

Building a Smart-Necklace

by
Bruce E. Hall, W8BH

1) INTRODUCTION

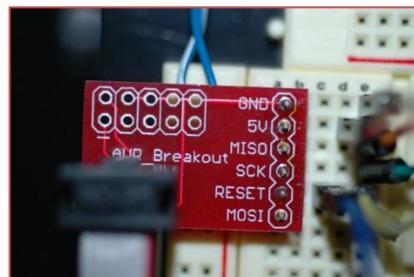
[Nuts & Volts](#) magazine is a great rag for the electronic hobbyist. In the July 2013 issue there is an article “Make a Smart Necklace” by [Craig Lindley](#). He piggybacks an LED matrix module directly on an AVR microcontroller, creating various animated displays. Craig’s work was based on previous work by [Alex Weber](#) and [tigeruppp](#). I immediately thought that such a necklace would be a fun gift for my 9 year-old daughter. All it takes is \$5 in parts, basic soldering skills, and a whole lot of programming! If you are interested, read on.

2) HOW TO START

The centerpiece of this project is the ATtiny4313 microcontroller by Atmel, but you can also use the cheaper and more widely available ATtiny2313. You also need an LED matrix, the LiteOn LTP-757. It is possible to use other models, but modules with different pinouts will require changes to the microcontroller code.

Next, you need a way to program the microcontroller. I have the AVRISP II, which costs about \$37 from Digikey (or eBay). This unit connects to your computer via USB, and connects to the microcontroller via a 6pin female header. I use a breadboard to make all of the connections. A \$1 ISP-breakout board from Sparkfun, shown below, lets you breadboard the output of the AVRISP programmer. Add an optional \$3 28-pin ZIF socket for your atmel chip, and make the following connections:

AVR ISP	ATtiny4313
Gnd	Gnd (pin 10)
5V	Vcc (pin 20)
MISO	Miso (pin 18)
SCK	Sck (pin 19)
RESET	Reset (pin 1)
MOSI	Mosi (pin 17)



You'll need to supply +5V power between pin 20 (Vcc) and pin 10 (ground). Download AVR studio from the atmel.com, and plug in the AVRISP programmer into a USB port on your computer. If all goes well, you should see two green lights on the programmer. The LED inside the programmer indicates USB power & data transfer; the LED on the case indicates status. If the status light is green, your 6 pin ISP cable is powered and connected correctly. Red indicates lack of +5V power. Orange indicates that the ISP connections are reversed.

When you have your green lights, it's time to talk to the microcontroller! Start AVR studio, choose 'Device Programming' from the Tools menu, or press Ctrl-Shift-P. Choose AVRISP II as the Tool, ATtiny4313 as the Device, and ISP as the Interface. Click Apply. Now click on the Device Signature Read button. A result of '0x1E920D' indicates successful 2-way communication with your microcontroller.

From this device programming window you can also set the microcontroller's fuses. Click on 'Fuses' in the left-hand pane. All the fuses except 'SPIEN' should be unchecked. (You will need to uncheck the CKDIV8 fuse.) Also, the SUT_CKSEL fuse should be set to INTRCOSC_4MHZ_14CK_65MS. This will run the chip at 4 MHz, using the internal RC oscillator. After checking your values, click the program button. You need to program the fuses only once.

3) CODING

For this project I chose 'C' as my programming language. For me, C is a bit easier to use than assembly language. More importantly, the authors mentioned above also used C for their projects. The complete source code for my project is given at the end of this article. Like many microcontroller projects, the outer shell of the program is very simple:

```
int main (void)
{
    init();
    main_loop();
    return(0)
}
```

First, init() is called for do-once, initialization steps. Next, main_loop is called to create interesting displays on the LED matrix. This loop is typically set up as an infinite loop, so that the program never ends.

The first initialization job is to set up microcontroller pins as inputs or outputs. We need only outputs for this project. To set a pin as an output, we write a '1' to the ports data direction register. DDRA is the data direction register for port A. Look at the first three lines:

```
DDRA = 0x03;           // 0000.0011
DDRB = 0x7E;           // 0111.1110
DDRD = 0x1E;           // 0001.1110
```

In Port A, we use the lowest two lines (A0 and A1), so the corresponding bits are set to logic 1:

A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	0	1	1

The binary number is 00000011, or hexadecimal 0x03. The code to set these pins A1 and A0 as outputs is: DDRA = 0x03. We need a total of 12 output lines. Looking at the comments for the three lines of code above, you will see exactly 12 bits set to logic one: two in port A, 6 in port B, and 4 in port D. Count 'em.

The next four lines set up a counter for our interrupt routine. First, let's assume that we need our code to be interrupted about 390 times a second (we do). The controller contains two separate counters that can be used for this purpose, timer/counter 0 and timer/counter 1. The first line sets the counter into 'Clear Timer on Compare 'CTC' mode, which means that the timer will count up to a value specified by the OCR0A, and then reset to zero.

