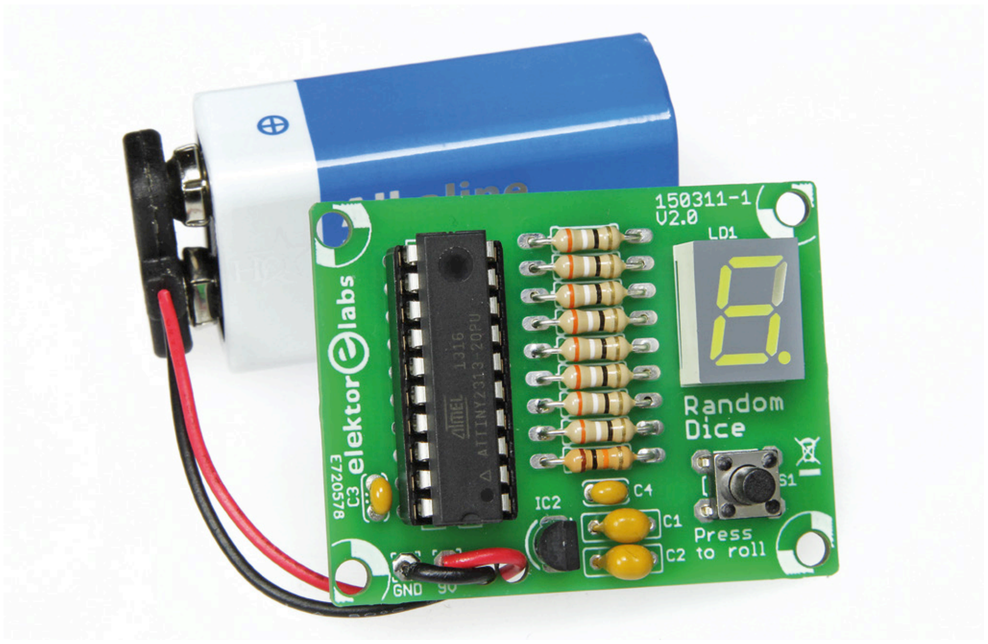
Tiny-Dice

Electronic dice using an ATtiny2313



By **Florian Schäffer** (Germany)

This simple electronic dice is an ideal starter project to introduce youngsters and those still young at heart to the dark arts of microcontrollers and circuit building. We take you step by step through the process. For your troubles you will build a useful electronic dice. It will put an end to you scrabbling around under the table to retrieve a dice that's been carelessly tossed and also an end to cheating… It landed on a six! It really did… yeah right.

This simple circuit simulates the rolling of a dice (or 'die' to pedants). When the roll button is pressed the display shows random numbers in the range of one to six, after a while the display settles with one number on the LED display.
To make the project easier to build for beginners we have ruled out the use of any SMD components and used leaded components throughout.

The dice circuit

The circuit in **Figure 1** shows the connections between the pushbutton, the seven-segment display and the Atmel microcontroller I/O pins. Resistors R1 to R8 are required to limit the current flowing to each of the LED display segments. IC2 is a simple regulator which converts the 9 V battery voltage to a stable 3 V for the circuit. C4 is used to suppress the signal produced by contact bounce from pushbutton S1. This button is used to start the dice rolling. When the circuit has not been used for a while the display is turned off to reduce power consumption and promote longer battery life. The decimal point LED is used to indicate that the circuit is on.
The complete dice kit is available from the Elektor Store [1]. Included in the kit is a pre-programmed microcontroller. The microcontroller's internal R/C oscillator generates an 8-MHz clock signal. The internal 1:8 divider is used to produce the 1-MHz controller clock frequency. Accurate timing is not essential in this application so an external crystal is not necessary. R9 is used as a pull-up resistor to

