Handwritten Character Recognition System Interfaced With A Resistive Touch Screen

Shantanu Agarwal
Darpan Sanghavi
Shreyans Jain

November 7, 2013

EP 315 - Microprocessor Lab Project

Supervised by Prof. Pradeep Sarin.

Abstract

The aim of our project is to design a touch screen controller that enables users to input decimal digits through a resistive touch screen. The controller will then try to recognize the inputted number and display it if the glyphs are sufficiently 'close' to some model digits. Currently we only aim to match the source 'as is' to the targets, without scaling, translating and rotating.

1 Introduction

Coming up with a generic algorithm for character identification is a challenging problem, due to the wide variety of ways in which a character may be written. Attempting to extract features (like curves and shapes) would prove too expensive for a microcontroller. Therefore, we use a distance function based on the bitmaps of the images, which quantifies the 'closeness' between them. There are several ways of defining this distance function. One of the most popular ways to compare two equal length strings is the Hamming distance, which is simply the number of positions at which the two strings are different. Though trivial to code and inexpensive, this does not take into account near misses, i.e., pixels are either matched or un-matched. To take care of such near miss cases, we use the Hausdorff distance between the two images to compare them[1].

2 Hausdorff Distance Between Two Images

Given two images as sets of dark pixels, the source S and the target T, the Hausdorff distance from S to T is defined as

$$h(S,T) = \max_{s \in S} \min_{t \in T} d(s,t) \tag{1}$$

where d can be any norm on the points of S and T. In this project, we have used the Euclidean norm $d(s,t) = \sqrt{(x_s - x_t)^2 + (y_s - y_t)^2}$. The Hausdorff distance defined in this manner is not symmetric. To make it symmetric, we can define the Hausdorff distance *between* S and T to be

$$H(S,T) = \max(h(S,T), h(T,S)) \tag{2}$$

This step is necessary to make the defined distance function a true metric. Otherwise we may have zero distances between distinct digits (for example, as the bitmap representing 3 is a subset of the one representing 8, the distance $h(3,8) = 0$), and the controller may report incorrect matches.

3 Setup

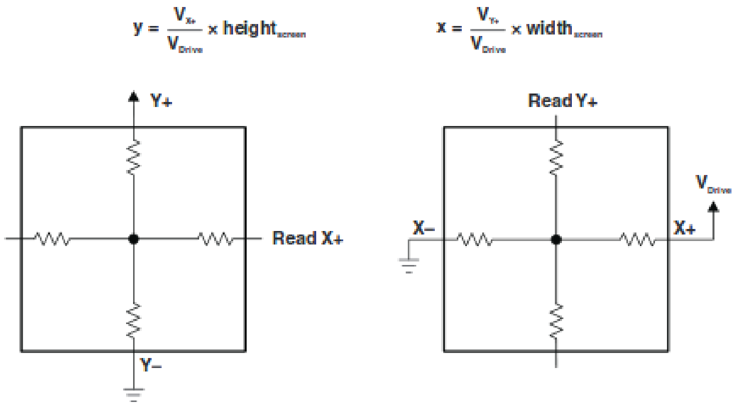
3.1 Microcontroller

We use an Arduino microcontroller board based on ATmega168 to make our touchscreen controller. It provides 1K of SRAM, which comfortably accommodates all the data except for the fixed model images. The flash memory (16K) provided is enough to store the program binary along with these models (We can store the model digits in the flash memory because they are read-only).

3.2 4-Wire Resistive Touch Screen

A resistive touch screen consists of two layers of transparent material coated with a conductor. When touched, the two layers come in contact and can act as a voltage divider. The coordinates of the touch can be found in two steps (Figure 1). First, a voltage gradient is applied in the Y direction on one of the layers and the voltage at $X+$ is measured. The ratio of this voltage to the drive voltage gives us the y-coordinate of the touch. The x-coordinate can similarly be found by applying the voltage gradient in the X direction and measuring the voltage at $Y+$. When the screen is not being touched the measured voltage is equal to the maximum possible voltage. We use this to detect if the screen is being touched ([2] provides a more robust way to detect a touch). To input characters using the touch screen, we make the microcontroller enter a loop when we detect a touch and keep storing the coordinates of the touch at small intervals. The loop is exited after the screen has not been touched for some time. This is done to accommodate for the fact that some characters may not be written with a single stroke. The resulting set of points is then treated as a bitmap of the input glyph.

Figure 1: Detecting the coordinates of a touch.