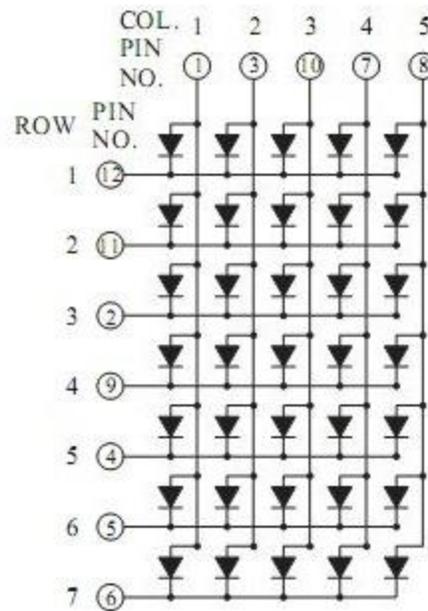
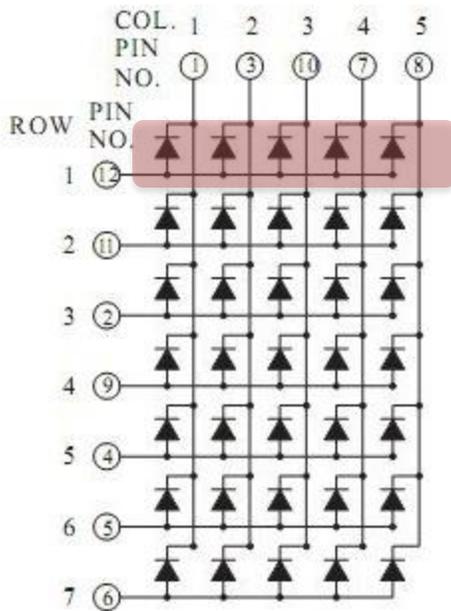
```
TCCR0A = _BV(WGM01);           // Set CTC mode
TCCR0B = _BV(CS02);           // Set prescaler clk/256 = 15625 Hz
OCR0A = 40;                   // 15625/40 = 390 interrupts/sec (5 cols = ~78fps)
TIMSK = _BV(OCIE0A);         // Enable T/C 0A interrupt
```

The speed at which it counts is set by the second line. This command sets the speed equal to the master clock divided by 256. In our case, that frequency is 4 MHz/256 = 15625 Hz. The third line specifies that the timer will reset after 40 counts, meaning that it will be resetting itself at a speed of 15625 Hz/ 40 = 390 times per second. The fourth and final line of code forces the microcontroller to be interrupted every time the counter resets.

But why do we need to interrupt our code? To do what? And why do we need to do it so often? We interrupt our code to multiplex the LED display. And we need to do it faster than 1/30 of a second, to take advantage of the 'persistence of vision' phenomenon.

3) MULTIPLEXING

Our display contains 35 individual LEDs, each with two connecting wires, and yet the entire module has only 12 pins. How can so many LEDs be controlled by only 12 pins? The answer is that the LEDs are connected in a matrix of 5 columns of 7 rows each. The two usual methods of creating this matrix are shown below.



The left schematic shows the common-row anode configuration (used by our LTP-757 matrix). Notice that in the highlighted bar, Row 1 is electrically connected to 5 LED anodes. The right schematic shows the common-row cathode configuration, in which each row pin is connected to 5 LED cathodes.

To turn on a LED at position (row1, col1) we apply an electrical current between pin 12 (row1) and pin 1 (col1). To turn the LED at (2,2) we apply a current between Row2 and Col2. But what if we want to turn on both of these LEDs at the same time? Because the LEDs share electrical connections in the matrix, activating rows 1-2 and cols 1-2 will turn on 4 LEDs rather than just the two we wanted. Our matrix reduces the number of electrical connections (and lets us use smaller microcontrollers), but restricts the combinations of LEDs that can simultaneously be lit.

To display arbitrary patterns, like alphanumeric digits, we use a technique called multiplexing. Starting with a blank display, each column is activated in turn. If the columns are sequenced fast enough, our eyes interpret the columns as being simultaneously lit. Problem solved! With a multiplexed display, we significantly reduce the number of electrical connections to the microcontroller.

3) THE INTERRUPT SERVICE ROUTINE

The ISR is responsible for handling our matrix multiplexing. It is called 390 times per second, regardless of whatever code the microcontroller is currently processing. The pseudo-code for the ISR is this:

1. Turn off LEDs in the previous column

2. Enable selected row bits for next column
3. Turn on the new column & return control

After five calls to the ISR, all five columns have been sequentially displayed. The display is therefore 'refreshed' at a rate of $390/5 = 78$ Hz, ensuring our eyes see a flicker-free display. Let's look at each of the three steps in the pseudo-code and see how they are accomplished.

First, turn off LEDs in the previous column. If only one column is displayed at a time, this is the same as turning off all the columns.

Look at the ports, and see which bits are connected to columns. Here are all the bits for each port, and their connections. R0 stands for Row 0, C1 for Column 1, etc:

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
PORTA	-	-	-	-	-	-	R2	C1
PORTB	-	R0	R1	C2	R3	C4	C3	-
PORTD	-	-	-	R6	R5	R4	C0	-

Remember that on this matrix the columns are cathodes, so they are activated with logic 0 and deactivated with logic 1. To turn off the LEDs, we must take all of the columns to a logic 1 state. Here is the same table, replacing all columns with logic 1 and everything else with logic 0:

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
PORTA	0	0	0	0	0	0	0	1
PORTB	0	0	0	1	0	1	1	0
PORTD	0	0	0	0	0	0	1	0

Now assign these values to each port. For example, Port B = binary 00010110 = 0x16.

```
// turn off all LEDs, by taking cathode (column) pins high
PORTA = 0x01;
PORTB = 0x16;
PORTD = 0x02;
```

The next task is to enable the row bits. I do these one at a time, looking at the desired pattern and setting the corresponding port bit with a logical OR instruction. Here is the code for the first row bit. After putting the desired row pattern into variable i:

```
if (i & _BV(0)) PORTB |= _BV(6);
```

The first part ($i \& _BV(0)$) is true if row0 needs to be activated. The second part activates row0 by setting bit 6 of PORTB, while keeping the other bits of PORTB unchanged. In the table above you'll see that bit6 of PORTB is connected to row0. The macro " $_BV()$ " creates a bit pattern with the desired bit set. In this case $_BV(6)$ equals binary 01000000, with bit6 a logic 1 and the remaining bits logic 0.