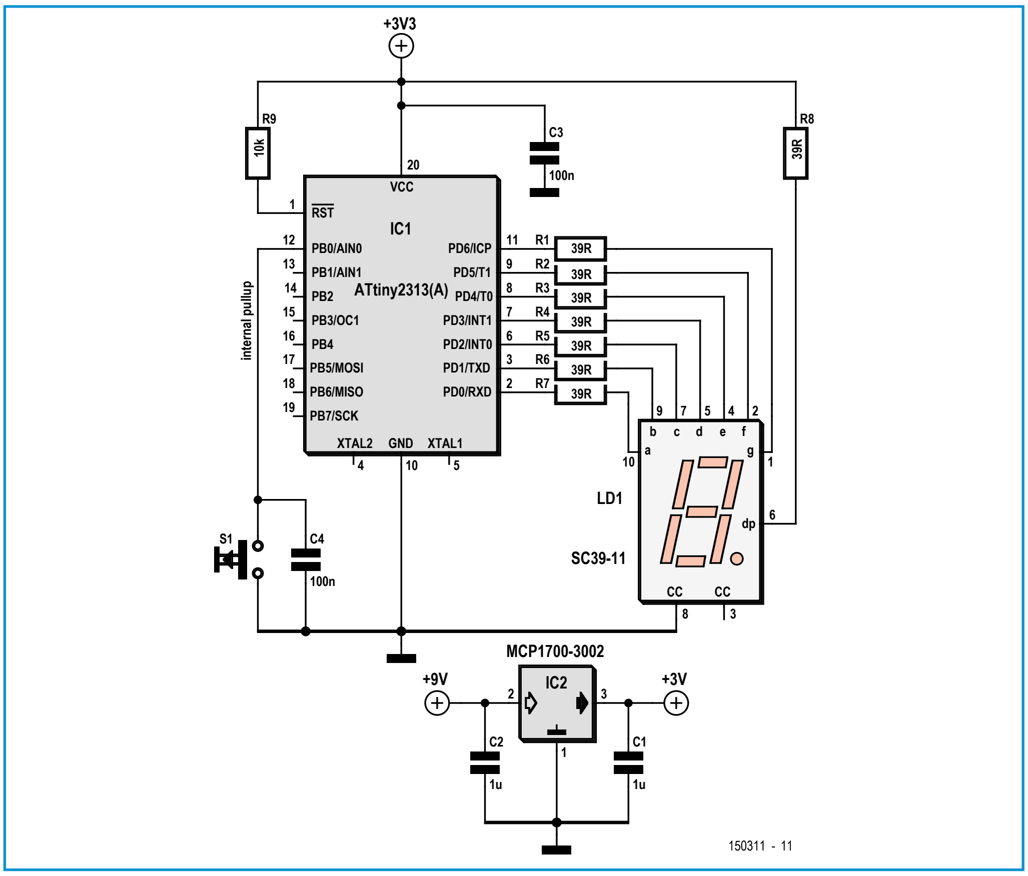


Figure 1. The circuit using an Atmel microcontroller looks… dicey.

tie the high-impedance Reset input up to a logic High (i.e. to disable it).

The source code

The dice program shown in **Listing 1** has been produced using the free WinAVR tool chain [2]. The GCC compiler contained in the tool chain generated the machine code for the ATtiny2313A controller. The basic principle of operation is quite simple; the microcontroller does not have a built-in random number generator so we use an 8-bit timer as a counter which is configured to continually count up to a maximum, overflow to zero and count up again. The counter overflows about fifteen times per second.

Each press of the pushbutton generates two interrupts (one on pressing and one on releasing). At every second interrupt a loop is initiated which shows on the display (with ever increasing delay) numbers from 1 to 6 to simulate a dice rolling to a halt. Once this loop is executed the value of the counter which is running continually in the background is read and a value from 0 to 6 is calculated (the remainder value after dividing by 6, plus 1). The resulting value is then displayed. Unless you can accurately synchronize your buttoning pushing with the internal counter clock, each number from one to six has an equal probability of being displayed and can be considered random.

The displayed LED segments for the numbers are represented in the code (0 to 9, only 1 to 6 are used). The program can of course be changed to show other numbers or even letters on the display.

