SLC-TS™

Analog Touch Screen for TERN's SmartLCD-Color

User's Manual

TERN INC.
1724 Picasso Avenue, Davis, CA 95616-0547, USA
Tel: 530-758-0180 Fax: 530-758-0181
Email: sales@tern.com *http://www.tern.com*

Version 1.1

June 9, 2004

© 1993-2004 TERN INC.
1724 Picasso Avenue, Davis, CA 95616-0547, USA
Tel: 530-758-0180 Fax: 530-758-0181
Email: sales@tern.com ***http://www.tern.com***

Important Notice

TERN is developing complex, high technology integration systems. These systems are integrated with software and hardware that are not 100% defect free. ***TERN products are not designed, intended, authorized, or warranted to be suitable for use in life-support applications, devices, or systems, or in other critical applications. TERN*** and the Buyer agree that ***TERN*** will not be liable for incidental or consequential damages arising from the use of ***TERN*** products. It is the Buyer's responsibility to protect life and property against incidental failure. ***TERN*** reserves the right to make changes and improvements to its products without providing notice.

Chapter 1: Introduction

1.1 Introduction

The purpose of this manual is to introduce the behavior and programming associated with the analog touch screen (TS) installed on TERN's SmartLCD-Color (SLC). Application notes are also included, with the purpose of helping the user create and design an effective user interface via the touch screen on the SLC.

1.2 Functional Description

The TS on the SLC is an analog touch screen. Its style allows the user to create buttons of random size and shape to accommodate the most unique application needs. The touch screen is interfaced to the SLC via a touch screen controller. When an object touches the screen, the touch screen controller interrupts the host CPU and makes analog readings at the host CPU's request. The user is then able to compare readings with preset values to determine the physical location that has been touched. This read/compare process allows the user to create unique user interfaces.

TERN provides sample code to demonstrate how to calibrate the touch screen to coordinate with the shape of the LCD. In other words, they demonstrate how to map the touch screen readings to the pixel locations of the LCD. By creating this map the user is able assign to routines to occur as a result of a touch on a certain part of the touch screen.

Yet the analog touch screen does introduce a challenge in programming an effective user interface. As with any analog signals, there exists noise. This is present in the SLC-TS system. The user must be aware of this and adjust the application accordingly. In addition to noise, varying "touch style" introduces another degree of complexity when creating an application based on the touch screen. By design, the TS is meant to be used with a uniform object, something that does not deform under light pressure. An example would be a stylus similarly used by PDAs (Personal Digital Assistants). These styluses offer consistent shape and speed when striking the touch screen. Another simpler, yet equally effective, tool is a pencil eraser. The eraser end of a standard pencil is an excellent object to use with the touch screen. It offers the same properties as the PDA pen. Ultimately, using an object of uniform shape, similar to the given examples, provides a higher of performance and precision when working with the TS, and reduces software overhead.

1.3 Terms

For purposes of this manual, let's create and define a set of vocabulary that will be used throughout this manual.

TS – Acronym for **T**ouch **S**creen.

touch – The act of striking the touch screen one time

TS value or **touch value** – This is the decimal value that is returned by software which contains the location of the touch

1.4 Application Planning

When deciding how to implement the TS on the SLC, thinking ahead is the best technique available. By doing a small amount of planning before beginning to write an application, you will save time, reduce software size, and reduce software complexity. There are a few aspects of the TS behavior to consider as a part of the pre-planning. There are discussed next.

1.4.1 Variation in touches

Since the TS on the SLC is an analog device, there exists noise, as previously mentioned. One behavioral outcome is reading different TS values from the same location of the TS. In other words, multiple touches from the exact same place on the touch can return different values. This variation in value from the same location will be small, but present nonetheless. The consequences of this behavior will be explored later.

The predominant effect of this variation occurs when the touch style varies between users. As the inconsistency of touch style increases, the variation in readings from a fixed location also increases. Here, touch style refers to how the user strikes the TS. Is the strike firm and quick, or gentle and slow? Also, if a finger is used, how does the size of the finger change the strike? Does the user use the pad of the finger, or the fingernail? All these questions introduce error. Planning ahead to handle such variations can only benefit TS performance and usability.

1.4.2 TS to LCD Mapping

Another application consideration is how to correlate the read TS values to a physical location on the TS. Two styles will be introduced here.

In no particular order, the first style mimics that of a PDA. It suits an application where the entire touch screen must be mapped into the physical location. This arises if the application allows the user to “draw” objects, or drag items to any part of the screen. In others words, the application demands that the software know how to handle activity on the entire area of the touch screen. This method of touch screen mapping requires a start-up calibration routine at each power-up.

A second method of mapping the TS to the LCD is when the area(s) that the user is allowed to touch are known and constant. For example, there exists *n* buttons of fixed size *i* with an unchanged location on the LCD. Since each button’s location and size remains constant, the touch values associated with those buttons remain constant. This allows the application designer to only be concerned with the touch areas associated with those buttons. This alleviates the need to map the entire touch screen to the entire physical LCD. This method does have drawbacks in the sense that the touch values are hard-coded into the application software, and cannot be represented as a variable. Yet, this method does introduce more precision.