After all the selected rows bits are enabled, the last task is the turn on the LEDs by activating the column. According to the table above, to turn on column 0 we reset the value of PortD, bit 1. To set this bit we'd use a logical OR instruction, like above. But our common-cathode display needs this value to be zeroed. To reset a selected bit we use a logical AND with a zero at the bit position:

```
PORTD &= ~_BV(1); // activate column 0, turning on LEDs
```

Here, tilde is the NOT operator, turning `_BV(1)` from `0b00000001` into `0b11111110`.

After turning on the desired row bits and taking the desired column low, the selected LEDs in the current column turn on. The display will remain this way for 2.56 mS, until the ISR is called again.

3) BUFFERED DISPLAY

Our ISR handles all the dirty work. It turns on the necessary port pins to drive our display and makes sure the display is refreshed at an appropriate rate. The rest of the code simply provides data to display. The data handoff happens in the display buffer, a global variable that the main code and ISR can both access. A simple data buffer for our 5 column matrix is an array of 5 bytes, each byte corresponding to a column in the display:

	byte 0	byte 1	byte 2	byte 3	byte 4
Bit7	-	-	-	-	-
Bit6					
Bit5					
Bit4					
Bit3					
Bit2					
Bit1					
Bit0					

If we want to display the letter 'T' in our display, we must load the 5-byte buffer with the appropriate bits that will make the T symbol. Look at the table. There are 8 bits in a byte, but we need only 7 rows. So the top row is unused. Byte 0 will need one bit (bit 6) set. Setting the first byte to the value of `010000000 = 0x40` will give us this pattern. Byte 1 will need the same value. In Byte 2, the vertical bar of the T, the pattern is `01111111 = 0x7F`. Bytes 3 & 4 finish the top of the T with values `0x40`.

So, to display the letter T, will fill our buffer with the value (`0x40, 0x40, 0x7F, 0x40, 0x40`). An upside-down T would be (`0x01, 0x01, 0x7F, 0x01, 0x01`).

Whew! Do we have to map the whole alphabet onto a 5x7 matrix, and figure out the correct bits for each letter? Yes, but it's been done before;

you don't have to reinvent the wheel. The source code includes 107 different symbols for you to use. But if you know how the table is constructed, you can easily add your own symbols.

4) DISPLAYING A TEXT MESSAGE

With all the hard stuff out of the way, putting a text message on the display is very easy. For each character in your message, copy its symbol into the display buffer, then wait about a second. Repeat until the whole message is done.

```
void DisplaySymbol(int index)
// loads a font symbol into the display buffer
{
    for (int y = 0; y < COLS; y++)
    {
        buf[y] = pgm_read_byte(&(FONT_CHARS[index][y]));
    }
}

void DisplayText(const char *text)
// displays given text, one character at a time
{
    for (int i=0; i<strlen(text); i++)
    {
        DisplaySymbol(text[i]-' '); // display char
        DelaySecond();              // wait a while
    }                               // repeat for all chars
}
```

5) SCROLLING A TEXT MESSAGE

It's confession time: I have no idea how one is supposed to implement a text scroll function. So what follows may be the worst text scrolling routine ever made. But I made it for a recent raspberry pi project, and it works.

To scroll you need two pieces of data: the data that is currently being displayed and the data that is about to be displayed. This means we will keep track of (= buffer) data for two characters instead of one. To scroll, we shift our current display one column to the left, moving data from the invisible to the visible portion of the buffer by the same amount. English is written left-to-right, so leftward scrolling works the best.

Let's try an example, the word 'Pi':

X	X	X	X			X	X	X	
X				X			X		
X				X			X		
X	X	X	X				X		
X							X		
X							X		
X						X	X	X	

Here is our 10 byte buffer: the five blue columns represent what is being displayed on the LED matrix. The five green columns represent the buffer data waiting to be displayed.

The 'P' is being displayed, and the 'i' waiting its turn. If we scroll to the left by one column, the leftmost blue column disappears. Although only the 'P' is visible, both characters have shifted slightly to the left. To do this in code, each buffer column gets the contents of the column to its right, or `buf[i] = buf[i+1]`:

X	X	X			X	X	X		
			X			X			
			X			X			
X	X	X				X			
						X			
						X			
					X	X	X		

```
void ShiftLeft()
//    shifts the buffer one column to the left
{
    for (int i=0; i<10; i++)
    {
        buf[i] = buf[i+1];
    }
}
```

Look at the next scroll. The leading part of the 'i' is now appearing on the display. But wait a second – what if we were doing a longer word, like 'pie'. We need to add the 'e', don't we? There is too much space behind the 'i' already.

X	X			X	X	X			
		X			X				
		X			X				
X	X				X				
					X				
					X				
				X	X	X			

Don't worry. Remember that the green box is just a buffer, and isn't being displayed. We'll fill it with a new character as soon as the previous character has been completely shifted 'into the blue'.

It is time to complete the scroll routine. Scrolling a complete character is just shifting the character (and the initially-invisible next character) five times.

```
void Scroll()
//    scrolls a character onto the display
{
    for (int i=0; i<COLS+1; i++)
    {
        ShiftLeft();           // shift display 1 column to left
        DelayCS(SCROLLEDELAY); // and wait a while
    }
    // repeat 5x for whole character
}

void ScrollText(const char *text)
// scrolls given text across matrix, right to left
{
    for (int i=0; i<strlen(text); i++)
    {
        LoadSymbol(text[i]-' ') // get char into invis portion of buffer
        Scroll();               // and scroll it to be visible
    }
    // repeat for all chars
}
```

Now we have all the parts we need to scroll. Start with a blank display, and load the first character into the nonvisible (green) portion of the buffer. Scroll one column at a time, and display the data. Every 5th scroll, load a new character into the buffer. Done!

