SLCD Application Note AN-100
Sample program for Rabbit / Zworld RCM-3720

Reach Technology, Inc.

February 10, 2005

This application note describes a sample / demonstration program for the Rabbit Semiconductor / Zworld RCM3720 evaluation kit connected to the Reach SLCD controller. It includes on-screen pushbuttons to light the LEDs on the demonstration boards and icons that change when the board buttons are pushed, as well as a digital panel meter, keyboard, and bar chart / graph example.

Reach Technology, Inc.
sales@reachtech.com
(503) 675-6464

1. Overview

This note describes a sample program written to demonstrate some of the capabilities of the SLCD controller, and to provide a code base to work from in developing custom applications. The sample illustrates how to send commands to the SLCD and how to field responses from it. The code is written in Dynamic C, which is provided with the Rabbit / Zworld evaluation kit.

Controlling the SLCD is fairly straightforward, but there are two areas that deserve description outside of the line-by-line code annotation. One is the SLCD transmit and receive interface which handles full duplex communication; the transmit and receive can have independent operations happening at the same time. The other is the overall structure of how to display pages and respond to button presses.

This example was written specifically for the RCM3720 prototype board that has certain LEDs and pushbuttons. It can be run on practically any other Rabbit core module as long as these ports are re-defined and the appropriate serial port is connected and the code changed to accommodate the different port.

It is assumed that the reader has a copy of the SLCD Manual to look up the format and syntax of the SLCD commands.

1.1. *SLCD communications interface*

The serial interface to the SLCD is full duplex - commands can be sent independently of return acknowledgements and button press responses can be received at any time once a button is defined on the screen.

This sample implements a circular receive buffer which collects characters from the Dynamic C internal interrupt driven buffer. This buffer is parsed for SLCD command prompts and button presses. Only one button press is allowed to be outstanding at any time; there is no queuing.

The `anySLCDrx()` routine handles the SLCD receive stream. It is called by the main program loop whenever command prompts and buttons need to be checked for. If multiple commands are allowed to be outstanding, then this routine is called whenever that limit is reached. This routine:

- checks for characters in the Dynamic C buffer and adds them to the circular buffer
- parses the buffer for SLCD command prompts (acknowledgements or errors)
- parses the buffer for button presses. If a button press response is found, the global button structure variable "btn" is updated.

Note that the SLCD supports software flow control, and instead of limiting the number of commands outstanding, the more general approach would be to implement true software flow control. This was not done in order to reduce complexity.

1.2. *Program structure*

Modern desktop programs use "windows" which are 1) generally smaller than the full screen, 2) can overlap, and3) have the concept of "focus" (the window with focus is the one that is currently in control). For a small display like the QVGA, and for the kind of control applications it is used for, multiple overlapping windows can be more confusing than helpful. Therefore the most useful interface style is one where each screen displays certain information and controls, and if different information needs to be displayed, a complete new screen is drawn.

The sample program uses this interface style, and implements a hierarchy of display/control screens. Each screen implementation has the same structure:

1. Display the page including control buttons

2. Loop waiting for button presses and while waiting do data acquisition and control processing and update display elements.

3. When done, return , and the calling page is re-displayed.

2. Annotated Code file

```
/*************************************************************
   SLCD Demo program for the RCM3720 kit

      !!!!! NOTE !!!!!
      WHILE RUNNING FROM THE DYNAMIC C ENVIRONMENT, YOU MUST RUN
      THIS PROGRAM IN "NO POLLING" MODE TO AVOID MISSING RECEIVE
      CHARACTERS AT 115,200 BAUD (DEFAULT) WHILE RUNNING THE
      CONTINUOUSLY UPDATING CHART DEMO
      !!!!! NOTE !!!!!


*************************************************************/

#define DEBUG
```

The Dynamic C debug environment polls the target for breakpoint and other status. This polling can cause receive characters to get missed at 115,200 baud. When an application has been debugged, run it using the "no polling" mode.

DEBUG flag -

The SLCD can accept more than one command at a time. This allows communication time and command execution time to overlap for fastest display speed. However, it can make debugging more difficult, so this define forces only one command to be executed at a time. Comment this out for fastest display speed in the real application once it is working correctly.

```
/*
   This demo assumes the following bitmaps are loaded into the
   SLCD in the order shown below. Bitmaps 01 thru 20 are reserved
   for the SLCD demo macros. This program uses bitmaps 21 through
   40. The text file for loading these bitmaps via the BMPload
   program is slcd_demo.lst

01_button.bmp
02_button_click.bmp
03_check_box.bmp
04_check_box_click.bmp
05_Hitachi_Logos.BMP
06_Hitachi_PanelInfo.BMP
07_SLCDInfo.BMP
08_Contact_Blank.BMP
09_logo_bounce.bmp
10_button_up.BMP
11_button_dn.BMP
```

Bitmaps 1 through 20 are reserved for the internal demo that is included with all SLCD kits. This sample program assumes that the bitmaps 21 through 40 as shown are loaded into the SLCD.

In the SLCD commands, bitmaps are referred to by index number instead of by name, so to keep the number and name consistent, bitmap files include their load index.

```
12_input_box.bmp
13_null.bmp
14_null.bmp
15_null.bmp
16_null.bmp
17_null.bmp
18_null.bmp
19_null.bmp
20_null.bmp
21_MainScrn.bmp
22_round_button.bmp
23_round_button_dn.bmp
24_check_box.bmp
25_check_box_click.bmp
26_LEDOFF.bmp
27_LEDON.bmp
28_board_sw_off.BMP
29_board_sw_on.BMP
30_big_button.BMP
31_big_button_dn.BMP
32_Bezel.bmp
33_volts_on.bmp
34_volts_off.bmp
35_mA_on.bmp
36_mA_off.bmp
37_button_on.bmp
38_button_off.bmp
39_button_up.bmp
40_button_dn.bmp

*/

// these defines are available for the bitmaps this demo uses
#define  BMP_MainScrn             21
#define  BMP_round_button         22
#define  BMP_round_button_dn      23
#define  BMP_check_box            24
#define  BMP_check_box_click      25
#define  BMP_LEDOFF               26
#define  BMP_LEDON                27
```

These defines are here mostly for reference; in most commands the actual bitmap number is used because #defines do not work inside text strings. The funcrion bmp() is defined at the end of this file which displays a bitmap by name.

```
#define  BMP_board_sw_off         28
#define  BMP_board_sw_on          29
#define  BMP_big_button           30
#define  BMP_big_button_dn        31
#define  BMP_Bezel                32
#define  BMP_volts_on             33
#define  BMP_volts_off            34
#define  BMP_mA_on                35
#define  BMP_mA_off               36
#define  BMP_button_on            37
#define  BMP_button_off           38
#define  BMP_button_up            39
#define  BMP_button_dn            40

#class auto

#define DS1 6     //port F bit 6
#define DS2 7     //port F bit 7
#define S1  4     //switch, port F bit 4
#define S2  7     //switch, port B bit 7

#define DINBUFSIZE  255
#define DOUTBUFSIZE 255



///////
// change serial baud rate here
///////
#ifndef _232BAUD
#define _232BAUD 115200
#endif
```

DS1 and DS2 are the port pins used to control LEDs on the RCM3720 prototype board. S1 and S2 are pushbutton switches.