Programming

As we already mentioned, the kit for this project includes a pre-programmed microcontroller. There is no reason why you shouldn't change the program and re-program the chip, after all that's half the fun of hacking. Once any changes have been made or new code has been created using an editor program, it will need to be converted into machine code using a compiler. The resulting hex file can then be transferred to the microcontroller's flash memory using a program such as AVR-DUDE [3]. Here you will need an extra bit of hardware. In addition to a programmer

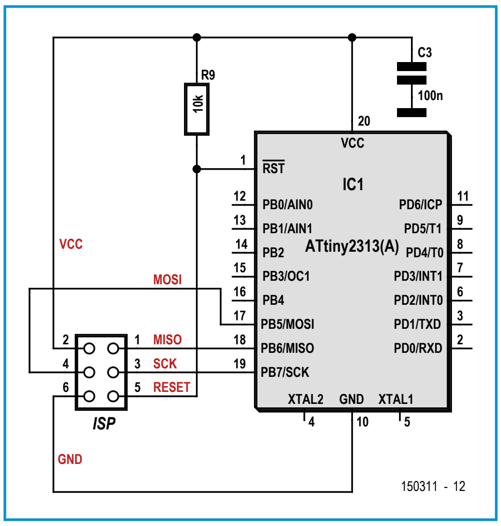


Figure 2. The small programming adapter with a six pin ISP connector.

such as the Atmel AVR-ISP MK2 you will need a programming adapter with an IC socket; it can be easily built using a small square of perf board (**Figure 2**). This handy bit of hardware will also be useful in the future for reprogramming the microcontroller if you need to make changes. Another aspect of microcontroller programming is setting the internal fuses which are used to configure the microcontroller (i.e. to enable operation with the internal oscillator for example). This project just uses the default settings so it is not necessary to worry about fuse settings at the moment.

Construction

Soldering: Should you be lucky enough to have a soldering iron with an adjustable bit temperature set it to around 350 to 370 °C for classic rosin-cored, lead/tin solder which produces reliable joints on DIY projects. Lead-free solder will need a temperature of around 380 to 400 °C.

1. Insert the components into the PCB from the component side. See the placement plan near the parts list.
2. Make good contact with the soldering tip onto the solder pad and the component lead.
3. After about half a second introduce the solder so that it contacts the pad and lead.
4. When the solder melts and flows (should take around one second) remove the solder and then the iron from the joint.
5. Check the joint is good.
6. Clip off protruding component lead.

Component mounting sequence: Component leads can be bent using flat nose pliers. Always grip the lead on the component side of the bend. Don't make the bend too close to the component body.

1. Resistors (arrange them so that the color rings indicating their value can be read from left to right).
2. Capacitors (check their values).
3. Pushbutton.
4. The IC socket (line up pin one with position 1 on the layout).
5. The seven segment display (check orienta-

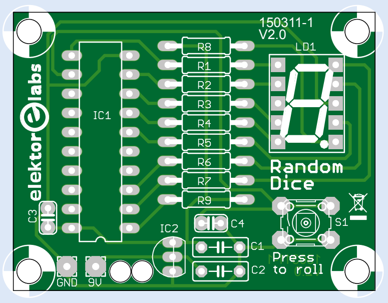
Component List

Resistors
R1–R8 = 39Ω 5%, 0.25W
R9 = 10kΩ

Capacitors
C1,C2 = 1µF, 5mm pitch
C3,C4 = 100n, ceramic, 0.1'' pitch

Semiconductors
IC1= ATtiny2313A, programmed
IC2 = MCP1700
LD1 = SC39 7-segment LED

Miscellaneous
S1 = pushbutton, momentary action
Clip-on connector for 9-V battery
201-way IC socket.
PCB # 150311-1 v1.0



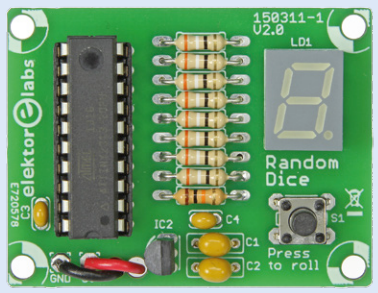


Figure 3. How it should look when you're finished.

tion of the decimal point lead).