5) SLEEP

A fresh CR2032 coin battery will keep this necklace running for weeks, especially if you do not run the display often. The display uses far more current than the microcontroller. But a running microcontroller draws about 0.4 mA constantly. Over time it will drain the battery. To conserve battery power, you *could* remove the battery after each use.

I know that my daughter is unlikely to remove the battery, so I decided to use the 'sleep' function instead. Sleeping reduces power consumption to 3 uA, a 100-fold improvement. You get an additional 10-fold reduction in power (0.3 uA) by disabling the watchdog timer.

Implementing this deep sleep requires very few instructions. Here is the initialization code:

```
MCUCR = 0x30;           // 0011.0000 (sleep enabled, power-down mode)
WDTCR = 0x18;           // 0001.1000 set WD turn-off and WD enable bits
WDTCR = 0x10;           // 0001.0000 quickly reset WD enable to complete WD turnoff.
```

These lines turn on the necessary bits of the MCUCR and WDTCR registers. You may use macros in <sleep.h> and <wdt.h> that perform the same function, if you prefer.

When you are ready to put the chip to sleep, just call 'sleep()'. The sleep command will power-down the microcontroller until it is reset.

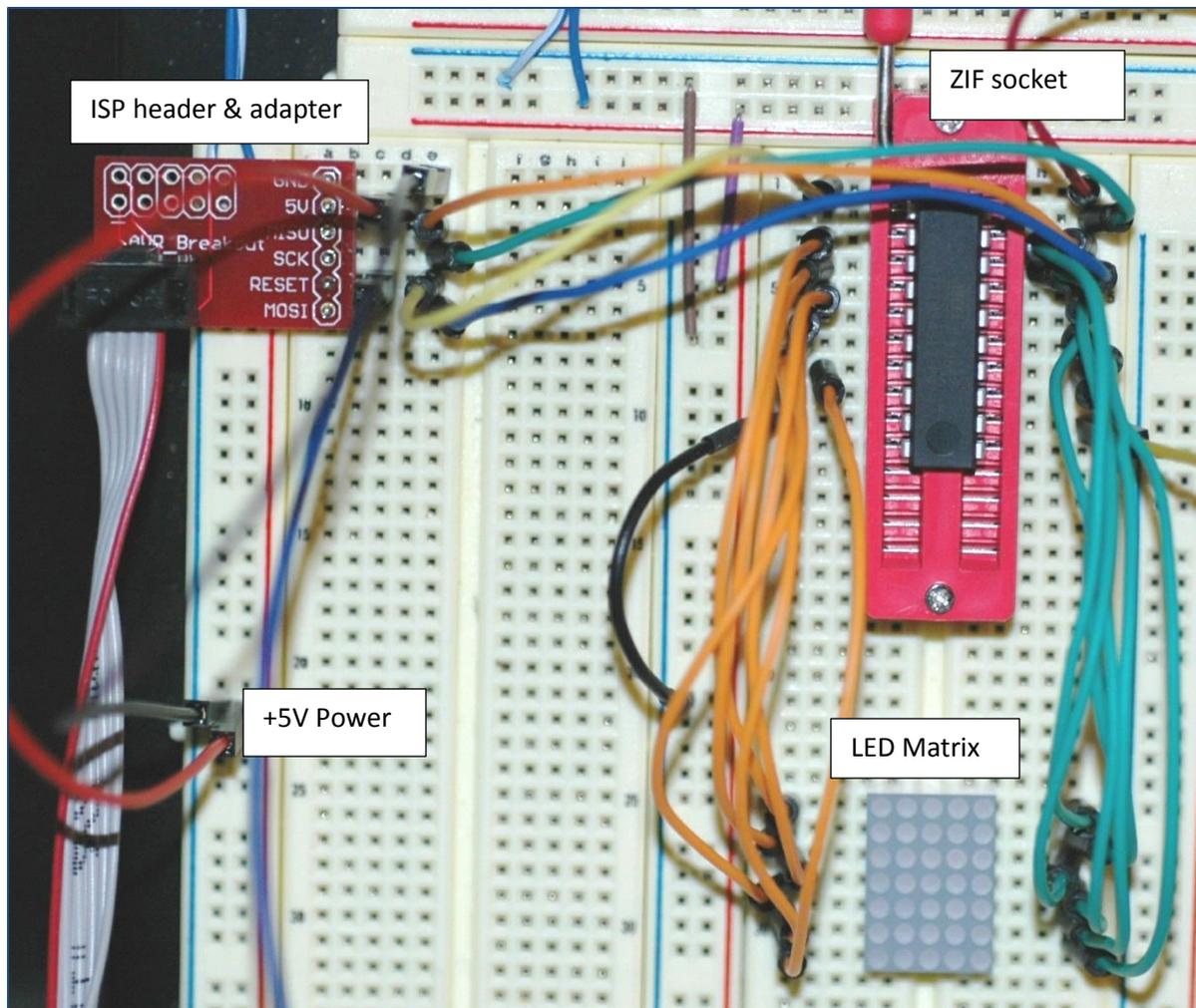
6) PROTOTYPING

My breadboard setup is pictured below. A +3.3/5 V power supply (not shown) powers the bus strips along the top and left side of the breadboard. I use a ZIF socket for my atmel chip, making it very easy to insert/program/remove the chip without disturbing the circuit. Notice that I have wired the LED matrix to the microcontroller without removing any of the programming circuitry. That is the beauty of in-circuit programming!

Here are the physical pin connections between the microcontroller & matrix:

Atmel (l)	Matrix (l)	Atmel (r)	Matrix (r)
Pin 3	Pin 1	18	12
4	2	17	11
5	3	16	10
6	4	15	9
7	5	14	8
8	6	13	7

The wires run parallel to each other without crossing. This pin mapping allows us to piggyback the LED matrix on top of the microcontroller.