This sample assumes the SLCD is connected to the RCM3720 prototype board via serial port D. To change serial ports, replace all “serD” function calls with “serX” where X is the port you want to use. Also replace the 'D' in these defines with the same 'X' as above.

Serial port D is the one that looks like a PC 3 wire port on a DB-9 connector when used with the standard Rabbit/Zworld grey flat serial cable.

This sets the serial port baud rate on the RCM3720. This cannot be changed without changing the baud rate on the SLCD.

```
119  //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
120  //  Set MAX_CMDS to the maximum number of commands outstanding
121  //  to the SLCD at any time.
122  //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
123  #ifdef DEBUG
124  #define MAX_CMDS 1
125  #else
126  #define MAX_CMDS 4
127  #endif
128
129  typedef unsigned char uchar;
130  typedef unsigned int uint;
131
132  struct button {
133        uchar index;
134        uchar state;
135  };
136  // only one button active at a time, so this can be global
137  struct button btn;
138
139  void slcd(char *s );
140  uchar anySLCDrx(struct button *btn);
141  void sRxInit(void);
142  void protoboardDemo(void);
143  void pbLedOn(int led);
144  void pbLedOff(int led);
145  void slcdDemo1(void);
146  void slcdDemo2(void);
147  void bmp(uint bitmap, uint x, uint y);
148  void pauseMs(int delay);
149  void moreDemos(void);
150  void levelGraph(void);
151  void drawLevelBar( uchar index, int x, int y );
152
153
154  // count of commands outstanding
155  static unsigned char cmdCount;
156
```

This sets the maximum number of commands queued in the SLCD. Used in the slcd() function. When only one command is outstanding, the command that caused it is the last one sent.

Useful typedefs

The button structure holds the most recently pushed button information. It doesn't need to be a global but it doesn't hurt since button pushes are not queued.

All functions are prototyped up front so they can be used before they are defined.

This tracks the number of commands outstanding (queued) to the SLCD.

```
main()
{
   auto int testing;

   brdInit();                    //initialize board for this demo

   BitWrPortI(PEDR, &PEDRShadow, 0, 5);   //set low to enable rs232 device

   serDopen(_232BAUD);
   serDwrFlush();
   serDrdFlush();
   sRxInit();
   cmdCount = 0;

   // clear any junk that may be in the SLCD's input buffer
   serDputc('\r');      // send <return>
   pauseMs(100);        // wait for any responses
   serDrdFlush();       // flush



   // verify we are connected...
   testing = 1;

   while( testing )
   {
      slcd("");                  // send <return>
      pauseMs(10);               // wait for response
      anySLCDrx(&btn);           // process returns
      if( cmdCount == 0 )
      {
         // got a prompt
         testing = 0;
      }
      else
         cmdCount = 0;
   }
```

Program execution begins here…

Initialize serial port D

Initialize receive circular buffer, and count of commands outstanding to the SLCD.

When attaching a serial port, or when switching serial modes in a Rabbit module, sometimes miscellaneous characters end up sent on the port connected to the SLCD. Sending a null command (<return> character) which will flush any outstanding characters. We don't care about the response as it was not a command we sent.

This routine represents a general purpose mechanism for determining that an SLCD unit is in fact connected to the serial port. A null command (<return> character) is sent, which increments the global cmdCount variable. The anySLCDrx() routine will decrement cmdCount for each SLCD prompt received. If cmdCount is not zero (no response), then it is cleared and the test keeps running. If there was anything else to do, the delay could be replaced by that functionality.

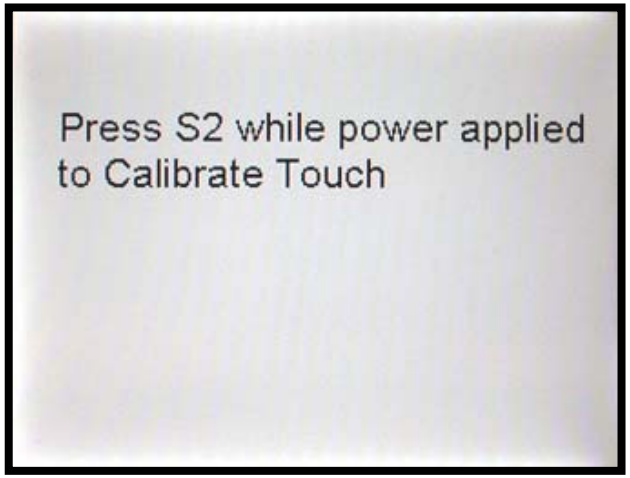
The slcd() routine sends a command to the SLCD and implements the command counter and enforces the maximum commands outstanding limit. The anySLCDrx routine looks for command acknowledgements and button presses. It decrements cmdCount when an SLCD command acknowledge has been received.

The first screen notifies the user that the touch can be calibrated by pressing the prototype pushbutton S2 on power-on. It is very useful to have a hardware-invoked touch calibration so that the average user cannot recalibrate (and mess up) the touch screen.