1.4.3 Variation between Modules

A final consideration when creating a graphical user interface with the SmartLCD-Color is to realize that each TS will not produce the same exact results. Mounting variations, noise, and variation in analog circuitry can all produce a slight change in how the TS behaves from one unit to the next. As a result, knowing this ahead of time can avoid application overhaul down the road.

For example, given two SLCs, and touching the exact middle of the TS on each unit will result in a small variation between the touch values. Adjusting the application can minimize the effects of this behavior.

1.5 Sample Projects and Code

The TERN installation CD comes with a few sample projects and sample code ready to demonstrate the capabilities of the SLC. The sample code is also used to help start application development and to provide additional documentation about the code.

Mentioned below are the projects and sample code available on the TERN installation CD (relating to the TS):

Projects

slc.ide

This project includes targets such as “slc_c43” and “slc_g43”. They provide basic TS examples.

slc_gphx.ide

Each target in this project demonstrates the TS on the SLC.

Sample code

slc_draw.c

This file contains all of the graphics and button routines. Look in this file for routines that will be introduced later in this manual such as; put_bttn, push_bttn, scan_ts, sort, etc.

slc_ghpx.h

Refer to this header file for detailed documentation on button/graphics structures and routines.

Chapter 2: Reading the TS

2.1 Introduction

This chapter will cover a variety of topics with the focus on understanding how the user application reads a touch value. Topics covered include the hardware interface, software drivers, and software algorithms.

2.2 Hardware Interface

The SLC uses a touch screen controller to scan the TS and return values to the CPU. The touch screen controller interfaces the TS via 4-wires, and the touch screen controller interfaces the CPU via several PIO lines, including an external interrupt line. This section will give an overview of how the three components work together.

The touch screen is connected to the touch screen controller via a 4-wire flat cable. This provides the differential inputs to the touch screen controller for the x and y dimensions on the touch screen. The touch screen controller is a Burr-Brown ADS7843 (details can be found on the data sheet, \tern_docs\parts\ads7843.pdf). It uses a 6 wire serial interface to the CPU: clock, chip select, data in, busy output, data out, and pen interrupt. These are all tied to PIOs from the 186 CPU. Details can be found in the data sheet, but the basic theory of operation will be covered here. An 8-bit control byte is clocked into the touch screen controller. It contains information on how to handle the conversion, such as resolution and power options. At the end of the conversion, the touch screen controller will clock out the conversion result from data out.

As previously stated, the touch screen controller uses a 6 wire serial interface. The connections to the CPU pins are shown in the table below. TERN provides software drivers to drive the serial interface of the touch screen controller. Just one function call will return the result of a conversion.

ADS7843 Pin	CPU Pin	Description
DCLK	P24	Serial Clock. Synchronizes conversion and serial data I/O
/CS	P15	Active low chip select
DIN	P25	Serial data input. Latched on rising edge of DCLK
BUSY	P5	Busy output. High impedance when /CS is high
DOUT	/CTS0	Serial data out. Data shifted out on falling edge of DCLK
/PENIRQ	/INT2, P9	Pen interrupt. Requires external pull-up resistor.

2.3 Software Interface

TERN provides a software driver to return the result of a conversion on the touch screen controller. The control byte passed into the software driver determines the behavior of the touch screen controller. There is only one valid control byte needed for each axis. Thus, the user only needs to use the function call to get the conversion result and then determine how to handle the returned value.

```
unsigned char t8(unsigned char c)
```

The passed in argument ‘c’ is an 8-bit control byte that determines the touch screen controller’s behavior. The returned value is the 8-bit result of the previous conversion. Since a call to ‘t8’ returns the result of the previous conversion, one call is needed to start the conversion, and another to return the result. In addition, both the ‘x’ and ‘y’ coordinates are required, so the piece code for returning the ‘x’ and ‘y’ coordinates for a touch is:

```
t8(0x98);
x = t8(0x98);
t8(0xd9);
y = t8(0xd8);
```

Where ‘x’ and ‘y’ are of type unsigned char. The passed-in values of ‘0x98’ and ‘0xd8’ can be interpreted by referring to the ADS7843 data sheet. In order to read the touch screen, there is one more step to developing the piece code for a touch screen read. The touch screen uses an interrupt line to signal to the CPU when a touch has occurred, to alert the CPU when to complete the conversions. This line is tied to the CPU’s INT2. This interrupt is routed through a Schmitt trigger inverter (U9), changing the signal at the touch screen controller to /INT2. The P9 (A19) line is also tied to the /PENIRQ line of the touch screen controller. The touch screen controller needs to have bi-directional operation on the /PENIRQ line, and since the INT2 line routed through the trigger inverter, it can only be input. P9 acts as the output to the touch screen controller.