$y = \frac{V_{X+}}{V_{Drive}} \times height_{screen}$
$x = \frac{V_{Y+}}{V_{Drive}} \times width_{screen}$
Y+
Read X+
Y–
Read Y+
V_{Drive}
X–
X+

3.3 Display System

To display the detected number, we build a display system using seven-segment common cathode LED displays and CD4511 BCD-to-7 segment Latch Decoder ICs. Interfacing the LED displays directly with the microcontroller is not feasible because of the limited number of digital pins available on the Arduino board. We use 4 digital pins to output the detected digit in binary to the display system. These pins are common to all the digits in the display. In addition, we reserve one digital pin for each digit of the display to disable the latch of the corresponding IC when changing that digit. This ensures that only the desired LED display changes when a digit is written on the 4 output pins. The rest are 'latched' to their old states.

4 Code

The code used for this project is described briefly now. Most of it is a straightforward implementation of the concepts described in the previous sections.

4.1 Model Digits

We use bitmaps of the characters 0–9 of the GNU Unifont as model digits in our code. Each of these characters have a resolution of 8×12 (which also includes a one pixel boundary). These digits are stored in the flash memory as a list of dark pixels. The keyword PROGMEM indicates that modelneeds to be stored in the flash memory of the microcontroller. To use PROGMEM we must include the header pgmspace.h.

```
#include <avr/pgmspace.h>

 const unsigned
```

```
int model[10][26][2] PROGMEM = {
        {                               // 0
          {1,3}, {1,4},
          {2,2}, {2,5},
          {3,1}, {3,6},
          {4,1}, {4,6},
          {5,1}, {5,6},
          {6,1}, {6,6},
          {7,1}, {7,6},
          {8,1}, {8,6},
          {9,2}, {9,5},
          {10,3}, {10,4}
        },
        {                               // 1
          {1,4},
          {2,3}, {2,4},
          {3,2}, {3,4},
          {4,4},
          {5,4},
          {6,4},
          {7,4},
          {8,4},
          {9,4},
          {10,2}, {10,3}, {10,4},
                {10,5}, {10,6}
        },
                         ⋮

        {                               // 9
          {1,2}, {1,3}, {1,4}, {1,5},
          {2,1}, {2,6},
          {3,1}, {3,6},
          {4,1}, {4,6},
          {5,2}, {5,3}, {5,4},
               {5,5}, {5,6},
          {6,6},
          {7,6},
          {8,6},
          {9,5},
          {10,2}, {10,3}, {10,4}
        }
}
```

4.2 Getting Handwritten Characters

4.2.1 Getting the X and Y Coordinates

We define two functions, getxtouch() and getytouch(), which detect the coordinates of the point being touched.

```
void getxtouch()
{
  pinMode(y1+14, OUTPUT);
  pinMode(y2+14, OUTPUT);
  pinMode(x1+14, INPUT);
  pinMode(x2+14, INPUT);
  digitalWrite(y1+14, LOW);
  digitalWrite(y2+14, HIGH);
  delay(1);
  x = analogRead(x1);
  if (x > 512)
  {
    displayno = 1;
    posx = ((x-512)*BREADTH)/ANALOGB;
  }
  else
  {
    displayno = 0;
    posx = (x*BREADTH)/ANALOGB;
  }
}

void getytouch()
{
  pinMode(x1+14, OUTPUT);
  pinMode(x2+14, OUTPUT);
  pinMode(y1+14, INPUT);
  pinMode(y2+14, INPUT);
  digitalWrite(x1+14, LOW);
  digitalWrite(x2+14, HIGH);
  delay(1);
  y = analogRead(y1);
  posy = (y*HEIGHT)/ANALOGH;
}
```

They function exactly as described in section 3.2, the slight difference is that here the touch screen is split into two vertical sections which correspond to different digits in the display system. The posx, posy and displayno variables store the coordinates of the touch and the active LED display.

4.2.2 Loop Forming an Image of the Traced Curve

Once a touch is detected, the program goes in the following loop

```
for (int i = 0; i < 20; i++)
{
  do
  {
    if (posx > 0 && posy > 0
        && posx < (BREADTH - 1)
        && posy < (HEIGHT - 1)
        &&!dig[posy][posx])
    {
      dig[posy][posx] = true;
      dighi[count][0] = posy;
      dighi[count++][1] = posx;
    }
    getxtouch();
    getytouch();
    delay(10);
  } while (x < 1000 && y < 1000);
delay(10);
}
```

This loop checks for the coordinates of the touched point every 10ms, which is sufficient to give a close and continuous line at normal writing speed. The loop also makes allowance for multiple strokes by waiting a total of 200ms in 'non-touch' state. The time is sufficient to make 2 strokes (all 10 digits can easily be drawn with 2 strokes) and small enough that it does not cause any noticeable delay. The dig array stores the actual bitmap of the image drawn whereas dighi only stores a list of the dark pixels.