As this is the first time commands are sent to the SLCD, it must be reset in case it was left in a different state than what is desired. That is the purpose of the first three commands - set color, clear screen, and set font.

```
// calibration screen
slcd("s 0 1"); // set color
slcd("z");  // clear screen
slcd("f24");   // set font
// send the text - use a backslash " to embed a quote inside a string. This is a 'C' convention
slcd("t \"Press S2 while power applied\nto Calibrate Touch\" 25 50");
pauseMs(1000);
```

Screen now displays this →



```
211      // check for touch calibration request - S2 pushed on power-on
212      if( !BitRdPortI(PBDR, S2) )
213      {
214         pauseMs(20);  // debounce
215         if( !BitRdPortI(PBDR, S2) )
216         {
217            slcd("z");
218               slcd("t \"Touch calibration requested\n(S2 pushed at power-on)\" 25 50");
219            pauseMs(500);    // let user see message
220               slcd("tc");
221            anySLCDrx(&btn); // process accumulated returns
222         }
223      }
224
225      while (1)
226      {
227         //
228         // Logo splash screen
229         //
230         slcd("s 0 1");
231         slcd("z");
232         slcd("o 0 0");
233         slcd("xi 21 0 0");
234         pauseMs(2000);  // let user see screen for 2 seconds
```

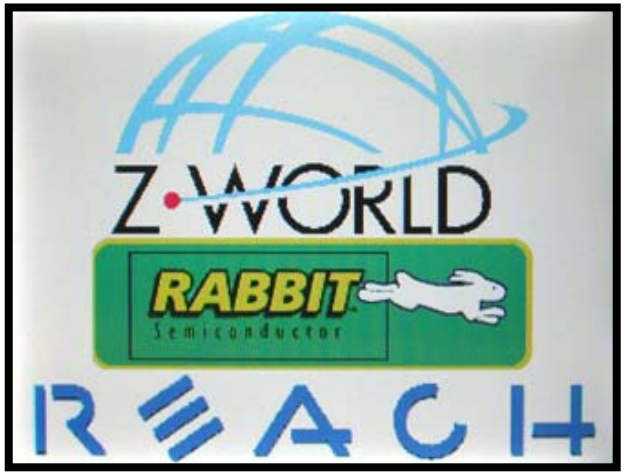
Check for the button press and invoke the touch calibration routine ("tc") if needed.

Main program - never stops.

Display the Rabbit/Zworld/Reach splash screen. Note that we use the index of the main screen bitmap, 21. See Appendix A for the bitmap images.

(To use the bitmap #defines and display bitmaps by name instead of number, you can use the function bmp() defined at the end of this file.)

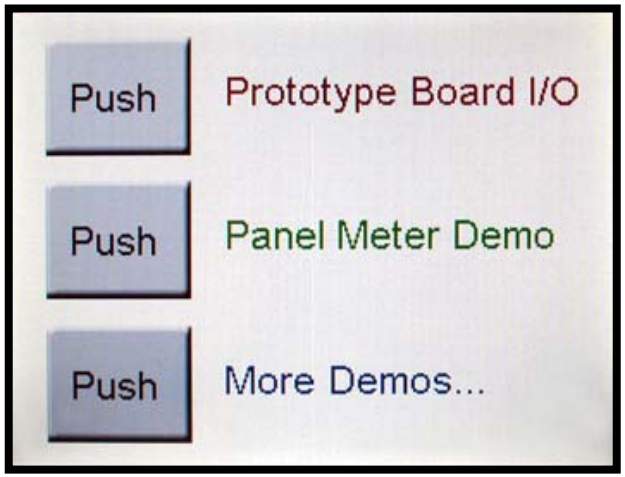
Screen now displays this →



```
235         slcd("z");
236         slcd("f24");
237         slcd("o 20 14");
238         slcd("s 0 1");
239         slcd("bd 1 0 0 1 \"Push\" 12 20 30 31");
240         slcd("S 900 FFF");      // dark red
241         slcd("t \"Prototype Board I/O\" 94 17");
242         slcd("o 20 89");
243         slcd("s 0 1");
244         slcd("bd 2 0 0 1 \"Push\" 12 20 30 31");
245         slcd("S 090 FFF");      // dark green
246         slcd("t \"Panel Meter Demo\" 94 17");
247         slcd("o 20 165");
248         slcd("s 0 1");
249         slcd("bd 3 0 0 1 \"Push\" 12 20 30 31");
250         slcd("S 009 FFF");      // dark blue
251         slcd("t \"More Demos...\" 94 17");
252
```

This is the setup for the first screen which just presents choices for the next page. It is very basic and just has three buttons. All visual elements are drawn here.

Screen now displays this →



```
253         // wait for a button
254         while( !anySLCDrx(&btn) );
255
256
257         switch( btn.index )
258         {
259            case 1:
260               protoboardDemo();
261               break;
262            case 2:
263               slcdDemo1();
264               break;
265            case 3:
266               moreDemos();
267               break;
268            default:
269               break;
270         }
271      }
272   }
273
```

Once the screen is drawn, we don't have anything to do except wait for a button.

The btn structure contains the index and state of the button that was pressed. The first page only has momentary (stateless) buttons, and each invokes the next level routine.

When the routine finishes, we go back to the top of the infinite loop, display the splash screen and the main choice screen again.

This implements the second choice screen which is similar to the first.

```
274  void moreDemos(void)
275  {
276     slcd("z");
277     slcd("f24");
278     slcd("o 20 14");
279     slcd("s 0 1");
280     slcd("bd 1 0 0 1 \"Push\" 12 20 30 31");
281     slcd("S 900 FFF");   // dark red
282     slcd("t \"Keyboard Demo\" 94 17");
283     slcd("o 20 89");
284     slcd("s 0 1");
285     slcd("bd 2 0 0 1 \"Push\" 12 20 30 31");
286     slcd("S 090 FFF");   // dark green
287     slcd("t \"Level / Graph Demo\" 94 17");
288
289     // wait for a button
290     while( !anySLCDrx(&btn) );
291
292     switch( btn.index )
293     {
294        case 1:
295           slcdDemo2();
296           break;
297
298        case 2:
299           levelGraph();
300           break;
301
302        default:
303           break;
304     }
305  }
306
```

Second choice page invokes the keyboard demo or the levelGraph demo. When either of these returns, this function returns and we go back to the first choice screen.

```
307  void drawLevelBar( uchar index, int x, int y )
308  {
309    char buf[100];
310
311     sprintf(buf,"o %d %d",x,y);
312     slcd(buf);
313     sprintf(buf,"ld %d 0 0 20 80 0 0 1 888 100 F00 65 FF0 50 0F0",index);
314     slcd(buf);
315     slcd("p 2");
316     slcd("l 21 0 21 81");
317     slcd("l 0 81 21 81");
318     sprintf(buf,"lv %d 0", index);
319     slcd(buf);
320  }
```

This draws a "levelbar" (single element of a bar chart) with a given index at location x, y. Note the use of the sprintf function to create a text string to be sent to the SLCD.
The SET ORIGIN command is used to change the location of the drawn bar.