7) BURN, BURN, BURN

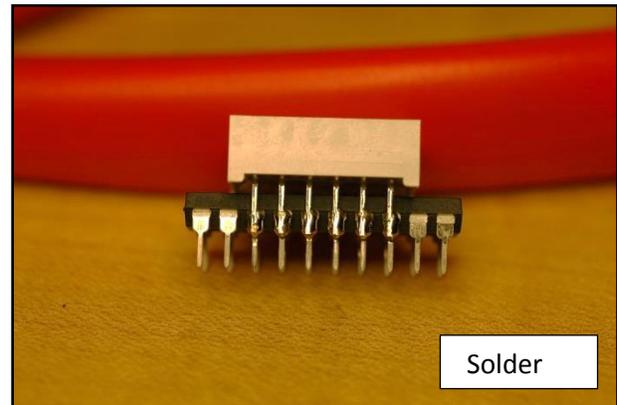
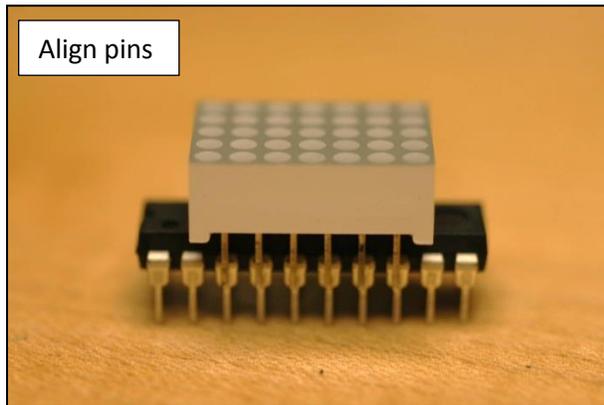
Compile your project in AVR studio. Then, under the Tools menu, select 'Device Programming'. Tool (AVRISP), Device (ATtiny4313) and Interface (ISP) should already be selected. Click Apply, then Memories. In the section called Flash, check the filename to make sure it corresponds to your just-compile file. Then click Program to flash your chip. After the programming is complete, the chip will reset and your code will run. There is no need to remove or manually reset the chip. Make sure that your code works the way you want, because reprogramming the chip your project is assembled will be considerably more difficult!

8) CONSTRUCTION

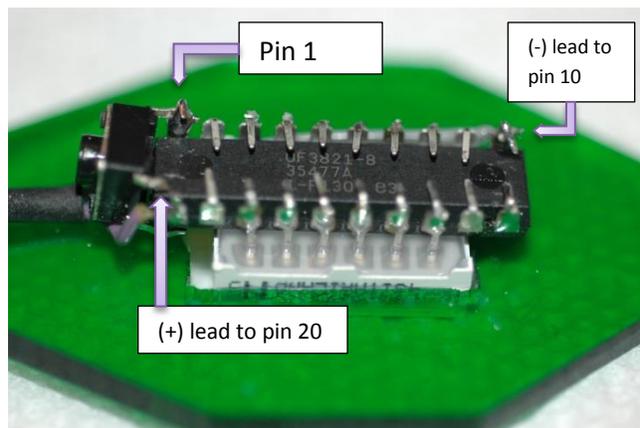
Remove the microcontroller and LED matrix from your prototyping circuit. Examine the pins on each carefully noting which is pin 1. On the atmel chip, pin 1 is at the notched end, and there is small dot in the case adjacent to it. On the matrix, the lettered edge is pins 6-12. If you hold the

matrix from the top between your thumb and forefinger, and your forefinger covers the lettered edge (pins 6-12), your thumb will point to pin 1.

Now put the LED matrix on top of the atmel chip, centering it so that pin 1 of the matrix is on top of pin 3 of the microcontroller. Each matrix pin should line up on top of a microcontroller pin, leaving two uncovered rows of microcontroller pins on top, and two rows on the bottom:



Solder one of the aligned sets of pins, and nudge the chips/pins so that the remaining sets are aligned. Now solder the remaining 11 sets of pins. Your circuit is now 90% complete. Insert the microcontroller/matrix combo back into the ZIF socket, and verify that the circuit works. In fact, you can still reprogram it! When you are satisfied everything works, remove the combo and continue construction.



I added a small reset switch to my project, so that the wearer can easily restart the display. I soldered one end of the pushbutton to the reset line, conveniently located on pin 1 at the top of the chip. The other end I connected to the battery (-) lead.

The two wires from the battery form the necklace chain. Connect the wire from the (+) battery lead to pin 20. Connect the (-) lead to the reset switch and pin 10.

I fed both wires through a small piece of heat shrink tubing for strain relief. You should mount your circuit on a pendant; otherwise the display won't stay face up when worn. I used a small piece of acrylic, cut out a 12mm x 17mm rectangular hole for the LED, then filed the hole until the LED fit snugly. I used Weld-On acrylic glue to secure the LED, but epoxy resin should also work. Finally, consider using a coat of hot glue over the exposed pins – the edges are sharp. Enjoy!

In [Part 2](#) of this series we will create a toolbox full of simple animations.