When a touch is detected on the TS, the touch screen controller pulls /INT2 low (high at the CPU because of the inverter), generating and interrupt on the CPU. The CPU can then service the interrupt by initiating the TS conversions. After the CPU has detected the interrupt, it pulls P9 low to prepare the touch screen controller for the conversions. At the end of the conversions, software must set P9 as an input (to avoid conflict with the output of the touch screen controller). When the touch on the TS is removed, the touch screen controller pulls /PENIRQ high to signal the end of the touch. Below is an expanded piece code needed to read the touch screen. This piece code assumes that the ISR (interrupt service routine for interrupt 2) simply increments a flag (called ‘int2_cnt’ in this example). This polling technique for handling interrupts shortens the length of the ISR and makes the code easier to debug. This piece code is not exhaustive. See the sample code \tern\186\samples\slc\slc_c43.c for a complete example on reading the touch screen.

```
1     if(int2_cnt)                          // We have seen an interrupt
2     {                                     // Means a touch has occurred
3
4           int2_init(0,int2_isr);          // Disable while we read TS
5           pio_init(9,2);                  // Set P9 to Output mode
6           pio_wr(9,0);                    // Drive P9 low to enable
7                                           // conversions
8
9           t8(0x98);                       // Start conversion on X axis
10          x = t8(0x98);                   // Get result X axis
11          t8(0xd9);                       // Start conversion on Y axis
12          y = t8(0xd8);                   // Get result of Y axis
```

```
13          pio_init(9,1);          // Set P9 back to input to
14                                  // avoid conflict with /PENIRQ
15                                  // output
16
17          // wait until the ADS7843 tells us the touch is over
18          while(!(0x0200 & inport(0xff74)));
19
20          int2_init(1,int2_isr);        // Re-enable for next touch
21          int2_cnt = 0;                 // Clear flag
22
23    }     // Done. Now use 'x' and 'y' to take appropriate action
```

Again, this piece code does not represent all code needed for a read on the touch screen. It is used here to develop an understanding of the process flow needed to get a read on the touch screen. Later, we will build upon this piece code and incorporate sampling methods to create a more reliable touch screen interface.

2.4 Building an algorithm

Now that we have the basics for reading the touch screen complete, let's explore ways to make our piece code more robust. This section will show ways to understand the values that the touch screen controller returns. Do the values make sense? What patterns and consistencies, if any, can we see? This section will explore ways to improve the algorithm responsible for reading the touch screen. The methods used are only examples and not meant to represent the best way, or the only way to read the TS.

In the above section we make one read for each axis, simply placing the conversion result into the unsigned chars 'x' and 'y'. But using just one sampling for each axis can be risky. It does not do a very good job of accommodating for noise, and touch variation that was discussed in chapter 1. In the above section, lines 9-12 are responsible for actually retrieving the touch screen values. Lets replace those lines with a **for** loop that takes multiple samples for each axis. Make sure to change the 'x' and 'y' variables into arrays of appropriate size. Consider the following:

```
for(i=0;i<NUM_SAMPLES;i++)
{
      t8(0x98);                     // Start conversion on X axis
      x[i] = t8(0x98);              // Get result X axis
      t8(0xd9);                     // Start conversion on Y axis
      y[i] = t8(0xd8);              // Get result of Y axis
}
```

Now, for each axis we have 'NUM_SAMPLES' samples. This technique is important in making sure the touch values are consistent, and most importantly accurate. By taking multiple samples and then averaging, the performance will increase.

Now let's do some investigation of the values we just got. Typically, the SLC touch screen has a horizontal range of about 28 – 218, for a total of 190 counts, and a vertical range of about 30 – 220, for a total count of 190. These values will vary slightly per unit based on variations discussed in Chapter 1. They are meant to give an idea of the typical resolution seen on the touch screen. If we know the approximate range for each axis, let compare the read values in 'x' and 'y' with the know ranges, specifically looking for the difference between the largest and smallest values in our arrays 'x' and 'y. If that difference approaches a certain threshold, it becomes reasonable to say that the current touch is invalid. Consider the following code, placed after line 14:

```
unsigned char xMax, xMin;
unsigned char yMax, yMin;
xMax = x[0];                    // Set to first values of array
xMin = x[0];
yMax = y[0];
yMin = y[0];

for(i=0;i<NUM_SAMPLES;i++)
{
      if(x[i] >= xMax) xMax = x[i];       // Find largest value in array
      if(x[i] <= xMin) xMin = x[i];       // Find smallest value
      if(y[i] >= yMax) yMax = y[i];
      if(y[i] <= yMin) yMin = y[i];
}
if(xMax – xMin > xThreshold)  // If difference is too big, discard touch
      return;           // Get out, invalid touch

if(yMax – yMin > yThreshold)  // If difference is too big, discard touch

      return;           // Get out, invalid touch
```

The values that are assigned to the constants 'xThreshold' and 'yThreshold' can be application specific, dependent on a number of factors. The designer should choose these values to reflect the desired precision and accuracy. For purposes of example, let's pick a number that could be used. If we know that the complete horizontal range of the touch screen is 190, then it would be reasonable to say that if the difference between 'xMax' and 'xMin' is greater than 48 (which is about one quarter of 190), then the touch is invalid. If that difference is greater than one fourth the width of the entire screen, then it was most likely a highly inaccurate touch, and should be discarded. The number chosen here obviously still allows a good deal of error to be considered okay. It is up to the application designer to choose the value to fit the application.

