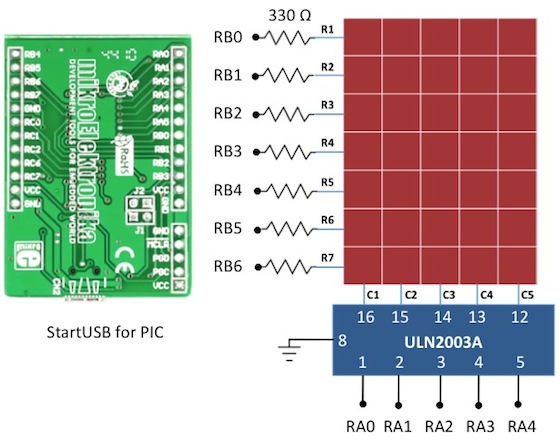
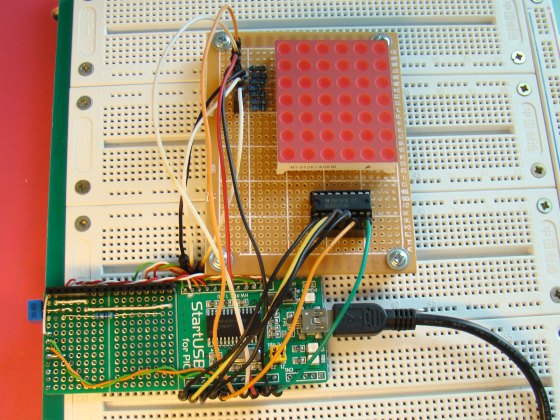
**Circuit Setup**

The circuit setup for this experiment is quite simple. You need seven 330 ? resistors in series with rows R1 through R7 to limit the current through the LEDs. Then the rows are driven by RB0 through RB6 pins of PIC18F2550. The columns are connected to the five outputs of ULN2003A. The corresponding five input pins of ULN2003A IC are controlled by RA0 through RA4 pins of PIC18F2550. The microcontroller will, therefore, scan across the column by sending appropriate bits to PORTA. For example, setting RA0 to 1 and clearing RA1 through RA4 bits, will select the first column. The microcontroller will wait for about 1 ms before switching to the next column. At each column, the microcontroller will output the corresponding row value at PORTB to turn on the appropriate LEDs in the column that are required to display the specific character. The switching between columns is fast enough to deceive the human eyes and a steady character is displayed.

[.[](http://embedded-lab.com/blog/wp-content/uploads/2011/04/LEDMatrixCircuit-.jpg)](http://embedded-lab.com/blog/wp-content/uploads/2011/04/LEDMatrixCircuit-.jpg)

Circuit diagram for interfacing a 5x7 LED dot matrix with PIC18F2550

[](http://embedded-lab.com/blog/wp-content/uploads/2011/04/LED_MatrixSetup.jpg)

Circuit setup showing a StartUSB board with a 6x7 LED dot matrix (the sixth column is discarded here)

**Software**

The major part of this experiment is the software routine to scan the columns and feed the rows with appropriate values. The column-specific row values for display characters can be either defined in RAM or stored in the program memory in case the on-board RAM is not sufficient enough. In mikroC, the variables are saved in RAM and constants are stored in program memory. So, if your PIC does not have enough RAM, you can define a constant array to store the row values so that a part of the program memory is occupied by it to free up the on-board RAM. PIC18F2550 has quite a bit of RAM (2 KB), so I have used RAM to store the row values for alphabets A through Z. Here’s how I define it in mikroC,

**unsigned short** Alphabets[130]={ 0x7e, 0×09, 0×09, 0×09, 0x7e, // A

0x7f, 0×49, 0×49, 0×49, 0×36,  // B

0x3e, 0×41, 0×41, 0×41, 0×22,

0x7f, 0×41, 0×41,0×22, 0x1c,

0x7f, 0×49, 0×49, 0×49, 0×63,

0x7f, 0×09, 0×09, 0×09, 0×01,

0x3e, 0×41, 0×41, 0×49, 0x7a,

0x7f, 0×08, 0×08, 0×08, 0x7f,

0×00, 0×41, 0x7f, 0×41, 0×00,  // I

0×20, 0×40, 0×41, 0x3f, 0×01,

0x7f, 0×08, 0×14, 0×22, 0×41,

0x7f, 0×40, 0×40, 0×40, 0×60,

0x7f, 0×02, 0×04, 0×02, 0x7f,

0x7f, 0×04, 0×08, 0×10, 0x7f,

0x3e, 0×41, 0×41, 0×41, 0x3e,

0x7f, 0×09, 0×09, 0×09, 0×06,

0x3e, 0×41, 0×51, 0×21, 0x5e,

0x7f, 0×09, 0×19, 0×29, 0×46,

0×46, 0×49, 0×49, 0×49, 0×31,  // S

0×01, 0×01, 0x7f, 0×01, 0×01,

0x3f, 0×40, 0×40, 0×40, 0x3f,

0x1f, 0×20, 0×40, 0×20, 0x1f,

0x3f, 0×40, 0×30, 0×40, 0x3f,

0×63, 0×14, 0×08, 0×14, 0×63,

0×07, 0×08, 0×70, 0×08, 0×07,

0×61, 0×51, 0×49, 0×45, 0×43 // Z

};

And this is how mikroC allows you to store arrays in the program memory.

**const unsigned short** characters[30]={

0×24, 0x2A, 0x7f, 0x2A, 0×12, // $

0×08, 0×14, 0×22, 0×41, 0×00, // <

0×41, 0×22, 0×14, 0×08, 0×00, // >

0×14, 0×14, 0×14, 0×14, 0×14, // =

0×36, 0×49, 0×55, 0×22, 0×50, // &

0×44, 0x3c, 0×04, 0x7c, 0×44, // PI

};

- See more at: http://embedded-lab.com/blog/?p=2478#sthash.lEhal0S1.dpuf