4.3 Matching the Image to the Models

The following piece of code computes the Hausdorff distance between the image and models (defined in section 2). To better resolve matchings, a variable mindisdeg is also maintained for each digit, which is used in case two models are equally distant from the image. It simply stores the ratio of the number of pixels which are Hausdorff distance apart to the total number of dark pixels in an image. A greater value (when distance is non-zero) indicates lesser similarity as there are a greater number of pixels at the maximum distance.

```
unsigned int mindis = 10000;
float mindisdeg = 0.0;
int match = -1;
for (int d = 0; d < 10; d++)
{
  unsigned int digdis1 = 0;
  unsigned int digdis2 = 0;
  unsigned int deg1 = 0;
  unsigned int deg2 = 0;
  for (int i = 0; i < pgm_read_word(&sz[d]); i++)
  {
    unsigned int curdis = 10000;
    for (int j = 0; j < count; j++)
    {
      unsigned
      int tmp = (pgm_read_word(&(model[d][i][0]))
                    - dighi[j][0])
```

```
               * (pgm_read_word(&(model[d][i][0]))
                       - dighi[j][0])
              + (pgm_read_word(&(model[d][i][1]))
                       - dighi[j][1])
               * (pgm_read_word(&(model[d][i][1]))
                       - dighi[j][1]);
    if (tmp < curdis)
           curdis = tmp;
  }
  if (digdis1 < curdis)
  {
    digdis1 = curdis;
    deg1 = 0;
  }
  deg1++;
}
for (int i = 0; i < count; i++)
{
  unsigned int curdis = 10000;
  for (int j = 0; j < pgm_read_word(&sz[d]); j++)
  {
    unsigned
    int tmp = (pgm_read_word(&(model[d][j][0]))
                       - dighi[i][0])
               * (pgm_read_word(&(model[d][j][0]))
                       - dighi[i][0])
              + (pgm_read_word(&(model[d][j][1]))
                       - dighi[i][1])
               * (pgm_read_word(&(model[d][j][1]))
                       - dighi[i][1]);
      if (tmp < curdis)
      curdis = tmp;
  }
  if (digdis2 < curdis)
  {
    digdis2 = curdis;
    deg2 = 0;
  }
  deg2++;
}
float tmpdeg = (float(deg1 + deg2))
              / (count + pgm_read_word(&sz[d]));
if (mindis > max(digdis1, digdis2) ||
              (mindis == max(digdis1, digdis2)
              && mindisdeg > tmpdeg))
{
  mindis = max(digdis1, digdis2);
  mindisdeg = tmpdeg;
  match = d;
  }
}
```

4.4 Displaying the Matched Number

The following code checks if the image is close enough to some model digit. If it is, then the 4 output binary bits are overwritten to store this new digit.

```
if (mindis < 7)
{
  bit3=false;
  bit2=false;
  bit1=false;
  bit0=false;
  Serial.println(match);
  if(match>7) bit3=true;
  match%=8;
  if (match>3) bit2=true;
  match%=4;
  if (match>1) bit1=true;
  match%=2;
  if(match>0) bit0=true;
}
```

Finally the displaydig() function unlatches the active display and outputs the new digit on it.

```
void displaydig()
{
  digitalWrite(latch + displayno, LOW);
  digitalWrite(pin0, bit0);
  digitalWrite(pin1, bit1);
  digitalWrite(pin2, bit2);
  digitalWrite(pin3, bit3);
  digitalWrite(latch + displayno, HIGH);
}
```

5 Results and Conclusion

The character recognition system worked fairly accurately. The number of false detections were $< 5\%$ whereas the number of cases in which a close enough match could not be found even after an honest attempt by the user was about 15%.

Considering the fact that a simple algorithm was used and the computing resources available were limited, such accuracy was unexpected. Incorporating rigid motions and scaling of images should give even better results. Though a variety of algorithms exist for character recognition, the method used here is fast and easy to implement. This project is more of a demonstration of the power of a theoretical construct like the Hausdorff distance than an attempt to build something of practical utility.

References

[1] D. Huttenlocher, G. Klanderman, and W. Rucklidge, *"Comparing Images Using the Hausdorff Distance,"* IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 15, no. 9, pp. 850-863, Sept. 1993.

[2] Texas Instruments, Appl. Report SLAA384A, Feb. 2008 [Revised Nov. 2010].