Lastly, let's simply average the values. Consider the following, placed after line 16:

```
unsigned int xFinal, xSum = 0;
unsigned int yFinal, ySum = 0;
for(i=0;i<NUM_SAMPLES;i++)
{
      xSum += x[i];                 // Add all values of array
      ySum += y[i];
}
xFinal = xSum / NUM_SAMPLES;        // Divide by number of samples
yFinal = ySum / NUM_SAMPLES;
```

In our efforts to create a more robust sampling method for the touch screen, we have added an averaging loop and checked the range of the sampled values. Let take a look at what we have so far, all together:

```
1     if(int2_cnt)                          // We have seen an interrupt
2     {                                     // Means a touch has occurred
3
4           int2_init(0,int2_isr);          // Disable while we read TS
5           pio_init(9,2);                  // Set P2 to Output mode
6           pio_wr(9,0);                    // Drive P9 low to enable
7                                           // conversions
8           for(i=0;i<NUM_SAMPLES;i++){
9                 t8(0x98);                       // Start conversion on X
10                x[i] = t8(0x98);                // Get result X axis
11                t8(0xd9);                       // Start conversion on Y
12                y[i] = t8(0xd8);                // Get result of Y axis
13                pio_init(9,1);          // Set P9 back to input to
14          }                             // avoid conflict with /PENIRQ
15                                        // output
16
17          unsigned char xMax, xMin;
18          unsigned char yMax, yMin;
19          xMax = x[0];            // Set to first values of array
20          xMin = x[0];
21          yMax = y[0];
22          yMin = y[0];
23
```

```
24          for(i=0;i<NUM_SAMPLES;i++)
25          {
26          if(x[i] >= xMax) xMax = x[i];       // Find largest value in
27                                              // array
28          if(x[i] <= xMin) xMin = x[i];       // Find smallest
29          if(y[i] >= yMax) yMax = y[i];
30          if(y[i] <= yMin) yMin = y[i];
31          }
32          if(xMax - xMin > xThreshold)  // If difference is too big
33                                        // discard touch
34                return;
35
36          if(yMax - yMin > yThreshold)  // If difference is too big
37                                        // discard touch
38                return;
39
40          unsigned int xFinal, xSum = 0;
41          unsigned int yFinal, ySum = 0;
42          for(i=0;i<NUM_SAMPLES;i++)
43          {
44                xSum += x[i];           // Add all values of array
45                ySum += y[i];
46          }
47          xFinal = xSum / NUM_SAMPLES;  // Divide by number of samples
48          yFinal = ySum / NUM_SAMPLES;
49
50          // wait until the ADS7843 tells us the touch is over
51          while(!(0x0200 & inport(0xff74)));
52
53          int2_init(1,int2_isr);        // Re-enable for next touch
54          int2_cnt = 0;                 // Clear flag
55
56    }     // Done. Now use 'x' and 'y' to take appropriate action
```

Note that in this combined piece code we have a few statements out of order. A compiler would complain about lines 17, 18, 40, and 41. They simply need to be moved to the beginning of the loop. For this example they are left where they are. The value assigned to NUM_SAMPLES is completely arbitrary. It is

again up to the application designer to determine what value it should take based upon variations learned about in Chapter 1 of this manual. TERN samples use 16 or 32.

2.5 Advanced Algorithm Techniques

This section will discuss additional ways to improve the touch screen performance via software. Techniques offered here can be incorporated into the algorithm developed above. As always, it is the application designer's choice how to ultimately implement these methods.

2.5.1 Omitting out-of-range values

Another way to examine the precision of a touch and the corresponding touch values is to omit values that do not fall within a reasonable range of the average value read.

Consider the algorithm created in section 2.4. Assume that NUM_SAMPLES is equal to 32. Now, instead of just blindly averaging the 32 values, let's look at the values first. Start by sorting the values numerically, that way we can get a better picture what king of numbers make up the 32 samples. Let's sort the numbers. These function calls could be inserted into line 15:

```
unsigned char xSort[NUM_SAMPLES];   // Array for sorted values
unsigned char ySort[NUM_SAMPLES];   // Array for sorted values

sort(&x[0], &xSort[0]);       // source, destination
sort(&y[0], &ySort[0]);       // source, destination
```

where ***void sort(unsigned char * s, unsigned char * d);*** could be written elsewhere in the code as the following:

```
void sort(unsigned char * s, unsigned char *d)
{
   int i=0,j=0;
   for(j=0;j<NUM_SAMPLES;j++)
   {
     *d = *s;
     for(i=0;i<(NUM_SAMPLES-1);i++)
     {
        if(*(s+i+1) > *(d+i))
        {
           *(d+i+1) = *(d+i);
           *(d+i) = *(s+i+1);
        }
        else
```

```
        {
           *(d+i+1) = *(s+i+1);
         }
      }

      // copy everything from destination back into source
      for(i=0;i<NUM;i++)
       *(s+i) = *(d+i);
   }
}     // end sort
```