9) SOURCE CODE:

```
//-----  
// BMATRIX: NECKLACE with LED-MATRIX that displays text messages  
//  
// Based on Nuts&Volts Jul 2013 article: "Smart Necklace", p. 40  
//  
// Author   : Bruce E. Hall bhall66@gmail.com  
// Website  : http://w8bh.net  
// Version  : 1.0  
// Date     : 27 Jul 2013  
// Target   : ATTINY4313 or ATTINY2313 microcontroller  
// Language : C, using AVR studio 6  
//  
//-----  
//  
// Uses LITE-ON LTP-757G 5x7 LED MATRIX (Column Cathode) Display  
//  
// LED          to ATTINY4313          b PORTA  PORTB  PORTD  
//-----  
// LED FUNCTION to PORT (PIN)          7 -    -    -  
//-----  
// Col 0 - PD1 // Row 0 - PB6          6 -    row0 -  
// Col 1 - PA0 // Row 1 - PB5          5 -    row1 -  
// Col 2 - PB4 // Row 2 - PA1          4 -    col2 row6  
// Col 3 - PB1 // Row 3 - PB3          3 -    row3 row5  
// Col 4 - PB2 // Row 4 - PD2          2 -    col4 row4  
//          // Row 5 - PD3          1 row2 col3 col0  
//          // Row 6 - PD4          0 col1 -    -  
//  
// Since this a column cathode display,  
// Columns are active LOW; to set a col, PORTx &= ~(1<<bit)  
// Rows are active HIGH; to set a row, PORTx |= (1<<bit)  
//  
// Fuse settings: 4 MHz osc with 65 ms Delay, SPI enable; *NO* clock/8  
  
//-----  
// DEFINES  
  
#define F_CPU          4000000L          // run CPU at 4 MHz  
#define ROWS           7                // LED matrix has 7 rows, 5 columns  
#define COLS           5  
#define SCROLLDELAY    15              // delay in cs between column shifts  
#define FLASHDELAY     17              // delay in cs between symbol flashes  
#define BEATDELAY      30              // delay in cs between heartbeats  
#define HEARTCHAR      99  
#define TEXT1 "I Love You! "  
#define TEXT2 "Ich Liebe Dich! "  
  
//-----  
// INCLUDES  
  
#include <avr/io.h>                    // deal with port registers  
#include <avr/interrupt.h>             // deal with interrupt calls  
#include <avr/pgmspace.h>              // put character data into progmem  
#include <util/delay.h>                // used for _delay_ms function  
#include <string.h>                    // string manipulation routines  
#include <avr/sleep.h>                 // used for sleep functions  
  
//-----  
// GLOBAL VARIABLES  
  
char buf[12];                          // display buffer; each byte = 1 column
```

```

// buf[0] is the left-most column (col0)
// buf[4] is the right-most column (col4)
// buf[5] is a blank column between chars
// buf[6]..buf[10] are scrolled onto display

int curCol; // current column; values 0-4

const unsigned char FONT_CHARS[107][5] PROGMEM =
{
    { 0x00, 0x00, 0x00, 0x00, 0x00 }, // (space)
    { 0x00, 0x00, 0x5F, 0x00, 0x00 }, // !
    { 0x00, 0x07, 0x00, 0x07, 0x00 }, // "
    { 0x14, 0x7F, 0x14, 0x7F, 0x14 }, // #
    { 0x24, 0x2A, 0x7F, 0x2A, 0x12 }, // $
    { 0x23, 0x13, 0x08, 0x64, 0x62 }, // %
    { 0x36, 0x49, 0x55, 0x22, 0x50 }, // &
    { 0x00, 0x05, 0x03, 0x00, 0x00 }, // '
    { 0x00, 0x1C, 0x22, 0x41, 0x00 }, // (
    { 0x00, 0x41, 0x22, 0x1C, 0x00 }, // )
    { 0x08, 0x2A, 0x1C, 0x2A, 0x08 }, // *
    { 0x08, 0x08, 0x3E, 0x08, 0x08 }, // +
    { 0x00, 0x50, 0x30, 0x00, 0x00 }, // ,
    { 0x08, 0x08, 0x08, 0x08, 0x08 }, // -
    { 0x00, 0x60, 0x60, 0x00, 0x00 }, // .
    { 0x20, 0x10, 0x08, 0x04, 0x02 }, // /
    { 0x3E, 0x51, 0x49, 0x45, 0x3E }, // 0
    { 0x00, 0x42, 0x7F, 0x40, 0x00 }, // 1
    { 0x42, 0x61, 0x51, 0x49, 0x46 }, // 2
    { 0x21, 0x41, 0x45, 0x4B, 0x31 }, // 3
    { 0x18, 0x14, 0x12, 0x7F, 0x10 }, // 4
    { 0x27, 0x45, 0x45, 0x45, 0x39 }, // 5
    { 0x3C, 0x4A, 0x49, 0x49, 0x30 }, // 6
    { 0x01, 0x71, 0x09, 0x05, 0x03 }, // 7
    { 0x36, 0x49, 0x49, 0x49, 0x36 }, // 8
    { 0x06, 0x49, 0x49, 0x29, 0x1E }, // 9
    { 0x00, 0x36, 0x36, 0x00, 0x00 }, // :
    { 0x00, 0x56, 0x36, 0x00, 0x00 }, // ;
    { 0x00, 0x08, 0x14, 0x22, 0x41 }, // <
    { 0x14, 0x14, 0x14, 0x14, 0x14 }, // =
    { 0x41, 0x22, 0x14, 0x08, 0x00 }, // >
    { 0x02, 0x01, 0x51, 0x09, 0x06 }, // ?
    { 0x32, 0x49, 0x79, 0x41, 0x3E }, // @
    { 0x7E, 0x11, 0x11, 0x11, 0x7E }, // A
    { 0x7F, 0x49, 0x49, 0x49, 0x36 }, // B
    { 0x3E, 0x41, 0x41, 0x41, 0x22 }, // C
    { 0x7F, 0x41, 0x41, 0x22, 0x1C }, // D
    { 0x7F, 0x49, 0x49, 0x49, 0x41 }, // E
    { 0x7F, 0x09, 0x09, 0x01, 0x01 }, // F
    { 0x3E, 0x41, 0x41, 0x51, 0x32 }, // G
    { 0x7F, 0x08, 0x08, 0x08, 0x7F }, // H
    { 0x00, 0x41, 0x7F, 0x41, 0x00 }, // I
    { 0x20, 0x40, 0x41, 0x3F, 0x01 }, // J
    { 0x7F, 0x08, 0x14, 0x22, 0x41 }, // K
    { 0x7F, 0x40, 0x40, 0x40, 0x40 }, // L
    { 0x7F, 0x02, 0x04, 0x02, 0x7F }, // M
    { 0x7F, 0x04, 0x08, 0x10, 0x7F }, // N
    { 0x3E, 0x41, 0x41, 0x41, 0x3E }, // O
    { 0x7F, 0x09, 0x09, 0x09, 0x06 }, // P
    { 0x3E, 0x41, 0x51, 0x21, 0x5E }, // Q
    { 0x7F, 0x09, 0x19, 0x29, 0x46 }, // R
    { 0x46, 0x49, 0x49, 0x49, 0x31 }, // S
    { 0x01, 0x01, 0x7F, 0x01, 0x01 }, // T
    { 0x3F, 0x40, 0x40, 0x40, 0x3F }, // U
    { 0x1F, 0x20, 0x40, 0x20, 0x1F }, // V
    { 0x7F, 0x20, 0x18, 0x20, 0x7F }, // W
    { 0x63, 0x14, 0x08, 0x14, 0x63 }, // X
    { 0x03, 0x04, 0x78, 0x04, 0x03 }, // Y
    { 0x61, 0x51, 0x49, 0x45, 0x43 }, // Z
    { 0x00, 0x00, 0x7F, 0x41, 0x41 }, // [
    { 0x02, 0x04, 0x08, 0x10, 0x20 }, // "\ "
    { 0x41, 0x41, 0x7F, 0x00, 0x00 }, // ]

