ENEL206: Computing and Modelling Laboratory 100 LED Display Simulator

Andrew Bainbridge-Smith, A404

June 3, 2008

1 Objective

To drive a simulated LED matrix display to show various visual patterns. This laboratory builds upon bitwise manipulations, looks at multiplexing and running a set of operations at a fixed known rate.

The arrangement used in this simulator is similar, but not identical, to the UCFK kit you will build.

2 Housekeeping

Download the zip file for the laboratory (lab100.zip), it contains 4 files: *pdisp.o, pdisp.h, utilities.h, main.c.*

You will be required to copy and edit *main.c* a number of times in this laboratory. The program are compiled using a two step process:

```
gcc -c -Wall -I. -o main.o main.c
g++ -o pdisp pdisp.o main.c ‘/usr/bin/fltk-config --ldflags‘
```

where the filename *main* changes on your copy.

The file *pdisp.o* is some c++ code that implements the LED display graphical interface, we must *link* our code (written in C) using the c++ compiler/linker (called g++). This is done in the second step.

The LED matrix is arranged as 5 columns in 7 rows. An individual LED is switched on by appropriately setting the value in 3 registers: PORTB, PORTC, and PORTD. These registers are modelled as 3 uint8_t global variables. We will also be using 3 real clock timers modelled as either a uint8_t or uint16_t global variable.

The LED matrix is connected to PORTB, PORTC and PORTD as shown in Figure 1. The matrix uses *active low* or *negative* logic — this means a zero turns the LED on and a one turns the LED off. Why is active low logic most commonly used?

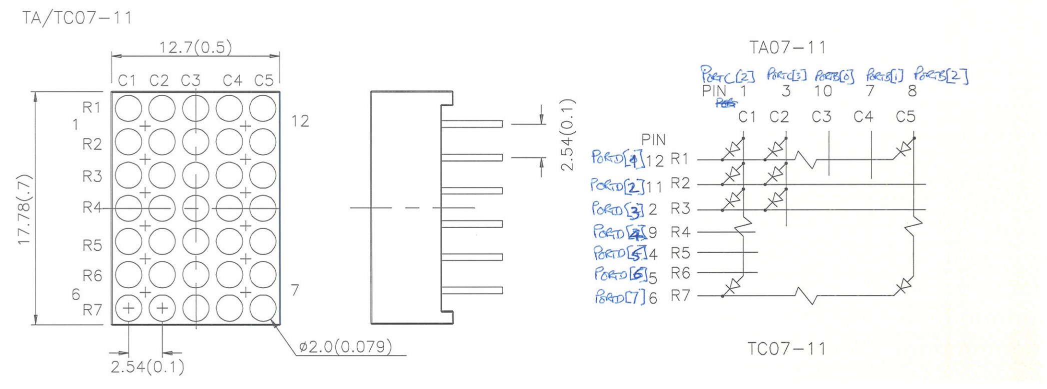

Figure 1: Control of the LED matrix requires row addressing (PORTD) and column addressing (PORTC and PORTB), i.e. the row addresses are multiplexed over the columns.

3 Simple LED patterns

Compile and run main.c with out modification. This example turns on the top left LED.

Now make a copy of main.c and using this copy modify the program to switch on another LED and a second modification to turn on a group of LEDS along a particular column.

Now with a new copy of main.c modify the program to remove the code while(1), but keep the code that is in the curly braces (indeed it is safe to keep the curly braces too). Explain why the program runs the way it does.

4 Flashing LED

In this section of the laboratory we are going to look at using a clock to control the execution of our code to occur at a regular interval. We will make two different flashing patterns.

The technique of controlling the regular interval of execution that we will use is known as a *paced loop*. The code takes the form of:

```
while(1)
{
   reset timer
```

```
      ... code to run at a regular paced rate

      while (timer < set time interval)
         continue;
   }
```

where timer is some real time clock, such that its integer value increments with each clock tick. In the simulator we are provided with 3 timers: TCNT0, TCNT1, TCNT2. Both TCNT0 and TCNT2 are 8-bit timers with a period of 20 milliseconds, while TCNT1 is a 16 bit timer with a period of 4 milliseconds. So to implement a paced loop with an interval or period of 1 second using timer TCNT0 we could write code like this:

```
   while(1)
   {
      TCNT0 = 0;

      ... code to run at a regular paced rate

      while (TCNT0 < 50)  // period is 1 second
         continue;
   }
```

An alternative way of writing this code is:

```
   while(1)
   {

      ... code to run at a regular paced rate

      while (TCNT0 < 50)  // period is 1 second
         continue;
      TCNT0 = 0;
   }
```

which will function in an identical way (except possibly at start up). This alternative form can be used to move the last 3 lines of code into a function to implement a timed delay — something you will look at latter with Michael Hayes.

Now using your knowledge of how to write a paced loop make a copy of main.c and modify it to make the top-left LED flash at a regular rate with a period of 0.5s. A wee hint: remember that the XOR function can be used to toggle the value of a bit.

Now make another copy of main.c and modify this version to successively turn on one LED in column 0 moving from the top to the bottom and returning to the top etc. , such that only one LED is on at a time and that the period each LED is on is 0.25s.

5 Displaying a complex pattern

We would like to display complex patterns on the LED matrix, such a character (e.g. 'A'). In terms of the LED matrix this could be drawn as the pattern:

○	○	●	○	○
○	●	○	●	○
●	○	○	○	●
●	●	●	●	●
●	○	○	○	●
●	○	○	○	●
●	○	○	○	●

We can also capture the same information as an array of 5 7-bit numbers, where the top row is the least significant position and the bottom row the most, and encoding from left column to right as: {0x7C, 0x0A, 0x09, 0x0A, 0x7C}.

Now the simplest way to control a display of 35 LED is to have an individual switch for each. However, in our simulator (and usually in practice) we must address an individual LED via a row address (PORTD) and a column address (PORTC and PORTB) — this leads to substantially fewer controllers, but a more complex control mechanism. We have effectively multiplexed the row address to a number of columns.

In order to display a complex pattern we must turn on the LEDs for a particular column, one column at a time. We must hold the LEDs on long enough so the eye can see them as on, but switch from one column to the next fast enough so that the eye is fooled into thinking that all the columns are being turned on at the same time. To achieve this we need a refresh rate (writing all columns) of somewhere between 25 and 100Hz.

Now make a copy of main.c and modify it to display a complex pattern. You will need a paced loop that has a frequency in the order given above. On each iteration of the paced loop will write a different column of the display.

Experiment with changing the speed of the paced loop, including outside of the bounds I recommend. On a real LED display the speed of the loop can effect the perceived brightness of the LEDs are you able to see this on the simulator.