Here a line is drawn as a "shadow" for the levelbar. This makes it stand out a little better.

```
321  void levelGraph(void)
322  {
323    char buf[100];
324     float f;
325     int levelVal[5];
326     int chartVal[3];
327     int i, j;
328
329
330     // draw 5 levelbars
331     slcd("s 0 1");
332     slcd("z");
333     slcd("f 24BC");
334     slcd("t \"Data visualization charts\" 28 0");
335     drawLevelBar(0, 43, 40);
336     drawLevelBar(1, 96, 40);
337     drawLevelBar(2, 149, 40);
338     drawLevelBar(3, 202, 40);
339     drawLevelBar(4, 255, 40);
340
341     // draw long chart
342     slcd("o 10 140");
343     slcd("cd 0 0 0 301 49 1 3 1 100 008 2 F00 2 0F0 2 FFF");
344     slcd("p 2");
345     slcd("l 302 0 302 50");
346     slcd("l 0 50 302 50");
347     slcd("o 0 0");
348
349     // next button
350     slcd("f13B");
351     slcd("bd 1 257 202 1 \"Next\" 16 11 22 23");
352
353
354
```

This routine implements the levelbar and chart example screen.

As with all screens, we start by drawing the visual elements

Put a line on 2 sides of the chart to make it stand out.

This defines the button that moves us to the next screen.

```
355      while(1)
356      {
357         // while waiting for button press, generate data and update display
358         while( !anySLCDrx(&btn) )
359         {
360            // this costate does the data acquisition or in our case, simulation
361            costate
362            {
363               // generate data for 5 levelbars
364               for(i=0;i<5;i++)
365               {
366                  f = rand();
367                  levelVal[i] = (int)( 10+ f * 90);
368               }
369
370               // generate data for chart
371               f = rand();
372               chartVal[0] = (int)(4 + f * 25);
373               f = rand();
374               chartVal[1] = (int)(35 + f * 25);
375               f = rand();
376               chartVal[2] = (int)(70 + f * 25);
377
378               // use a delay to reduce update rate
379               waitfor(DelayMs(100));
380            }
381
382            // this costate displays the levelbars with its own update rate
383            costate
384            {
385               for(i=0;i<5;i++)
386               {
387                  // update level bars
388                  sprintf(buf,"lv %d %d",i,levelVal[i]);
389                  slcd(buf);
390               }
391
392               waitfor(DelayMs(250));
393            }
394
```

Until the next button is pressed which ends this routine, we keep displaying the charts and graph.

The Dynamic C costate functionality simplifies this loop by separating the data generation and display sections. In standard C this would be done with a discrete timer variables and a state machine.

We use the random number generator for data - PLACE YOUR APPLICATION CODE HERE to acquire real data to be displayed.

This costate displays the levelbar data with its own update rate 4 per second.

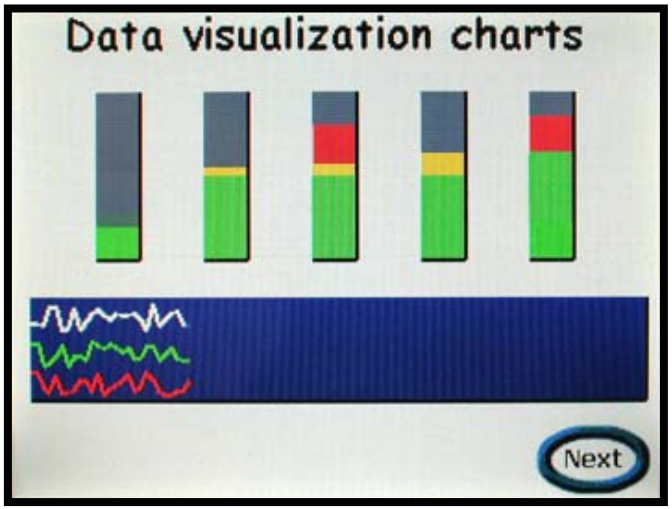
```
395             // this costate displays the chart with its own update rate
396             costate
397             {
398                // update chart
399                sprintf(buf,"cv 0 %d %d %d",chartVal[0], chartVal[1], chartVal[2]);
400                slcd(buf);
401
402                waitfor(DelayMs(100));
403             }
404          }
405
406       switch( btn.index )
407       {
408          case 1:  // Next
409                return;
410             break;
411
412             default:
413             printf("Internal error - invalid button #%d\n",btn.index);
414             break;
415       }
416       }
417    }
418
```

This costate displays the chart data with its own update rate 10 per second.

A button has been pressed - handle it

Only button is Next - return to the calling page (the main selection page).

Screen displays continuously updated levelbars and chart



This is the keyboard demo routine.

```
#define MAX_CHARS 20
void slcdDemo2(void)
{
   int i;
   char cmd[80];
   char input[MAX_CHARS+1];
   uchar inputIndex;
   char c;

   // where to put incoming characters
   inputIndex = 0;
   // clear input so there is a valid end of string everywhere
   for(i=0;i<MAX_CHARS+1;i++)
      input[i] = 0;

   slcd("s 0 1");
   slcd("z");
   slcd("o 0 0");
   slcd("f24");
   slcd("t \"Keyboard Demo\" 70 10");
   // draw alpha keyboard
   // first row
   slcd("bd 49   0 112 1 \"1\" 8 5 39 40");
   slcd("bd 50  32 112 1 \"2\" 8 5 39 40");
   slcd("bd 51  64 112 1 \"3\" 8 5 39 40");
   slcd("bd 52  96 112 1 \"4\" 8 5 39 40");
   slcd("bd 53 128 112 1 \"5\" 8 5 39 40");
   slcd("bd 54 160 112 1 \"6\" 8 5 39 40");
   slcd("bd 55 192 112 1 \"7\" 8 5 39 40");
   slcd("bd 56 224 112 1 \"8\" 8 5 39 40");
   slcd("bd 57 256 112 1 \"9\" 8 5 39 40");
   slcd("bd 48 288 112 1 \"0\" 8 5 39 40");
   // second row
   slcd("bd 81   0 144 1 \"Q\" 8 5 39 40");
   slcd("bd 87  32 144 1 \"W\" 6 5 39 40");
   slcd("bd 69  64 144 1 \"E\" 9 5 39 40");
   slcd("bd 82  96 144 1 \"R\" 8 5 39 40");
   slcd("bd 84 128 144 1 \"T\" 9 5 39 40");
   slcd("bd 89 160 144 1 \"Y\" 9 5 39 40");
   slcd("bd 85 192 144 1 \"U\" 8 5 39 40");
```