Now, after all the values have been sorted numerically, they can be examined. During a touch on the touch screen, software may begin to return touch values as the user is still pressing the finger onto the screen. These first values read might contain a lot of error and noise because the finger has not yet stopped on the screen (and could still contain some lateral movement). And, depending on the speed of the touch, more values could be read as the finger is being removed from the touch screen. These first and last values often do not accurately represent the actual touch value expected for that location on the touch screen. By sorting the NUM_SAMPLES numerically, we can easily throw away these out-of-range values, before starting the rest of the touch screen algorithm. We are now at a point where the application designer can make a choice based upon application needs and performance expectation. For purposes of example, let's omit the largest 8 values and the smallest 8 values, with the idea that the middle 16 values (numerically) more closely represent the final touch value we are looking for. This is shown below:

```
      unsigned char xSort[NUM_SAMPLES];   // Array for sorted values
      unsigned char ySort[NUM_SAMPLES];   // Array for sorted values

      sort(&x[0], &xSort[0]);       // source, destination
      sort(&y[0], &ySort[0]);       // source, destination

      for(i=0;i<16;i++)
      {
            x[0] = xSort[i+8];
            y[0] = ySort[i+8];
      }
```

We simply added the for loop in **bold** which copies the middle 16 values of the sorted array back into our original array. We can now proceed with the rest of the algorithm and concentrate only on the first 16 values of the 'x' and 'y' arrays. By using this technique, we can avoid having entire touches thrown out because of a few values that are obviously wrong. In addition, it should improve the accuracy of the final averaged value.

2.5.2 Debouncing

Despite waiting for the user to complete the touch (line 51 of the algorithm), bounce can still occur. Depending on the application, bounce could be damaging to the behavior of the user interface. Although simple, a delay statement at line 52 will nullify most, if not all, bounce situations. It is ultimately up to the application designer to determine how much delay is necessary. Using a delay at line 53 is effective because it gives the user time to completely remove the touch apparatus from the screen before re-initializing the interrupt line. TERN code uses delays similar to the following:

```
delay_ms(100);
```

The delay amount (the argument of the function call) can vary, yet values in the range of 100-300 are known to be effective. The delay amount also depends if the application designer wants to allow the user to hit buttons at a high frequency. Clearly, a larger argument forces the user to slow the touch rate.

2.6 Final Notes

The piece code in this chapter provides most of what is need to read the touch screen on the SLC. However it is not complete. It requires variable declarations, initialization routines (such as slc_init). It also requires the user to initialize INT2. Refer to \tern\186\samples\ae\intx.c for an example of how to initialize interrupts.

Sample code in the \tern\186\samples\slc directory contains examples that demonstrate the topics of this chapter. See sample files such as

slc_c34.c

slc_g34.c

slc_draw.c
This file is not a complete sample. It simply provides function definitions for examples shown in this chapter.)

slc_gphx.ide provides samples which incorporate slc_draw.c.

Chapter 3: Create an Interface

3.1 Introduction

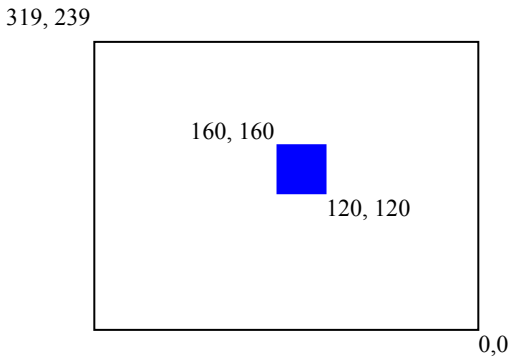
Now that we know how to read the touch screen, what do we do with the values we get? This Chapter will explore ways to create a useable user interface - in other words, how to correlate touch values with LCD graphics.

3.2 Create a simple button

Let's jump right into an example that we can build on throughout the course of this Chapter. We will create a simple button near the middle of the screen. The steps to accomplishing this will be: draw the button on the LCD and scan the touch screen to determine what touch values correspond with that button. Then in our application, if we see a touch value within the boundaries we created for that button, we can execute a task that we assign to that button. Let us start by drawing a box on the LCD to represent our button:

```
background(7);          // default for white
drawRect(120, 120, 160, 160, 1);    // see slc.h for details
```

These routines are included in "slc.lib" and the prototypes can be found in "slc.h". The result of these statements will produce an LCD that looks like the following:



With this done, we can know determine what the touch screen values are at the boundaries of our button. Repeat the same code, but add a loop to read touch values. As you touch the boundaries of the button, record four values, one for each boundary of the button. (The follow piece code assumes the algorithm from Chapter 2 has been incorporated into a single function. For this example we will call the function "scan_ts", and the prototype will be given below.)

```
background(7);          // default for white
drawRect(120, 120, 160, 160, 1);    // see slc.h for details
while(1)
      scan_ts(&location[0]);        // unsigned int location[2]
```

where the function **`unsigned int scan_ts(unsigned int * location);`** returns a '1' when a valid touch is seen and a '0' for no touch, or an invalid touch. When the function returns, the "location" array will contain the 'x' and 'y' values of the touch (assuming it returns a '1').

