Fun with LEDs and Microcontrolers

Christian Ammann

October 29, 2014

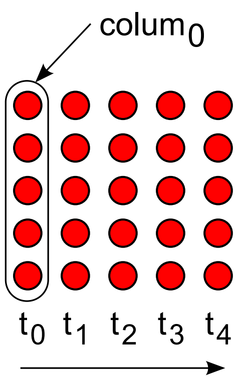

Figure 1: Controlling a 5x5 LED matrix with 10 pins

1 Introduction

This paper describes how to assemble and control a two-dimensional LED (light-emitting diode) matrix. LED matrices and cubes are a very popular form of digital art and usually come together with a microcontroller. I always wanted to create my own LED cube but due to my lack of knowledge in this domain, i started with a simple 5x5 two-dimensional LED matrix. This project was finished sucessfully and its results are described within this paper.

Nowadays, many embedded devices contain microcontrollers which makes knowledge about them more and more important for IT security research. Therefore, the aim of this paper is to give a basic introduction into circuit design, microcontrolers and embedded programming using a LED matrix as an example. Its structure is straight forward: Section 2 presents the corresponding hardware aspect. It contains a schematic for a simple 5x5 LED matrix and describes its connections to an Atmega8 microcontroller [1]. The ATmega8 is a cheap 8-bit device and developed by the Atmel Coperation [2]. They also provide a free C compiler. Therefore, section 3 presents the necessary C source code for controling the LED matrix. It consists of a driver and some basic animations.

2 Circuit Board

This section describes the layout of a 5x5 LED matrix. Although 5x5 is still very small, it already consists of 25 LEDs. Turning them on and off at the same time makes it necessary to connect them with 25 free pins of the microcontroller. This becomes problematic when building larger matrices (e.g. 10x10) because the total amount of free microcontroller pins is exceeded.

This problem can be solved by exploiting our brains image processing ability. Figure 1 visualizes the approach. It shows a 5x5 matrix with 25 red LEDs which can be divided into five colums ($colum_0$, $colum_1$, ... $colum_4$). Human eyes are capable of distinguishing between 25 different frames per second. Therefore, imagine the following situation: At t_0, the LEDs of $colum_0$ are turned on. After 0.04 seconds at t_1, the LEDs of $colum_0$ are turned off and instead $colum_1$ is activated. Again, we wait for 0.04 seconds and turn on $colum_2$. This step is repeated in an unfinite loop (when $colum_4$ is turned off, $colum_0$ is turned on again). In one second, the human eye would recognize 25 different colums beeing activated and deactivated.