```

```

{ 0x04, 0x02, 0x01, 0x02, 0x04 }, // ^
{ 0x40, 0x40, 0x40, 0x40, 0x40 }, // ▾
{ 0x00, 0x01, 0x02, 0x04, 0x00 }, // ▽
{ 0x20, 0x54, 0x54, 0x54, 0x78 }, // a
{ 0x7F, 0x48, 0x44, 0x44, 0x38 }, // b
{ 0x38, 0x44, 0x44, 0x44, 0x20 }, // c
{ 0x38, 0x44, 0x44, 0x48, 0x7F }, // d
{ 0x38, 0x54, 0x54, 0x54, 0x18 }, // e
{ 0x08, 0x7E, 0x09, 0x01, 0x02 }, // f
{ 0x08, 0x14, 0x54, 0x54, 0x3C }, // g
{ 0x7F, 0x08, 0x04, 0x04, 0x78 }, // h
{ 0x00, 0x44, 0x7D, 0x40, 0x00 }, // i
{ 0x20, 0x40, 0x44, 0x3D, 0x00 }, // j
{ 0x00, 0x7F, 0x10, 0x28, 0x44 }, // k
{ 0x00, 0x41, 0x7F, 0x40, 0x00 }, // l
{ 0x7C, 0x04, 0x18, 0x04, 0x78 }, // m
{ 0x7C, 0x08, 0x04, 0x04, 0x78 }, // n
{ 0x38, 0x44, 0x44, 0x44, 0x38 }, // o
{ 0x7C, 0x14, 0x14, 0x14, 0x08 }, // p
{ 0x08, 0x14, 0x14, 0x18, 0x7C }, // q
{ 0x7C, 0x08, 0x04, 0x04, 0x08 }, // r
{ 0x48, 0x54, 0x54, 0x54, 0x20 }, // s
{ 0x04, 0x3F, 0x44, 0x40, 0x20 }, // t
{ 0x3C, 0x40, 0x40, 0x20, 0x7C }, // u
{ 0x1C, 0x20, 0x40, 0x20, 0x1C }, // v
{ 0x3C, 0x40, 0x30, 0x40, 0x3C }, // w
{ 0x44, 0x28, 0x10, 0x28, 0x44 }, // x
{ 0x0C, 0x50, 0x50, 0x50, 0x3C }, // y
{ 0x44, 0x64, 0x54, 0x4C, 0x44 }, // z
{ 0x00, 0x08, 0x36, 0x41, 0x00 }, // {
{ 0x00, 0x00, 0x7F, 0x00, 0x00 }, // |
{ 0x00, 0x41, 0x36, 0x08, 0x00 }, // }
{ 0x08, 0x08, 0x2A, 0x1C, 0x08 }, // ->
{ 0x08, 0x1C, 0x2A, 0x08, 0x08 }, // <-
{ 0xFF, 0x41, 0x5D, 0x41, 0xFF }, // 096: psycho 2
{ 0x00, 0x3E, 0x22, 0x3E, 0x00 }, // 097: psycho 1
{ 0x06, 0x15, 0x69, 0x15, 0x06 }, // 098: nuke
{ 0x0C, 0x1E, 0x3C, 0x1E, 0x0C }, // 099: solid heart
{ 0x0C, 0x12, 0x24, 0x12, 0x0C }, // 100: outline heart
{ 0x0A, 0x00, 0x55, 0x00, 0x0A }, // 101: flower
{ 0x08, 0x14, 0x2A, 0x14, 0x08 }, // 102: diamond
{ 0x07, 0x49, 0x71, 0x49, 0x07 }, // 103: cup
{ 0x22, 0x14, 0x6B, 0x14, 0x22 }, // 104: star2
{ 0x36, 0x36, 0x08, 0x36, 0x36 }, // 105: star3
{ 0x0F, 0x1A, 0x3E, 0x1A, 0x0F } // 106: fox
};

// -----
// INTERRUPT SERVICE ROUTINE
//
// Function: Light a column on the LED matrix display, according to contents
//           of display buffer.  buf[0] = leftmost column; buf[4] = rightmost
//
// This routine is called about 390 times per second, yielding a refresh
// rate for the whole display of 390/5 = 78 frames per second.

ISR (TIMER0_COMPA_vect)
{
    if (++curCol >= COLS) // advance column counter
        curCol = 0;

    // turn off all LEDs, by taking cathode (column) pins high
    PORTA = 0x01;
    PORTB = 0x16;
    PORTD = 0x02;

    // turn on individual row bits in this column
    char i = buf[curCol];
    if (i & _BV(0)) PORTB |= _BV(6);
    if (i & _BV(1)) PORTB |= _BV(5);
}