After finding our four touch screen values that correspond to our button, we are ready to use the button. Let's call the touch screen boundaries `top, bottom, left,` and `right`. Assign the boundary values to these variables. Modify the code one more time to create our first basic user interface:

```
1     background(7);          // default for white
2     drawRect(120, 120, 160, 160, 1);    // see slc.h for details
3     while(1)
4           if(scan_ts(&location[0]))     // unsigned int location[2]
5           {
6                 // did we see a touch?
7                 if((location[0]<left)&&(location[0]>right)&&
8                    (location[1]<top)&&(location[1]>bottom))
9                 {
10                      // is it inside our button boundaries?
11                      execute_task();
12                }
13          }
```

Let's review what we have so far. On line **1** and **2**, we are setting up the visual of our user interface. On line **4**, we are waiting for a touch. Remember, if no touch is detected, **`scan_ts`** returns a zero, and we do nothing. If it returns a 1, then we know there was a touch and the location of the touch now resides in our **`location`** array. If we reach lines **7** and **8**, then we have seen a touch and we have the location, we just need to check if that location is inside our button. We check the location with the boundary values we got earlier. If the location is inside the boundaries, we can execute a task.

We now have a working user interface. In subsequent sections we can build on what we have here to create interfaces with an arbitrary amount of buttons of any size. We will also look to create a cleaner, more efficient way to handle buttons.

3.3 *Button structures*

At the end of the last section, we had created a working user interface. It uses only one button of a fixed size. What happens as the complexity of the application increases? It demands an efficient way to handle and monitor buttons. Since we know each button will have to contain the same information, let's create a structure to handle the values associate with the button behavior. This will allow us to create multiple buttons, while keeping the code organized.

```
typedef struct {
      // Pixel location of button
      unsigned int c1;  // 'x' value of upper left corner
```

```
        unsigned int r1;  // 'y' value of upper left corner
        unsigned int c2;  // 'x' value of lower right corner
        unsigned int r2;  // 'y' value of lower right corner
        // Touch value location
        unsigned int c3;  // touch value of left
        unsigned int r3;  // touch value of top
        unsigned int c4;  // touch value of right
        unsigned int r4;  // touch value of bottom

        // Miscellaneous fields
        unsigned char fill_color;  // color of button
        unsigned char state;       // ON/OFF?
} BTTN;
```

Now we can create multiple buttons with multiple instances of the BTTN structure. It will also allow us to create functions that modify and handle all aspects of the BTTN structure. The structure contains a few extra fields whose purpose should be clear.

3.4 *Button Functions*

By using a structure to handle all the values needed for a button, we can now design functions to handle the buttons.

Let us start by creating a function that will draw the button for us. This will replace line **2** of the piece code in Section 3.2

```
void put_bttn(BTTN * b)
{
        // Assumes bottom right corner is (0,0), top left (320,240)
        // and that (c1,r1) is top left corner of button

        drawRect(b->c2, b->r2, b->c1, b->r1, b->fill_color);
}
```

This function may seem overly simple, but will save time later when we want to increase the capabilities of our buttons. Let's continue by taking line **7** and **8** from section 3.2 and create a function to do this for us.

```
unsigned char push_bttn(BTTN * b, unsigned int * location)
{
  if((*location<(b->c3)) && (*location>(b->c4))&&
     (*(location+1)<(b->r3)) && (*(location+1)>(b->r4)))
   {
```

```
            b->state = ~b->state;       // detected a push on button 'b'
            return 1;
      }
      return 0;       // the passed-in location was NOT inside the button
}
```

Now we can simply call **push_bttn** with the location array and the button we want to check. If **push_bttn** returns a '1', then the touch was inside that button. If a '0' is returned, then the touch was not inside that button and we should check another button. This process can be repeated until **push_bttn** returns a '1', or we have checked all of our buttons.

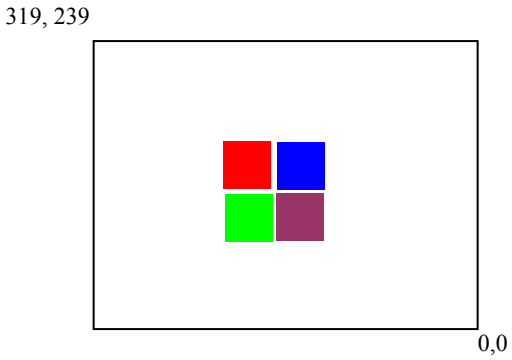
Let's re-write our piece code for the simple user interface using our newly created structure and functions.

```
BTTN b1;                  // create an instance of the BTTN structure
b1.c1 = 160;              // left column value
b1.r1 = 160;              // top row value
b1.c2 = 120;              // right column value
b1.r2 = 120;              // bottom row value
b1.fill_color = 1;        // default, 1-blue
b1.state = 0;
b1.c3 = left;             // touch screen boundaries for button
b1.r3 = top;              // created in Section 3.2
b1.c4 = right;
b1.r4 = bottom;
      ...
      ...

put_bttn(&b1);            // pointer to 'b1'

while(1)
      if(scan_ts(&location[0]))     // did we see a touch
      {
            // was the touch in our button?
            if(push_bttn(&b1,&location[0]))

                  // if yes, perform task assigned to that button
                  execute_task();
      }
```

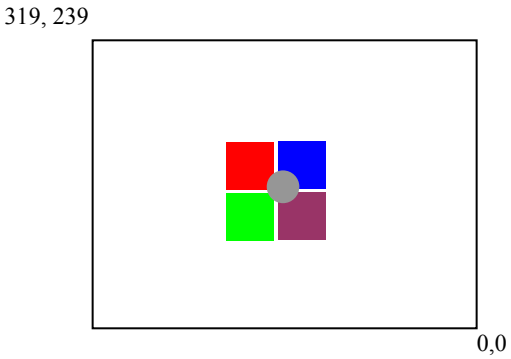
It becomes clear that, while using out structures and functions requires a bit more initial setup, we are well prepared to handle applications of several buttons. In the above example, if multiple buttons were used, adding a simple **for** loop would be all that was needed.

