled again next time around the inner loop.

Successful throws are saved in the appropriate variable using the indirect (FSR) register, and pointers are updated to roll the next die when rolldie is called again. When rolldie detects that the currently-selected number of dice have all been rolled, S_ROLL in dicestat is cleared to return to normal display mode.

Testing the switch — If the dice are not being rolled, then the High-Roller is going through the normal display sequence, and a call to switch is made. Switch is quite complicated since it has a lot to do. Its main role is to test for changes in the button status. If the user presses or releases the button, the code reads the value of TMR0 and saves it for later. On subsequent calls to switch, the button is tested for further changes. Each time a change is detected, TMR0 is again read and saved.

If no change is detected, TMR0 is checked to see if 10ms has elapsed (defined as DEBTIME) since the last transition. In this case, the new state of the switch is recorded in dicestat (S_PRESS flag). Note that all four possible transitions are decoded: released-released, released-pressed, pressed-released, and pressed-pressed. This avoids any problems due to switch bounce, very fast button presses, or transitions while S_ROLL is set, and also simplifies the wake-from-sleep code.

Action is only taken for released-pressed transitions, all others are discarded. When the High-Roller is in set-up mode, each press cycles the number of dice to be thrown from 1 to 7 and round again. More often, however, the High-Roller is displaying the previous throw and the dice are rolled. Rolling even seven dice takes place very quickly, usually less than a millisecond or so — exactly how long depends

```

=====
Example 2
;*****
; LFSR
;
; This subroutine uses a 16-bit linear feedback shift register to
; generate a pseudo-random sequence of bits. The feedback has been
; selected to give a maximal length sequence.
;
; INPUT:      FSR points to a register which will receive the next
;             random bit.
;
; OUTPUT:     The register pointed to by FSR is shifted left. Bit7
;             is shifted into the C flag, bit0 receives the new bit.
;
; Effect on shiftreg:
;   new bit15 = old bit15 XOR old bit14 XOR old bit12 XOR oldbit3
;   new bit14:0 = old bit15:1
;
; Cycle count: 18 (inc 2 for call)
;
;*****
lfsr clr feedback ; [1] Initialise feedback bit
movlw 1 ; [2] Bit 0 of W used to XOR
btfsc sr_hi,7 ; [3/4] xor in bit15
xorwf feedback,f ; [4]
btfsc sr_hi,6 ; [5/6] xor in bit14
xorwf feedback,f ; [6]
btfsc sr_hi,4 ; [7/8] xor in bit12
xorwf feedback,f ; [8]
btfsc sr_lo,3 ; [9/10] xor in bit3
xorwf feedback,f ; [10]
rrf feedback,f ; [11] Feedback bit to C flag, bit 7 is x
rrf sr_hi,f ; [12] Shift it into the high byte
rrf sr_lo,f ; [13] ..and the low byte
rlf INDF,f ; [14] Shift o/p bit into FSR->dice reg
retlw 0 ; [16]
;*****
; ROLLDIE
;
; Rolls a die pointed to by FSR.
;
; To roll a die, this routine calls lfsr three times to generate a number
; between 0 and 7. If a 6 or 7 is rolled, the FSR reg and rollctr are
; not adjusted, so that the same die is rolled next time. If the roll is
; good, then FSR is increased to point at the next die, and the roll
; counter (rollctr) is decreased. If rollctr gets to zero, all dice have
; been rolled, so we clear the S_ROLL flag and set variables for the
; next display sequence.
;
; INPUT:      FSR points to the die being rolled
;             rollctr has to be set to the total number of dice to roll
;
; OUTPUT:     FSR->die receives a number between 0 and 7
;             0 and 5, representing dice throws of 1 through 6.
;
; Cycle counts (inc 2 for call):
; 1. Roll 0-5, more dice to roll 65 cycles
; 2. Roll 0-5, last die 78 cycles
; 3. If the roll is a 6 or a 7 62 cycles
;*****
rolldie
clr INDF ; [1] Clear the die register
call lfsr ; [19] Shift in the first bit
call lfsr ; [37] ..and the second bit
call lfsr ; [55] ..and the third bit
movlw 6 ; [56] Maximum value we allow is 5
subwf INDF,w ; [57] ..so subtract 6 and test if -ve
btfsc STATUS,C ; [58/59] Test carry flag
gotodisplay ; [60] If set, rolled 6 or 7 so go round again
; Roll was good, so move to next die
incf FSR,f ; [60] Rolled 0-5 so bump FSR
decfsz rollctr,f ; [61/62] Set rollctr=rollctr-1
gotodisplay ; [63] Done? NO: Go round again if not done
;*****
; All dice have been rolled, set up for display
bcf dicestat,S_ROLL ; [63] YES: Clear the roll flag
call setdice ; [72] Reset pointer and counters
movlw SLEEPCNT ; [73] Get the number of display cycles
movwf dispctr ; [74] ...before sleep and set dispctr
gotodisplay ; [76]