Display the keyboard. Note that we use the decimal value of the key as the button index. This makes processing the key presses simple.

```
459      slcd("bd 73 224 144 1 \"I\" 11 5 39 40");
460      slcd("bd 79 256 144 1 \"O\" 8 5 39 40");
461      slcd("bd 80 288 144 1 \"P\" 9 5 39 40");
462      // third row
463      slcd("bd 65   0 176 1 \"A\" 9 5 39 40");
464      slcd("bd 83  32 176 1 \"S\" 8 5 39 40");
465      slcd("bd 68  64 176 1 \"D\" 9 5 39 40");
466      slcd("bd 70  96 176 1 \"F\" 9 5 39 40");
467      slcd("bd 71 128 176 1 \"G\" 8 5 39 40");
468      slcd("bd 72 160 176 1 \"H\" 8 5 39 40");
469      slcd("bd 74 192 176 1 \"J\" 9 5 39 40");
470      slcd("bd 75 224 176 1 \"K\" 9 5 39 40");
471      slcd("bd 76 256 176 1 \"L\" 9 5 39 40");
472      slcd("f 16B");
473      slcd("bd 32 288 176 1 \"sp\" 5 5 39 40");
474      slcd("f 24");
475      // fourth row
476      slcd("bd 90   0 208 1 \"Z\" 9 5 39 40");
477      slcd("bd 88  32 208 1 \"X\" 9 5 39 40");
478      slcd("bd 67  64 208 1 \"C\" 8 5 39 40");
479      slcd("bd 86  96 208 1 \"V\" 9 5 39 40");
480      slcd("bd 66 128 208 1 \"B\" 8 5 39 40");
481      slcd("bd 78 160 208 1 \"N\" 8 5 39 40");
482      slcd("bd 77 192 208 1 \"M\" 8 5 39 40");
483      slcd("bd 44 224 208 1 \",\" 10 3 39 40");
484      slcd("bd 45 256 208 1 \"-\" 10 5 39 40");
485      // rubout == backspace
486      slcd("f 16B");
487      slcd("bd 8  288 208 1 \"rub\" 4 9 39 40");
488
489      // next button
490      slcd("f13B");
491      slcd("bd 3 257 0 1 \"Next\" 16 11 22 23");
492
493      // font for keystroke string
494      slcd("f 12x24");
495      // underscore for cursor
496      slcd("t \"_\" 60 75");
497
498
```

This sets the font and the initial cursor for the displayed character string

The cursor is just the underscore character at the end of the string.

Screen now displays this →


```
499
500   while(1)
501      {
502         // wait for button press
503         while( !anySLCDrx(&btn) )
504            ;
505
506         switch( btn.index )
507         {
508            case 8:  // RUBOUT (destructive backspace)
509               if( inputIndex >= 1 )
510                  input[--inputIndex] = 0;
511               break;
512
513            case 3:  // NEXT
514
515               return;
516
517            default: // alpha-numeric keystroke
518               input[inputIndex++] = btn.index;
519               if( inputIndex == MAX_CHARS )
520                  inputIndex--;
521               break;
522         }
```

In this sample screen, there is nothing to do but wait for a button press.

Handle destructive backspace by moving the end of the string back one.

Next button - return to called screen routine

This is the easy part - each letter button has its ASCII character as its index, so no translation is needed.

```
523         // blank the line for rubout
524         slcd("sc 50 75");
525         for(i=0;i<MAX_CHARS;i++)
526            slcd("t \" \"");
527
528         // display stored string
529         slcd("sc 0 0");
530         sprintf(cmd,"t \"%s\" 50 75",input);
531         slcd(cmd);
532         // display underscore as "cursor"
533         slcd("t \"_\"");
534
535      }
536
537   }
```

After a key has been processed, clear the line and display the entire string. This is not particularly elegant, but for a small screen and a small string, it works fine.

```
538  void slcdDemo1(void)
539  {
540     char cmd[80];
541     float f;
542     enum {
543        m_volts,
544        m_mAmps,
545     } meterState;
546     float volts, mAmps;
547     char input;
548
549     // background slightly off white helps buttons stand out
550     slcd("S 000 EEE");
551     slcd("z");
552     slcd("o 0 0");
553     bmp(BMP_Bezel, 29, 16);
554     // volts selected
555     meterState = m_volts;
556     slcd("bd 1 258 24 21 \"\" \"\" 0 0 0 0 34 33");
557     // amps deselected
558     slcd("bd 2 258 72 20 \"\" \"\" 0 0 0 0 36 35");
559     // input select
560     slcd("f24");
561     slcd("t \"Input:\" 30 138");
562     // input 1 selected
563     input = 1;
564     slcd("bd 11 107 129 21 \"1\" \"1\" 9 5 9 5 39 40");
565     // input 2, 3 deselected
566     slcd("bd 12 151 129 20 \"2\" \"2\" 9 5 9 5 39 40");
567     slcd("bd 13 195 129 20 \"3\" \"3\" 9 5 9 5 39 40");
568     // next button
569     slcd("f13B");
570     slcd("bd 3 257 202 1 \"Next\" 16 11 22 23");
571
572
```

This demo implements a digital panel meter. Latching (statefull) buttons are used to select "volts" vs "mAmps" and the input source. This example shows how to handle simultaneously acquiring data and displaying it. In our case, the acquisition is faked with a random number generator.

These button define commands define latching buttons where button 1 (volts) is "on" when initialized and button 2 (mA) is "off".

These do the same as above, but for the input select buttons

```
573      while(1)                    Endless loop - returns (exits) when the "Next" button is pressed
574      {
575         // while waiting for button press, acquire data and update meter
576         while( !anySLCDrx(&btn) )
577         {
578            // this costate does the data acquisition
579            costate
580            {
581                  //
582                  // measure and filter voltage here...
583                  //
584                  if( meterState == m_volts )
585                  {
586                     f = rand();
587                     switch(input)                    Here we fake nominal 24V for input 1 and 12V for
588                     {                                input 2. PLACE YOUR APPLICATION'S
589                           case 1:                    ACQUISITION CODE HERE
590                                 //fake 24V +/- 3V
591                                 volts = 24 + (f*6)-3;
592                           break;
593
594                        case 2:
595                           // fake 12V +/- 1V
596                           volts = 12 + (f*2)-1;
597                           break;
598
599                        case 3:
600                        default:
601                           // fake 0
602                           volts = 0;
603                           break;
604                     }
605                     // use a delay to reduce update rate
606                     waitfor(DelayMs(250));
607                  }
608                  else
609                  if( meterState == m_mAmps )
610                  {
611                     f = rand();
612                     switch(input)
```

```
613                  {
614                        case 1:
615                        // fake 500mA +/- 100
616                              mAmps = 500 + (f*200)-100;
617                        break;
618
619                     case 2:
620                        // fake 150mA +/- 50;
621                        mAmps = 150 + (f*100) -50;
622                        break;
623
624                     case 3:
625                     default:
626                        mAmps = 0;
627                        break;
628                  }
629                  // use a delay to reduce update rate
630                  waitfor(DelayMs(250));
631            }
632      }
633
634      // this costate displays the meter value with its own update rate
635      costate
636      {
637            if( meterState == m_volts )
638            {
639               slcd("f32x64");
640               slcd("S F00 000");
641               sprintf(cmd,"t \"%5.2f\" 54 43",volts);
642               slcd(cmd);
643               // use a delay to set update rate
644               waitfor(DelayMs(400));
645            }
646            else
647            if( meterState == m_mAmps )
648            {
649               slcd("f32x64");
650               slcd("S F00 000");
651               sprintf(cmd,"t \"%5.1f\" 54 43",mAmps);
652               slcd(cmd);
```

