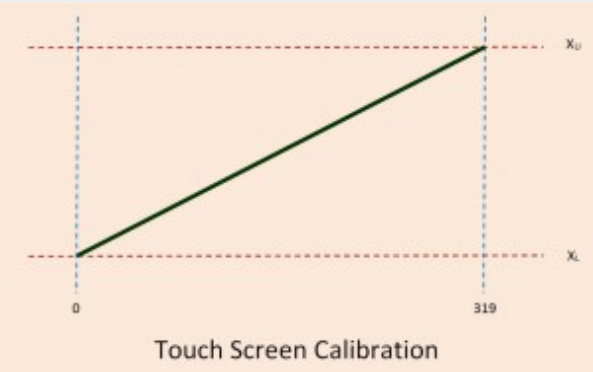
# Touch Screen Calibration

Posted on **February 10, 2012**

As promised, we're going to take a look at the calibration of resistive touch screens. If you need to brush up on touch screen basics, take a look at Touch Screen 101 first. When configured correctly, the touch screen outputs a voltage that corresponds to the position the touch location in the X axis and another voltage that corresponds to the voltage in the Y axis. For the X axis this might look something like this:




— Touch Screen Response

A lower voltage $X_L$ is output when the touch's X location is near the left edge of the screen, and $X_U$ is output for the right edge. In between the edges, the voltage is between $X_L$ and $X_U$ and linearly proportional to the touch position. Thus for a given voltage read, determining where the touch occurred is handled by the simple equation:

$$X = 320\frac{(X_{Read} - X_L)}{(X_U - X_L)}$$

Now for this all to work well, all we need to do is to establish the limits $X_L$ and $X_U$. It turns out that this is not as simple as you might think. I went through a number of iterations on this. Here's a

summary:

**Plan A**

My first idea was to try to measure $X_L$ and $X_U$ directly. If the user sweeps their touch off of the four edges of the screen, then at some point, we should read $X_U$ and $X_L$ as the current touch point. $X_L$ should be the lowest voltage we detect while the screen is touched and $X_U$ should be the highest one. The same applies for $Y_L$ and $Y_U$. Initially, “safe” values are used as limits that are greater than the lower limits and less than the upper limits and avoids non-sense like dividing by zero. The basic idea here looked something like this:

$$Initialization : X_U = X_{UpperSafeValue}, X_L = X_{LowerSafeValue}$$

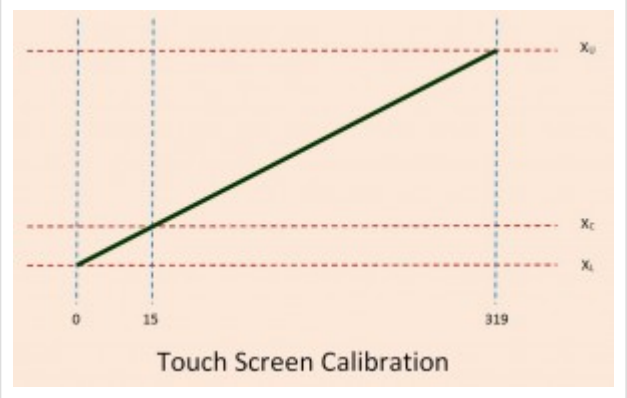
$$X_U = max(X_U, X_{Read}) \text{ and } X_L = min(X_L, X_{Read})$$

With similar code for the Y axis. And the code worked! Or at least the code seemed to work. Then I began noticing some problems. Sometimes the calibration routine would produce a limit that was too small causing the X,Y coordinate to be in error one way and sometimes it be too large causing an error the other way. It soon became apparent what was causing the problem. It seemed that the resistive touch screen’s noise level increases quite dramatically near the edges. This was then used to generate the limits (with noisy data) causing all parts of the touch screen to be inaccurate.

At first I thought I could just filter out the noise but it proved to be more difficult to resolve than that. It seemed that the touch data near the edges of the screen were just plain unreliable. Finally I had to conclude that you can’t calibrate with this noisy data; the results will always be disappointing.

**Plan B**

At this point I remembered that most touch screen calibration that I have seen place targets for the user to touch. These targets are not at the edge of the screen, but offset slightly in from the edge. In a flash, I saw what must be at work here. By retreating from the edge of the display, the quality of the data is greatly enhanced. The only problem is that the data is not the data we want.



This is shown in this diagram. Here the target has been inset 15 pixels from the left margin. The voltage we read in is $X_C$ and not what we want which is $X_L$. What we need here is to revisit the math classes of our youth and use some plain old linear algebra to do some linear extrapolation! We can see that the line from (0, $X_L$) to (319, $X_U$) has the same slope as the line from (15, $X_C$) to (319, $X_U$). The slope of a line is the rise over the run, so applying this to both lines we get:

$$\frac{X_U - X_L}{320} = \frac{X_U - X_C}{305}$$

So let's solve for $X_L$...

$$X_U - X_L = \frac{320}{305}(X_U - X_C)$$ – multiply by 320

$$X_L - X_U = -\frac{320}{305}(X_U - X_C)$$ – multiply by -1

$$X_L = X_U - \frac{320}{305}(X_U - X_C)$$ – add $X_U$ to both sides

$$X_L = X_U - \frac{64}{61}(X_U - X_C)$$ – reduce the fraction

Now there are a few loose ends in all this that you may have noticed. For starters, $X_U$ needs to be determined too. This approach deals with this in the following manner. Like Plan A above, all four limit values are given "safe" initial values. As the user press the various targets, the appropriate limits get updated. Even with poor limits, we can determine which corner the user touched and thus know which limit to update.

This approach gave much better and more accurate results than plan A ever did. Still there were lingering issues to be resolved:

1. A place was needed to store these parameters otherwise calibration would be needed every time the unit was powered up. Due to a design choice in the MMB32 hardware, going to "sleep" and keeping the results in RAM was not a real option.
2. The calibration is iterative. Updating one limits means that when the other limit is re-tested, a newer, better result is obtained. This means that the user must repeat each calibration point until "enough" accuracy can be obtained. How much is enough? Who can tell?

Issue 1 was handled by coming up with a way to utilize the M25P80 SPI flash device on the MMB32 in such a way as to NOT be hideously slow, or rapidly wear out the device and is the topic of a future article.

Issue 2 will be handled by a revamped calibration method "**Plan C**" coming soon!

Peter Camilleri (aka Squidly Jones)



This entry was posted in **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=251]** .