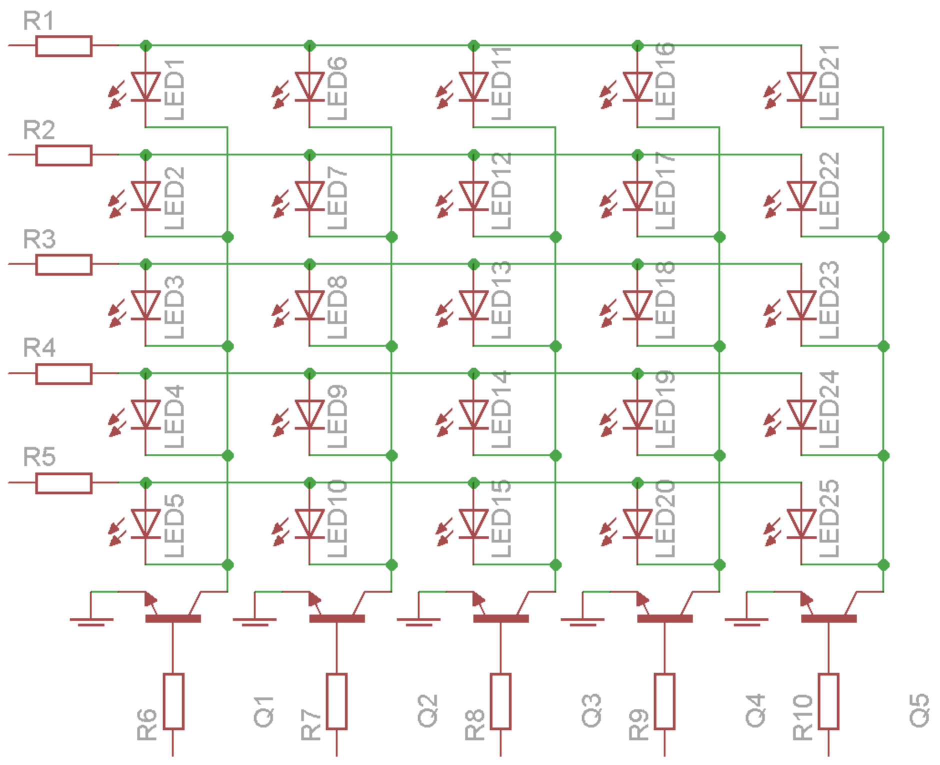

Figure 2: Schematic for a 5x5 LED matrix

If we increase colum change speed by four (t_y - t_x == 0.01s) 100 colums per second are being turned on and off. This is too fast for the human eye and makes us think 25 LEDs ($colum_0$ - $colum_4$) are turned on at the same time. Therefore, if our microcontroller activates and deactivates the corresponding colums fast enough, all we need are 10 pins for accessing a 5x5 matrix:

- Five pins control the LEDs of a colum.
- Five pins are used to select a colum.

Figure 2 shows the schematic of a 5x5 LED matrix. It consists of 5 transistors, 10 resistors and 25 red LEDs. The pins of R_1 - R_{10} are connected to an Atmega8. R_1 - R_5 are used for controlling five different LEDs while R_6 - R_{10} represent the pins for colum selection. The transistors Q_1 - Q_5 act as a kind of switch and connect the LEDs of a colum with ground. The following example demonstrates the approach: For turning on the upper-left LED, a current must flow from R_1 to ground. A connection to ground can only be established if a voltage is applied to the base of Q_1 at R_6.

The schematic in figure 2 is designed for a Vcc of 5V. The LEDs and transistors can not handle 5V directly and therefore are protected by resistors:

- R_1 - R_5 have a value of 120 Ω. The resistor value for a LED with a certain color and certain Vcc can be achieved by calculators available on the net.
- R_6 - R_{10} have a value of 2400 Ω. The precise value can be found in the corresponding transistor data sheets.
- Q_1 - Q_5 are BC238 amplifier transistors. Other NPN transistors like BC548 can be used as well.

According to figure 2, ten microcontroller pins are connected with the LED matrix. My design uses the following ones:

LED Matrix Pin	Atmega8 Pin
$Colum_0$	PD2
$Colum_1$	PB1
$Colum_2$	PD1
$Colum_3$	PB0
$Colum_4$	PD0
LED_0	PB4
LED_1	PD4
LED_2	PB3
LED_3	PD3
LED_4	PB2

The next section presents a software software design which is executed on an Atmega8 and performs some basic animations on the LED matrix.

3 Software

The software which controls the Atmega8 is written in pure C [3]. It consists of two parts:

- A LED matrix driver which is implemented as a timer interrupt. It also provides a 5x5 byte array which acts as an interface between driver and user land. Whenever a byte is set to 1, the corresponding LED is turned on.

- The user space which runs in an unfinite loop and displays some animations on the LED matrix due to writing into the 5x5 byte array of the driver.

The following two subsections describe driver and user space in more detail.

3.1 LED Matrix Driver

The LED matrix driver is implemented as a timer interrupt. Its main task is to activate the different colums of the LED matrix in an unfinite loop. It provides a 5x5 byte array called *videobuffer* as a user interface. If the statement *videobuffer[0][0]=1* is executed, the matrix driver activates the LED in the upper left corner of the matrix. According to this, *videobuffer[4][4]=1* controls the LED on the bottom at the right side. This implementation has one advantage: A developer doesn't have to be familiar with hardware aspects like the used GPIOs or precise timing. Instead, he can access an array and use it to control the LED matrix. This leads to the following timer interrupt implementation:

1. Initialize timer and a colum counter.
2. Deactivate the LEDs of the previously displayed colum.
3. Activate the LEDs of the current colum acccording to the bytes in the *videobuffer*.
4. Increment colum counter. Assign a 0 if $counter > 4$ becomes true.
5. Goto 2.