```

```

=====
Example 3
; YES: We have displayed SLEEPcnt times... time to sleep
movlw    LEDOFF    ; We want LEDs to be switched off while asleep
movwf    GPIO      ; ...so set outputs to switch them off
movfGPIO,w    ; Read port before sleeping (as per Microchip)
sleep    ; ... and goodbye!
=====
Example 4
;*****
; MACROS
;*****
mSET_ONTIME    macro    val
    if val == 0
        clrfontime
    else
        movlw    val
        movwf    ontime
    endif
endm
=====

```

on how many invalid throws are generated. To give the user an indication that a roll has occurred, switch also sets S_FLASH to trigger the roll flash sequence, which lasts for about a second, after which the new results are displayed.

Driving the LEDs — The final part of the inner loop is the code to drive the LEDs. A variable (ontime) is tested against the current value of TMR0. While the timer is less than ontime, the LEDs are kept switched off. When the timer reaches a value greater than or equal to ontime, the LEDs are switched on by setting the GPIO port to the value held in ledval. Ledval and ontime are set in various places to control what is displayed on the LEDs. A subroutine (getgpio) is available to convert dice values to their corresponding port drive values.

The end of the inner loop has a test to see if bit 7 of TMR0 is set, which occurs some 16.4ms after the timer was cleared. If you recall, TMR0 is clocked every 128us, which is therefore the limit of the longest path through the inner loop code to avoid missing TMR0 values. The longest path is actually 88us (see Example 1), so the design has a comfortable margin.

Out of the loop — After about 16ms, execution leaves the inner loop and a series of tests are performed to determine whether:

- Two seconds have passed, and so a change to the set-up status is needed. If set-up mode is entered, ledval is set to the number of dice, and ontime and flashtmr are used to make the LEDs flash this value.
- The device is flashing the LEDs to indicate a roll or end-of-display sequence. If so, ontime sets the brightness, and flashtmr controls the period and duration of the flash sequence.
- To continue displaying the same die, ontime must be moved earlier to make the LEDs a little brighter the next time through the inner loop. The LEDs are kept off the first time through the loop, then ontime is reduced by about 0.25ms each iteration, until the LEDs are on for the

whole 16ms of the loop. This means that the LEDs display each die value for approximately $(16/0.25) * 16\text{ms}$, or a little over one second.

- Maximum brightness has been reached. If so, it is time to display the next die.
- Five display cycles have occurred and it is therefore time to switch off the High-Roller. If so, the LEDs are switched off and the sleep command puts the '508A into low-power standby mode. As I mentioned earlier, the High-Roller configures the chip to wake up when the button is pressed. Other triggers — such as a watchdog timer — are available, but not used in this design.

The tests are performed in this order, so that, for example, going into set-up mode takes precedence above all else. Depending upon the outcome of these tests, the values of status bits in dicestat, and setuptmr, ledval, flashtmr, ontime and other variables are set as necessary for the next pass through the inner loop.