3.5 *Buttons in close proximity*

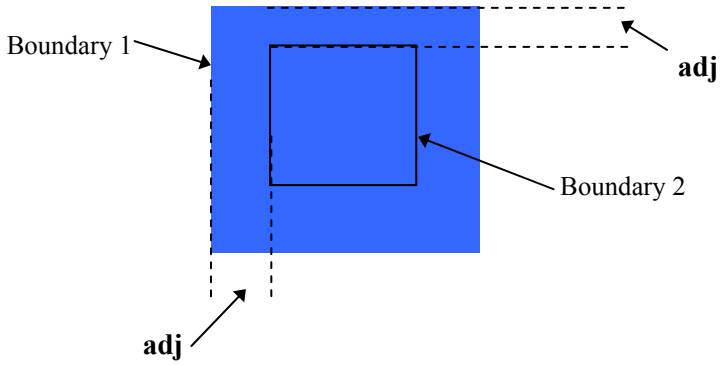
When creating an interface which requires multiple buttons, additional factors come into play. Consider the following interface:



Assume that each colored box is a button we have created. What happens if the user does not touch in the middle of the desired button, but on the boundary between two buttons?



For a more consistent and reliable interface, the application should be prepared to handle the situation above. How should the application interpret the touch if the gray circle represents the user's finger? For most applications, it is reasonable to say it would be better if the above touch was ignored than interpreted incorrectly. Remember, the touch screen is analog. It is obvious that the user probably wanted to touch the blue button because most of the finger is in the blue square. But since there is always a small amount of error, touching the gray circle above could produce a touch value that could fall with the touch boundaries of any of the four buttons, which makes a touch on the gray circle dangerous. Let's introduce an additional technique to help solve the above situation. Assume the diagram below shows an enlarged view of the blue button seen above. If we introduce a margin at the edges of the button, we can avoid getting an incorrect response from the touch screen.



In the above diagram, let the outside of the blue area be called Boundary 1 and the black box inside be called Boundary 2. Boundary 1 is the physical part of the button that the user sees and is defined by 'c1', 'r1', 'c2', and 'r2' of the BTTN structure. The fields 'c3', 'r3', 'c4', and 'r4' define the touch values read at Boundary 1. Let's redefine the valid touch area by reducing Boundary 1 by an amount known as **adj**. The resulting area is Boundary 2. The touch values of Boundary 2 can be expressed as:

```
c3 - adj
r3 - adj
c4 + adj
r4 + adj
```

By creating this margin, a touch near, or on the edge of the button is ignored because the new touch area (Boundary 2) is reduced. The **adj** value can be adjusted by the application designer based on desired performance. In general, as the **adj** value is increased (valid touch area decreases), the likelihood of touches being ignored is also increased, yet incorrect touches are reduced. As the value of **adj** is reduced (valid touch areas approaches Boundary 1) there are fewer touches ignore, but the chances of having an incorrect touch increase. The **adj** technique can be incorporated easily by making a small change to the **push_bttn** function and the BTTN structure.

```
// Added to the BTTN structure
int adj;
```

```
unsigned char push_bttn(BTTN * b, unsigned int * location)
{
  if((*location<(b->c3-b->adj)) && (*location>(b->c4+b->adj))&&
     (*(location+1)<(b->r3-b->adj)) && (*(location+1)>(b->r4+b->adj)))
   {
        // detected a push on button 'b'
        b->state = ~b->state;       // change state of button
        return 1;
```

```
    }
    return 0;
}
```

The bold text above was added to the function to account for the **adj** value. Each button can have its own margin, and can be customized depending on the proximity to other buttons, button size, etc. A few comments about the **adj** value:

(1) It is of type "int". This enables the **adj** value to be negative. This is useful in some situations, for example, when a button is relatively small, yet far away from other buttons. The **adj** value could be negative. This makes the valid touch area for that button larger than Boundary 1. It helps ensure that the user will always have accurate performance with that button, but without the risk of an incorrect touch because the button is far away from other buttons.