The following C source code demonstrates the algorithm:

Listing 1: Timer Interrupt Initiatisation for a LED matrix

```
1  volatile unsigned char videobuffer[ROWS][COLUMS];
2  volatile unsigned char videobufferCurrent[ROWS][COLUMS];
3  volatile unsigned char columCounter, vSync;
4
5  int main(void) {
6    TCCR0 = (1 << CS01);
7    TIMSK = (1 << TOIE0);
8    sei();
9  }
10
11 ISR(TIMER0_OVF_vect){
12   // ...
13 }
```

Listing 1 begins with the declaration of some important variables. The whole LED driver uses a double buffering technique. Therefore, two *videobuffer* arrays are declared. Line

1 contains an array which can be read or written by user space code. *VideobufferCurrent* in line 2 is read by the timer interrupt to activate the corresponding LEDs in the LED matrix. Whenever the fifth colum of the LED matrix has been displayed, the content of *videobuffer* is copied into *videobufferCurrent*. This avoids inconsistency when a user tries to write into a buffer while its read by the driver.

The variable declarations are followed by the the timer initialisation. It is shown in listing 1 between line 5 and line 9. I use in $timer_0$ which is an 8 bit timer. First, *CS01* is set to *true* in the timer control register which tells the microcontroller to use 8 as a prescaler value. This slows down the timer because it is only incremented after 8 clock cycles. Afterwards, line 7 unmasks the interrupt of $timer_0$. This leads to an execution of an interrupt service routine whenever an overflow of $timer_0$ occurs. Finally, the *sei* instruction enables the interrupts of the microcontroller. The following source code demonstrates the implementation details of the interrupt service routine:

Listing 2: Timer Interrupt Implementation for a LED matrix

```
1  ISR(TIMER0_OVF_vect){
2    unsigned char i=0;
3    unsigned char j=0;
4
5    setColum(columCounter);
6    columCounter++;
7
8    if(columCounter==COLUMS){
9      columCounter=0;
10     vSync=1;
11
12     for(i=0; i<ROWS; i++){
13       for(j=0; j<COLUMS; j++){
14         videobufferCurrent[i][j]=videobuffer[i][j];
15       }
16     }
17   }
18 }
```

Listing 2 contains an interrupt service routine which is executed automatically whenever an overflow in $timer_0$ occurs. It uses the method *setColum()* to activate the LEDs of a certain colum. Therefore, *setColum()* reads *videobufferCurrent* and activates the corresponding microcontroller output pins like PD0, PD1, etc. Afterwards, the colum counter is increased.

When the fifth colum is displayed, the content of *videobuffer* is copied to *videobufferCurrent*. Furthermore, the colum counter variable is set to 0. An import aspect is shown in line 10: When the fith colum is displayed, the interrupt service routine sets *vSync* to *true*. This variable is necessary for vertical synchronisation with the the user land and is used in the following method:

Listing 3: Vertical Synchronisation

```
1  void waitForFrames(unsigned char frames){
2    while(frames!=0){
3      while(vSync==0){
4        sleep_enable();
5        sleep_cpu();
6        sleep_disable();
7      }
8      vSync=0;
9      syncs--;
10   }
11 }
```

The method *waitForFrames()* returns, when a certain amount of frames was displayed on the LED matrix by the timer interrupt. Each frame consists of five columns. When the last column is activated on the LED matrix, *vSync* is set to *true* (see listing 2). Therefore, *vSync* can be used to check whether a complete frame was successfully displayed. Polling *vSync* in a loop increases power consumption. *WaitForFrames()* avoids this and makes use of *sleep* instructions (see line 4 - 6 in listing 3). This leads to the following behaviour: *WaitForFrames()* checks whether *vSync* has become *true*. If *vSync* is *false*, the CPU is set to sleep mode until the next interrupt occurs. Afterwards, *vSync* is checked again. If *vSync* is *true*, the timer interrupt has displayed a complete frame on the LED matrix. Therefore, *vSync* is set to *false* and *waitForFrames()* waits for the next frame.