6. IC 2 (make sure it's correctly orientated). Don't mount this too close to the board. Leave a gap of about 5 mm between the component base and the board surface. A component mounted close to the board can be subjected to relatively large levels of mechanical stress which may cause internal damage to the IC.
7. The battery clip (feed the wires through the strain relief holes first. Black wire to the GND pad and red wire the 9 V pad).

Final assembly: After assembling all the components on the board make a close inspection of all the joints. Lead/tin solder produces shiny joints; any crazing is an indicator of a dry joint, check for missing solder or solder bridges shorting two tracks together. It's also worth double checking battery connection polarity and the IC socket orientation. The finished board should look like **Figure 3**. Now you can connect up the battery but don't plug in IC1 just yet. The decimal point LED will light up to show that power is available to the board. Should the LED not light then you can begin with what is any engineer's favorite pastime — hardware debugging. Once everything is okay you can carry on:
Disconnect the battery.
Orientate the pre-programmed IC1 over its socket ensuring that the notch in the package indicating pin #1 is nearest to the battery wire connection point and not the other way round. Check that the package leads line up with all their positions in the socket. Apply light pressure to correct the position of any lead. Now carefully apply pressure evenly to the top of the package using two fingers until the IC is inserted. This may require some force.
Reconnect the battery.
Press the pushbutton and start gambling!

Now you're ready to roll! *Have fun!*

(150311)

Web Links

[1] Web page for this article:
www.elektormagazine.com/articles

[2] WinAVR: http://sourceforge.net/projects/winavr/

[3] AVRDUDE: www.nongnu.org/avrdude/

Listing 1.

```
#include <avr/io.h>
#include <util/delay.h>       // defined _delay_ms()
#include <avr/interrupt.h>    // IRQ handling

int main (void);

volatile uint8_t roll=0;
volatile uint16_t zeit=0;
const int8_t numbers [10] =   // 0..9: Binary association of ports with numbers. Active high
{
/*       A
      F   B
        G
      E   C
        D         */
   0b00111111,    // 0
   0b00000110,    // 1
   0b01011011,    // 2
   0b01001111,    // 3
   0b01100110,    // 4
   0b01101101,    // 5
   0b01111101,    // 6
   0b00000111,    // 7
   0b01111111,    // 8
   0b01100111,    // 9
};

/**
    @brief   IRQ Routine PCINT is called at IRQ. Allows the dice to roll
*/
ISR (PCINT_B_vect)
{
   uint8_t i=0;
   roll++;        // How often has IRQ been triggered? PCINT only recognizes a toggle
                  // IRQ is serviced twice per press.
                  // Only need to start rolling once for every two
   if (roll == 2)
   {
      roll=0;                            // Start from zero
      for (i=1; i < 40; i++)             // Simulates dice throwing
      {
            PORTD = numbers[(i % 6)+1];  // Output. Counter Modulo 6 = 0-5  => +1 = 1-6
            _delay_ms(i*3);
      }
      PORTD = numbers[(TCNT0 % 6)+1];    // Output. Timer value Modulo 6 = 0-5  => +1 = 1-6
      zeit=0;                            // counter to reset LEDs (zeit = time)
   }
}

/**
   @brief   Main routine
   @param   none
   @return  End-Status
*/
int main(void)
{
   PORTD = 0;              // PORTD all off
   DDRD = 0xFF;            // PORTD defined as outputs
   DDRB &= ~(1 << DDB0);   // B0 input
   PORTB |= (1 << PB0);    // Pull Up active

   TCCR0B = (1 << CS02) | (1 << CS00); // 8 Bit Timer, Prescaler CLK/1024 => 1,000,000/256 = 3.9 kHz
                                       //  => 0.000256 s/pulse => x256 > overflow occurs every
0.065s
   GIMSK |= (1 << PCIE);      // PCIE IRQs enabled
   PCMSK |= (1 << PCINT0);    // Assign IRQ to PB0
   sei();                     // IRQs enabled

   while (1)  // endless
   {
      // A loop to turn off the display after x secs
      for (zeit=0; zeit <=300; zeit++) // 300x100=30.000ms = 30s
         _delay_ms(100);

      PORTD = 0;  // All segments off.
   }
   return 1;      // never
}
```