At the end of this section of code, control returns back to the top, where the timer is reset and the inner loop is executed once again.

Debugging and design changes

Debugging a design based on the '508A is a challenge, particularly since I don't have access to any sophisticated hardware aids, such as an in-circuit debugger. However, careful use of the MPLAB simulator, coupled with close observation of the behavior of the buggy code, was sufficient to enable effective debug.

When I started this project, I had no feel for how much code would be required in the inner loop, so the TMR0 prescaler was initially set to 1:256. When I found I could perform the necessary function in less than 128us, I was able to change the prescaler to 1:128, which also reduced the total loop time from 32ms to 16ms.

Parts List

The parts are all very easily sourced from vendors such as Digi-Key, Newark Electronics, and Future Electronics. For development purposes, the UV-erasable (JW) version of the PIC12C508A is harder to find in stock, but both Digi-Key and Newark Electronics carry it.

R1-R7 — 330 ohm, 1/8W resistors, or use a seven-resistor R-pack

LED1-LED7 — General-purpose LEDs. I used 3mm, green LEDs

IC1 — PIC12C508A microcontroller

C1 — 10uF tantalum capacitor

C2 — 100nF capacitor

SPST — Single-pole, single-throw push to close switch

This also improved the display quality by eliminating a pronounced flicker. Making this change turned out to be fairly straightforward, and I believe this was primarily because I resisted the temptation to use numeric constants. Instead, I defined symbolic constants and variables. These, coupled with a logical structure, made the change simple to implement.

The last change I made was to put in an additional call to lfsr at the top of the outer loop, just before clearing the timer. While debugging, it was useful to have a repeatable series of "random" numbers following a power-on reset. Once the code was working, however, I wanted to eliminate this way of generating a repeatable series of rolls. Calling lfsr every 16ms makes the numbers generated depend on exactly when the button is pressed, which is essentially random when done by a human!

Conclusion

To be satisfied that the High-Roller was giving a fair roll to its dice, in the guise of an exercise in statistics, I convinced my children to tally some 1,200 rolls of the dice. The results were as follows (1-6): 202, 178, 205, 234, 180, 201. A colleague of mine did some analysis which indicated that the dice do, indeed, appear to be fair.

To make the High-Roller usable, I packaged the electronics and a battery-holder for three AAA cells in a small candy container, and designed a suitable graphic to go on the front of the unit. You can see the finished device in Figure 4 — the push button is between the two words on the graphic.

I had great fun designing and debugging the High-Roller, and learned a lot about the MCU. Now I have to think up a new project (I'm determined not to design a thermometer!). I hope you enjoy either making your own High-Roller or get interested enough to design your own devices, too. **NV**

About the author

Steve Russell started out his working life as a hardware design engineer. Somewhere along the way, he was seduced by computers, and began designing subsystems of various types for them, working for a multi-national computer company. Much later, after a spell in development management, he decided that salespeople needed his help and so he moved into technical marketing. He still misses the baleful green glow of an oscilloscope in a darkened lab, late at night ...

Contact Steve at pic.projects@ntlworld.com

References

Microchip: www.microchip.com

PICList: www.piclist.com

Linear feedback shift registers:

www.xilinx.com/xapp/xapp052.pdf

Switch bounce: http://dbserv.maxim-ic.com/appnotes.cfm/appnote_number/287



AC Voltage Tester

Non-contact test. Voltage Sensitivity: 100 ~ 240VAC. Detection Distance: <5mm. Over Voltage: 600V. Battery: 2AAA. Wt.: 40g.

AC110VD \$6.99

25W HiFi Audio Amp Kit

All the essential components to build a hifi amplifier module using the LM1875 IC. Heatsink, speakers and cables are to be supplied by the user.

Kit 50 \$12.49

300Ω 100W Rheostat

6" diameter. Without knob.