3.2 User Land

The previous chaper presents a driver which acts as an abstraction layer between the real hardware and the user land. It can be used the following way:

1. Turn some LEDs on or off. This can be done writing into *videobuffer*.
2. Call *waitForFrames()* which waits until the current frame is drawn, copies *videobuffer* into *videobufferCurrent* and draws its content.

The following basic animation demonstrates this approach. We want to run a program on our microcontroler which activates $led_{0,0}$, $led_{0,4}$, $led_{4,0}$ and $led_{4,4}$. The LEDs stay activated for approximatly one second. Afterwards, they are turned off. Again, we wait for a second and activate them again. This is repeated in an unfinite loop. The following source code shows the implementation details:

Listing 4: Basic Animation

```
int main(void)
{
  while(1)
  {
    videobuffer[0][0] = 1;
    videobuffer[0][4] = 1;
    videobuffer[4][0] = 1;
    videobuffer[4][4] = 1;
    waitForFrames(10);
    videobuffer[0][0] = 0;
    videobuffer[0][4] = 0;
    videobuffer[4][0] = 0;
    videobuffer[4][4] = 0;
    waitForFrames(10);
  }
}
```

Listing 4 is straight forward. We access four elements of *videobuffer*. Afterwards, *waitForFrames(10)* is called. Due to this, ten frames are displayed on the LED matrix. Afterwards, the previously activated LEDs are turned off and we display ten frames again. I soldered a simple LED matrix to test my work on real hardware. The result can be seen in figure 3. This was mainly done for learning purpose. You can also buy cheap LED matrices from asia if you don't want to assemble it yourself. Feel free to contact me if you have further questions.

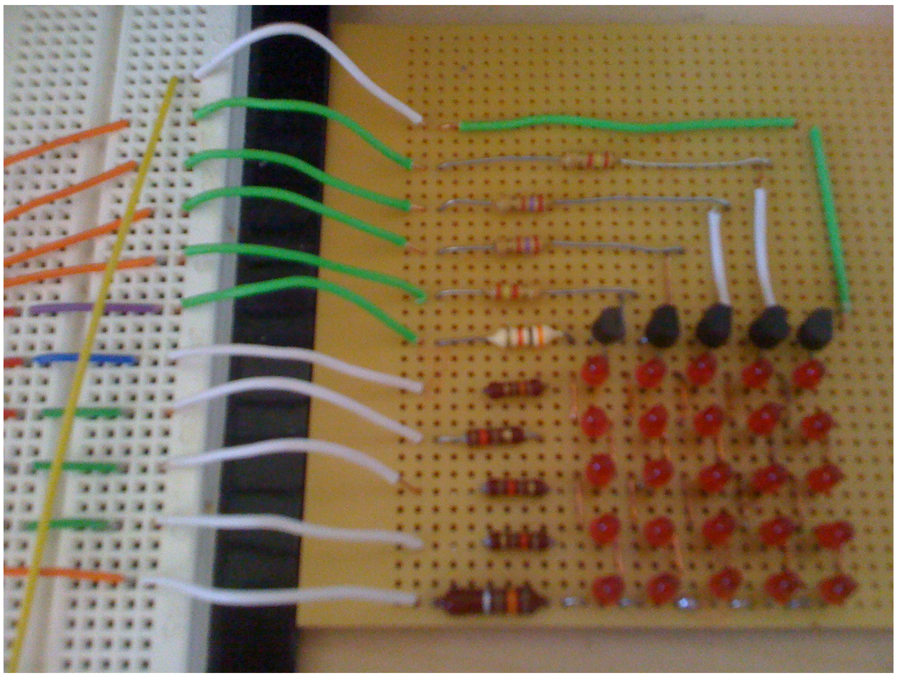

Figure 3: LED Matrix

4 Acknowledgement

I would like to express my gratitude to the whole *Nullsecurity* team for supporting this work.

License

References

[1] Atmel Corporation. Atmega8. `http://www.atmel.com/devices/atmega8.aspx`.

[2] Atmel Corporation. Atmel Corporation - Microcontrollers, 32-bit, and touch solutions. `http://www.atmel.com/`.

[3] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.