PLACE YOUR APPLICATION'S ACQUISITION CODE HERE

The display update rate that can be supported depends on the display. The TFT (active) display can be updated much faster than the STN (passive) display. This is set for the passive display. The update rate is determined empirically. Note that the passive display turn on and turn off time varies with temperature. You could use the SLCD's "temp" command to change the update rate with temperature...

```
653                  // use a delay to set update rate
654                  waitfor(DelayMs(500));
655               }
656
657         }
658      }
659
660      switch( btn.index )
661      {
662         case 1:  // VOLTS
663            if(btn.state == 1)      // selected
664            {
665                  // set amps off
666               slcd("ssb 2 0");
667               meterState = m_volts;
668            }
669               else
670            {
671                  // set amps on
672               slcd("ssb 2 1");
673               meterState = m_mAmps;
674            }
675            break;
676
677         case 2:  // AMPS
678            if(btn.state == 1)  // selected
679            {
680                  // set volts off
681               slcd("ssb 1 0");
682               meterState = m_mAmps;
683            }
684               else
685            {
686                  // set volts on
687               slcd("ssb 1 1");
688               meterState = m_volts;
689            }
690            break;
691
692         case 3:  // NEXT
```

Handle ganged volts and mAmps button presses. Both buttons are latching, so they could inherently be both up or both down and maintain their state. This code forces the "gang" in that if one is down, the other is up and vice versa.

when 1 is in the "down" state, turn 2 off

and when 1 is in the "up" state, push the other one down

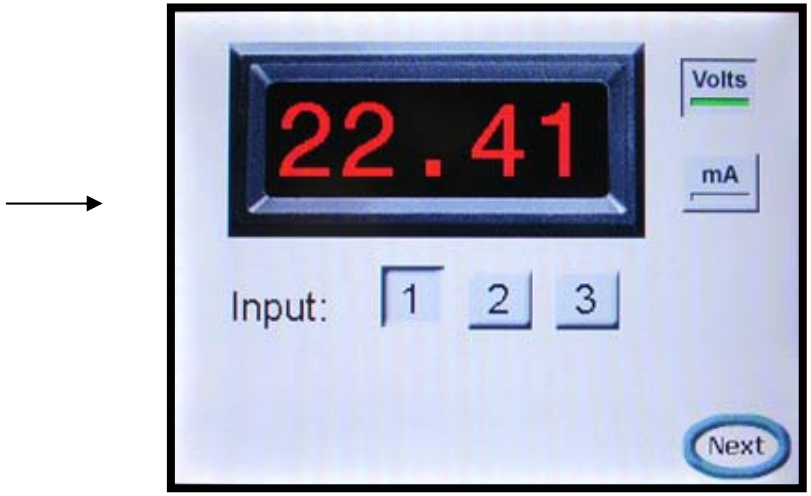
The "Next" button breaks out of the infinite while(1) loop by returning to the calling routine (display page code)

```
693             return;
694
695          case 11: // input 1
696             if(btn.state == 1)      // selected
697             {
698                input = 1;
699                // turn others off
700                slcd("ssb 12 0");
701                slcd("ssb 13 0");
702             }
703             else
704                input = 0;
705             break;
706
707          case 12: // input 2
708             if(btn.state == 1)      // selected
709             {
710                input = 2;
711             // turn others off
712                slcd("ssb 11 0");
713                slcd("ssb 13 0");
714             }
715             else
716                input = 0;
717             break;
718
719          case 13: // input 3
720             if(btn.state == 1)      // selected
721             {
722                input = 3;
723             // turn others off
724                slcd("ssb 11 0");
725                slcd("ssb 12 0");
726             }
727             else
728                input = 0;
729             break;
730
731
732          default:
```

This implements the ganged 3 button input selector. If all three buttons are in the up (state 0) position, the input is set to 0 which is logical - when nothing is selected the display shows 0.

```
733                 printf("Internal error - invalid button #%d\n",btn.index);
734                 break;
735             }
736         }
737
738     }
739
```

Screen looks something like this (value is random) after being drawn and updated but no buttons have been pushed yet

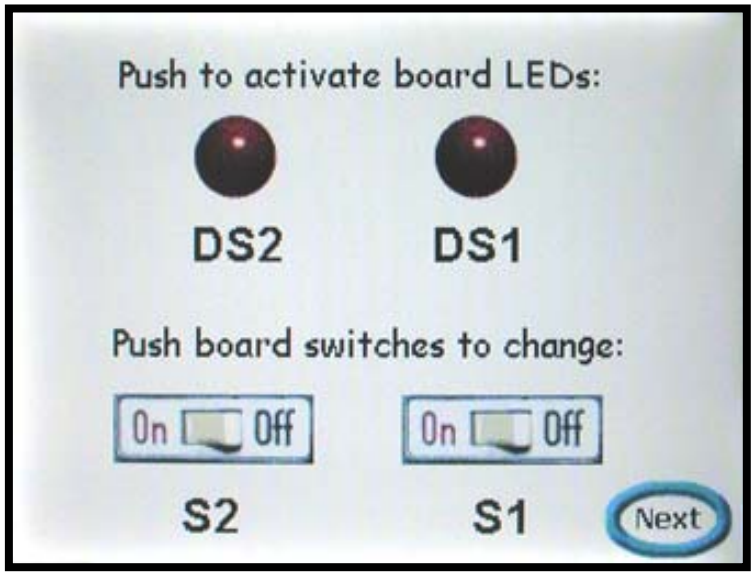


```
740  void protoboardDemo(void)
741  {
742     struct button btn;
743     uchar s1, s2;
744     uchar s1Image, s2Image;
745
746     // switches are "off"
747     s1 = s2 = 0;
748     // images are off
749     s1Image = s2Image = 0;
750
751     slcd("z");
752     slcd("s 0 1");
753     slcd("f18BC");
754     slcd("t \"Push to activate board LEDs:\" 47 19");
755     slcd("bd 2 70 35 2 \"\" \"\" 0 0 0 0 26 27");
756     slcd("f24B");
757     slcd("t \"DS2\" 79 90");
758     slcd("t \"DS1\" 183 90");
759     slcd("f18BC");
760     slcd("bd 1 174 35 2 \"\" \"\" 0 0 0 0 26 27");
761     slcd("t \"Push board switches to change:\" 45 135");
762     slcd("xi 28 46 166");
763     slcd("xi 28 171 166");
764     slcd("f24B");
765     slcd("t \"S2\" 75 207");
766     slcd("t \"S1\" 201 207");
767     slcd("f13B");
768     slcd("bd 3 257 202 1 \"Next\" 16 11 22 23");
769
```

This function implements the demo that allows screen buttons to control LEDs on the prototype board, and prototype board pushbuttons to control displayed bitmaps.