100R300 \$35.00

30VDC 400mA Wall Wart

Center negative. 3mm X 6.5mm plug.

99E005 \$7.95

CCD Color Video Camera

White plastic case, approx. 4.5"L x 2.5"W x 1.5"D. NTSC standard output. Runs on 5VDC @ 100mA for independent use. Cable is included for use as power adapter for use with PCs and video capture card.

99V006S \$39.95

Famous Brand Solder

1-lb spool of 60/40 rosin core solder.

99Z002 \$9.95

Hardware Grab Bag

Ten-pound assortment may include nuts, bolts, screws, brackets, springs, standoffs, arms, pawls, gears, pulleys and lots of indescribable metal stuff from Silicon Valley surplus to complement anyone's hardware hoard. Ten

92Z038 \$10.00



UHF Linear Amp

Input: 1 dB. Freq. Range: 890 to 940 MHz. Gain: 17 dB. Bias: 18 VDC @ 100 mA. Wilmanco #730.

96V001 \$14.95

Mini Project Box

2" x 2.75" x 0.625"; with 0.125" hole in both ends.

96U010 12/\$5.00

Cavro Stepper Motor Driver

Will drive two bipolar stepper motors up to 2A per phase. Board has the several useful ICs and other parts. Approx. 1 1/2"W x 5"L board with DB-15M connector. Sorry, no docs.

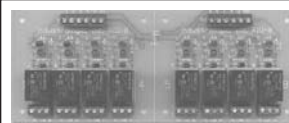
99C001 \$14.95

alltronics.com

Orders: P. O. Box 730, Morgan Hill, CA 95038-0730
Phone: (408) 847-0033 • Fax: (408) 847-0133
Download our Catalog: <http://www.alltronics.com>

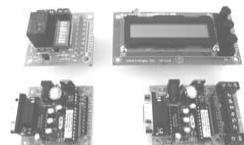
For more information on products, please visit our web site. Established 1978.
Shipping Additional on All Orders. Prices Good 60 Days from Date of Publication.

Circle #63 on the Reader Service Card.



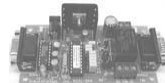
RIO-8-LL and RIO-8-OC \$59

Eight 10 amp relays for your micro. Logic level or open-collector drive. Can be split into two 4 relay boards!



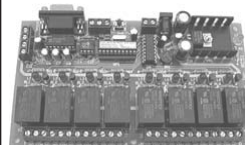
T51 \$49

Atmel 89C2051, 9 digital I/O, on-board Tiny Machine Basic, serial EEPROM, plus plug-in boards.



RS51-SPR \$49

Microcontroller based addressable and chainable 10 amp RS-232 relay.



RC51 \$129

Atmel 89C2051, serial EEPROM, eight 10 amp relays, 4 digital I/O, Tiny Machine Basic - a complete relay control system.

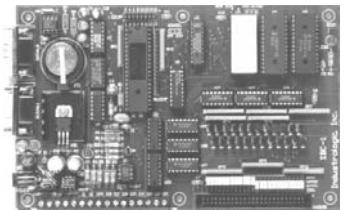
IndustroLogic®

3201 Highgate St. Charles, MO 63301 USA
(636) 723-4000 (800) 435-1975

www.industrologic.com

The Industrologic Advantage:

Power supply, cable, and software included with most products - no hidden charges for "accessories"!



SBC-1 \$199 (With 12 bit A/D - \$219)

Intel 80C51, EEPROM, 8 bit A/D, RAM, real time clock, 50 digital I/O, industrial I/O connectors, on-board Tiny Basic.

Featuring products that can be used as RS-232 serial data acquisition boards or as standalone industrial controllers!

Complete control systems for the price of an SBC.



TC51 \$129

Atmel 89C4051, serial EEPROM, two 12 bit A/D, four digital I/O, four 10 amp relays, plus on-board Tiny Machine Basic with LCD support - a full featured control system.

Circle #64 on the Reader Service Card.