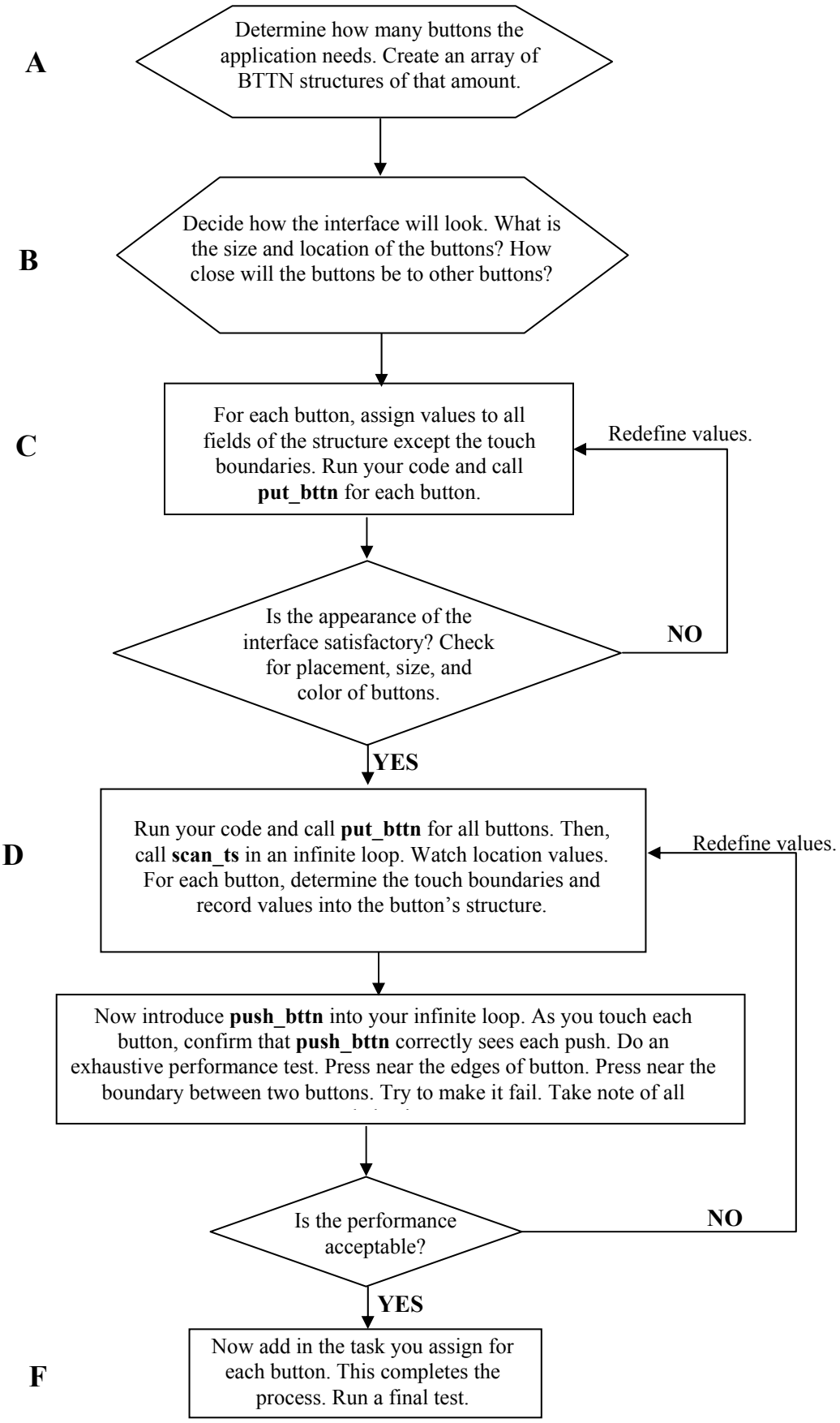
(2) When **adj** is positive, be cautious as the value increases. You never want either of the following equations to be true:

```
adj >= (c3-c4)/2  OR
adj >= (r3-r4)/2.
```

If either becomes true, it means that the valid touch area (Boundary 2) has become negative. A negative Boundary 2 will never allow a valid touch to be seen on that button. If desired, extra code could be written into **put_bttn** or **push_bttn** to check for this situation.

3.6 *Final Notes*

Creating a user interface with the SmartLCD requires a high level of understanding about the touch screen behavior. The greater the understanding, the better the user interface. Below is a flow chart which shows the process flow to create buttons and incorporate them into a user interface. A few steps have critical importance. Step D creates the touch values that define the touch boundaries for each button. This step is crucial in that the designer should try to anticipate how different touch styles and variations can affect the touch values at the boundaries. This means that one touch style can produce a certain set of values for the touch boundary (during the design process). But what if the end user has a contrasting touch style? This difference must be anticipated as much as possible in attempt to create the most robust interface possible. As buttons become closer together, the effect of different touch styles becomes harder to overcome. Step E is also crucial. It is where the designer must test the values generated in Step D.


A
Determine how many buttons the application needs. Create an array of BTTN structures of that amount.
B
Decide how the interface will look. What is the size and location of the buttons? How close will the buttons be to other buttons?
C
For each button, assign values to all fields of the structure except the touch boundaries. Run your code and call **put_bttn** for each button.
Redefine values.
Is the appearance of the interface satisfactory? Check for placement, size, and color of buttons.
NO
YES
D
Run your code and call **put_bttn** for all buttons. Then, call **scan_ts** in an infinite loop. Watch location values. For each button, determine the touch boundaries and record values into the button's structure.
Redefine values.
E
Now introduce **push_bttn** into your infinite loop. As you touch each button, confirm that **push_bttn** correctly sees each push. Do an exhaustive performance test. Press near the edges of button. Press near the boundary between two buttons. Try to make it fail. Take note of all
Is the performance acceptable?
NO
YES
F
Now add in the task you assign for each button. This completes the process. Run a final test.