Display the screen and define buttons

Screen now displays this →

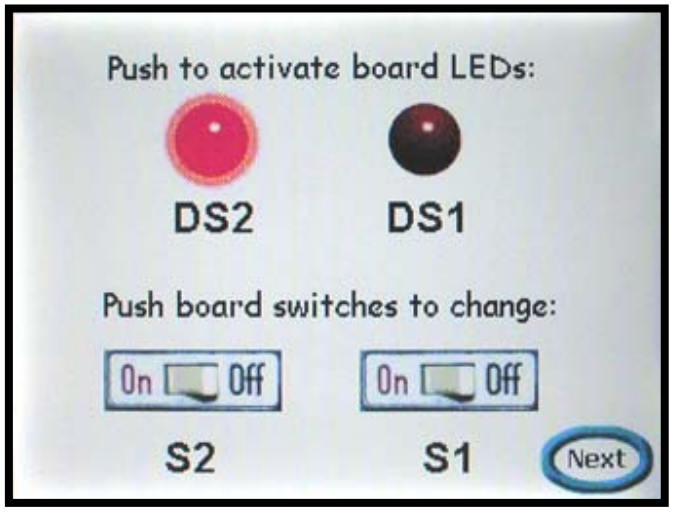


```
770      while(1)
771      {
772         // while no button press, check board switches
773         while( !anySLCDrx(&btn) )
774         {
775            // this code is modeled on the TOGGLESWITCH.C sample
776               costate
777            {
778               // check for high == "off" state
779               if (BitRdPortI(PFDR, S1))
780               {
781                  s1 = 0;                                    // set to off
782                  abort;                     // get out
783               }
784               waitfor(DelayMs(10));         //switch press detected if got to here
785               if (BitRdPortI(PFDR, S1))     //debounce - see if it still low
786                  abort;
787               // have a valid S1 press, so set switch state
788               s1 = 1;
789               abort;
790            }
791
792               costate
793            {
794               // check for high == "off" state
795               if (BitRdPortI(PBDR, S2))
796               {
797                  s2 = 0;                                    // set to off
798                  abort;                     // get out
799               }
800               waitfor(DelayMs(10));         //switch press detected if got to here
801               if (BitRdPortI(PBDR, S2))     //debounce - see if it still low
802                  abort;
803               // have a valid S1 press, so set switch state
804               s2 = 1;
805               abort;
806            }
807
808               costate
809            {  // set image
```

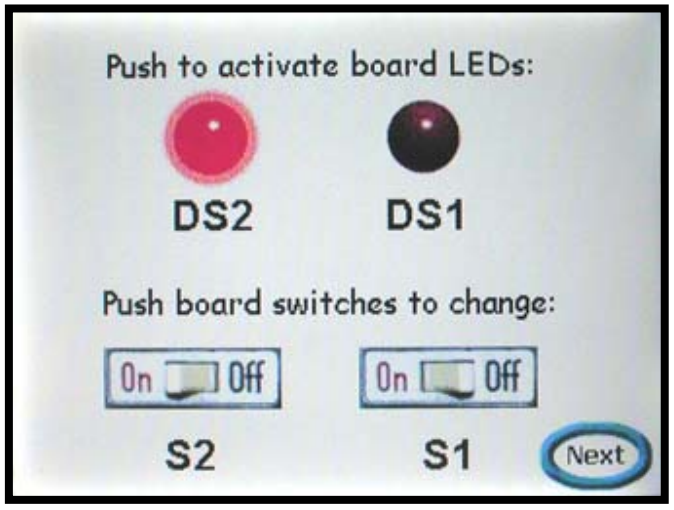
```
810             if(s2 != s2Image)
811             {
812                s2Image = s2;
813                if( s2Image == 0 )
814                   slcd("xi 28 46 166");
815                else
816                   slcd("xi 29 46 166");
817             }
818
819             if(s1 != s1Image)
820             {
821                s1Image = s1;
822                if( s1Image == 0 )
823                   slcd("xi 28 171 166");
824                else
825                   slcd("xi 29 171 166");
826             }
827          }
828       } // end while no button press from SLCD
829
830       switch( btn.index )
831       {
832          case 1:
833             if(btn.state == 1)
834                pbLedOn(DS1);
835                else
836                pbLedOff(DS1);
837             break;
838
839          case 2:
840             if(btn.state == 1)
841                pbLedOn(DS2);
842                else
843                pbLedOff(DS2);
844             break;
845
846          case 3:
847             return;
848
849             default:
```

```
850                 printf("Internal error - invalid button #%d\n",btn.index);
851                 break;
852             }
853         }
854     }
855
```

Push on screen LED button DS2 to show this; proto board LED also lights up.



S2 icon changes when proto board switch S1 is pushed.



```
#ifdef DEBUG
   static char lastCommand[120];
#endif

void slcd( char *s )
{
#ifdef DEBUG
   strcpy(lastCommand,s);
#endif
   if(*s) serDputs(s);
   serDputc('\r');
   cmdCount++;
   // want no more than MAX_CMDS commands outstanding
   while( cmdCount >= MAX_CMDS )
   {
      // process SLCD prompts; ignore button presses
      anySLCDrx(&btn);
   }
}
```

The slcd() routine sends the command strings to the SLCD. It also enforces the maximum commands outstanding and helps debugging by storing the last command so if a syntax error occurs we can print out the offending command. Define the DEBUG flag to enable this.