```

```

if (i & _BV(2))      PORTA |= _BV(1);
if (i & _BV(3))      PORTB |= _BV(3);
if (i & _BV(4))      PORTD |= _BV(2);
if (i & _BV(5))      PORTD |= _BV(3);
if (i & _BV(6))      PORTD |= _BV(4);

// turn selected column on
switch(curCol)
{
    case 0: PORTD &= ~_BV(1);    break;
    case 1: PORTA &= ~_BV(0);    break;
    case 2: PORTB &= ~_BV(4);    break;
    case 3: PORTB &= ~_BV(1);    break;
    case 4: PORTB &= ~_BV(2);    break;
}

}

// -----
// PROGRAM INITIALIZATION CODE

void init ()
{
    // set output pins
    DDRA  = 0x03;          // 0000.0011
    DDRB  = 0x7E;          // 0111.1110
    DDRD  = 0x1E;          // 0001.1110

    // setup Timer/Counter0 for LED refresh
    TCCR0A = _BV(WGM01);   // Set CTC mode
    TCCR0B = _BV(CS02);    // Set prescaler clk/256 = 15625 Hz
    OCR0A  = 40;           // 15625/40 = 390 interrupts/sec (5 cols = ~78fps)
    TIMSK  = _BV(OCIE0A);  // Enable T/C 0A interrupt

    MCUCR  = 0x30;         // 0011.0000 (sleep enabled, power down)
    WDTCSR = 0x18;         // 0001.1000 set WD turn-off and WD enable bits
    WDTCSR = 0x10;         // 0001.0000 reset WD enable to complete WD turnoff

    sei();                 // enable global interrupts
}

void DelayCS(int cs)
// Delays CPU for specified time, in centiseconds (1/100 sec)
// Calling _delay_ms in a routine prevents inlining, reducing code size,
// at the expense of slight timing inaccuracies.
{
    for (int i=0; i<cs; i++)
        _delay_ms(10);
}

void DelaySecond()
{
    DelayCS(100);
}

// -----
// CHARACTER SCROLLING ROUTINES

void ShiftLeft()
// shifts the entire display buffer one column to the left
{
    for (int i=0; i<11; i++)
    {
        buf[i] = buf[i+1];    // buf[0] on left; buf[11] on right
    }                          // each element represents a column
}                              // buf[0..4] are only elements visible

void Scroll()
// scrolls a character onto the display
{

```

```

        for (int i=0; i<COLS+1; i++)
        {
            ShiftLeft();           // shift display 1 column to left
            DelayCS(SCROLLDELAY);   // and wait a while
        }                           // repeat 5x for whole character
    }

void LoadSymbol(int index)
// loads a font symbol into the non-visible part of display buffer
{
    for (int y = 0; y < COLS; y++)
    {
        buf[y+5] = pgm_read_byte(&(FONT_CHARS[index][y]));
    }
    buf[11] = 0x00;               // add character spacing
}

void MakeVisible()
// copies char from non-visible to visible part of buffer
{
    for (int i=0; i<COLS; i++)
    {
        buf[i] = buf[i+5];
    }
}

void DisplaySymbol(int index)
// loads a font symbol into the visible display buffer
{
    LoadSymbol(index);
    MakeVisible();
}

void ScrollText(const char *text)
// scrolls given text across matrix, right to left
{
    for (int i=0; i<strlen(text); i++)
    {
        LoadSymbol(text[i]-' ')   // get char
        Scroll();                 // and scroll it
    }                             // repeat for all chars
}

void DisplayText(const char *text)
// displays given text, one character at a time
{
    for (int i=0; i<strlen(text); i++)
    {
        DisplaySymbol(text[i]-' '); // display char
        DelaySecond();              // wait a while
    }                               // repeat for all chars
}

// -----
// ANIMATION ROUTINES

void FlashHeart()
{
    DisplaySymbol(HEARTCHAR);      // flash heart on
    DelayCS(FLASHDELAY);          // wait
    DisplaySymbol(0);             // flash heart off
    DelayCS(FLASHDELAY);          // wait
}

void HeartBeat()
{
    FlashHeart();                 // heart on/off
    FlashHeart();                 // heart on/off
    DelayCS(BEATDELAY);           // wait
}

```

```

        FlashHeart();                // do it again!
        FlashHeart();
        DelayCS (BEATDELAY);
    }

// -----
// MAIN PROGRAM LOOP

void main_loop ()
{
    while(1)
    {
        for (int i=100; i<107; i++)
        {
            DisplaySymbol(i);        // display a fun symbol
            DelaySecond();
            DelaySecond();
            HeartBeat();              // heartbeats
            HeartBeat();
            DisplayText(TEXT1);      // display text1
            DelaySecond();
            HeartBeat();              // more heartbeats
            ScrollText(TEXT2);       // scroll text2
            DelaySecond();
        }                             // repeat 7 times
        sleep_cpu();                 // turn off display
    }
}

// -----
// MAIN

int main (void)
{
    init();                          // set up ports, CPU registers
    main_loop();                      // do the display, then sleep
    return (0);                       // that's all, folks!
}

```