If there are too many commands outstanding, process some return prompts until the command count comes down. This enforces a throttling so the SLCD does not get overrun. A fancier way would be to implement a circular buffer and true XON/XOFF, but this works fine. The main goal is to at least allow an overlap of command execution and command communications, and that would happen as long as there are at least 2 commands outstanding (the next transmitting while the last is executing).

```
#define RXBUF_SIZE 129
static char rxBuf[RXBUF_SIZE];
static char *rxHead;
static char *rxTail;
static uchar rxCount;
static uchar rxGotline;

// putChar - assumes there is room in the buffer...
void sPutChar(unsigned char c)
{
   *rxHead++ = (char) c;
    rxCount++;
   if( rxHead == rxBuf+RXBUF_SIZE )
   {
      rxHead = rxBuf;
   }
}
```

This is the start of the routines that implement the circular receive buffer. The buffer is 129 characters long, which is somewhat arbitrary given that there is no return string from the SLCD that is this long.

These routines do NOT check for overrun - it is assumed that by limiting the number of commands outstanding, the maximum number of return prompts outstanding is limited as well. There should be a check in sPutChar that rxCount is less than RXBUF_SIZE....

```
896
897   // getChar - assumes there is something to get...
898   uchar sGetChar(void)
899   {
900      unsigned char c;
901      c = *rxTail++;
902      rxCount --;
903      if( rxTail == rxBuf+RXBUF_SIZE )
904       {
905         rxTail = rxBuf;
906       }
907      return( c );
908   }
909
910
911   // getResponse - assumes string with a <return> is in the buffer
912   void getResponse(char *s, int len)
913   {
914      uchar c;
915
916      c = 0;
917      while( rxCount && len && c != '\r' )
918      {
919         c = sGetChar();
920         *s++ = c;
921         len--;
922      }
923      if( c != '\r' )
924      {
925         printf("internal error #2");
926      }
927      // null terminate the string
928      if(len)
929      {
930         *s = 0;
931      }
932      else
933      {
934         printf("internal error #3");
935      }
```

The getResponse() function gets the next SLCD response in the circular buffer, terminated by a <return>. It should only be called if the counter rxGotline is non-zero indicating that there is a complete response in the buffer.

```
936   }
937
938   // peekChar - assumes there is something to get...
939   uchar peekChar(void)
940   {
941      return( *rxTail );
942   }
943
944   void sRxInit(void)
945   {
946         int i;
947         // for debug
948         for(i=0;i<RXBUF_SIZE;i++) rxBuf[i]=0;
949
950         rxHead = rxTail = rxBuf;
951         rxCount = 0;
952         rxGotline = 0;
953         cmdCount = 0;
954   }
955
956   //=====================================================
957   // anySLCDrx()
958   // This checks the SLCD rx for a response.
959   // return is the number of the button press received
960   // return -1 menas no response
961   //=====================================================
962   #define BUF_LEN 80
963   uchar anySLCDrx(struct button *btn)
964   {
965      int i;
966      uchar c;
967      uchar buf[BUF_LEN];
968
969      //
970      // get any rx'd characters
971      //
972      while( -1 != (i= serDgetc()))
973      {
974         sPutChar((char)i);
975         if( i == '\r' )
```

This is the main routine called by screen processing routines. It collects incoming characters and counts <returns>, then it processes accumulated return strings from the SLCD.

```
976         {
977            rxGotline++;
978         }
979      }
980
981      //
982      // process responses (strings followed by a return)
983      //
984      while( rxGotline )
985      {
986         // get the response
987         getResponse( buf, BUF_LEN );
988         rxGotline--;
989
990         // see what to do with it
991         switch( buf[0] )
992         {
993            // command prompt
994            case '>':
995               if( buf[1] == '\r' )   // end of string
996               {
997                  if( cmdCount )
998                  {
999                     cmdCount--;
1000                 }
1001                 else
1002                 {
1003                    printf("command count mismatch - extra prompt\n");
1004                 }
1005              }
1006              else
1007              {
1008                 printf("> prompt without following <return>\n");
1009              }
1010              break;
1011
1012           // momentary button or hotspot press
1013           case 'x':
1014              i = atoi(&buf[1]);
1015              if( i > 255 )
```

Here, all possible SLCD return strings are processed and dealt with. There are standard prompts, syntax error prompts, as well as buffer overflows, and framing errors to catch and deal with. After that there are the button responses. If a button response is found, the calling routine's button structure is updated. Note that buttons are not queued - it is assumed that they can be handled quickly enough.

```
1016              {
1017                 printf("Bad rx string \"%s\"\n",buf);
1018                 return(0);
1019              }
1020              btn->index = i;
1021              return(1);
1022              break;
1023
1024           // latching button press
1025           case 's':
1026              i = atoi(&buf[1]);
1027              if( i > 255 )
1028              {
1029                 printf("Bad rx string \"%s\"",buf);
1030                 return(0);
1031              }
1032              btn->index = i/10;
1033              btn->state = i%10;
1034              return(1);
1035              break;
1036
1037           case '!':
1038              printf("Command sent to SLCD generated a syntax error\n");
1039   #ifdef DEBUG
1040              printf("Last Command was\n%s\n",lastCommand);
1041   #endif
1042              if( cmdCount )    cmdCount--;
1043              return(0);
1044              break;
1045
1046           case '^':
1047              printf("SLCD input buffer overflow\n");
1048              // on overflow, all commands are flushed
1049              if( cmdCount )    cmdCount = 0;
1050              return(0);
1051              break;
1052
1053           case '?':
1054              printf("SLCD input framing error\n");
1055              return(0);
```

```
1056                break;
1057
1058             default:
1059                printf("Unrecognized SLCD response: 1st char 0x%x, string = \"%s\"\n",buf[0],buf);
1060    #ifdef DEBUG
1061                printf("Last Command was\n%s\n",lastCommand);
1062    #endif
1063                return(0);
1064                break;
1065
1066          }
1067       }
1068       // this hits if there are only command prompts...
1069       return(0);
1070    }
1071
1072
1073    ////////////////////////////////////////////////////////
1074    // DS1 led on protoboard is controlled by port D bit 6
1075    // DS2 led on protoboard is controlled by port D bit 7
1076    // turns on when port bit is set to 0
1077    ////////////////////////////////////////////////////////
1078    void pbLedOn(int led)
1079    {
1080       BitWrPortI(PFDR, &PFDRShadow, 0, led);
1081    }
1082    void pbLedOff(int led)
1083    {
1084       BitWrPortI(PFDR, &PFDRShadow, 1, led);
1085    }
1086
1087    // delay function - hack when it rolls over
1088    void pauseMs(int delay)
1089    {
1090       unsigned long t, t2;
1091       char wait;
1092
1093       wait = 1;
1094       t = MS_TIMER;
1095       while( wait )
```

```
1096      {
1097         if( MS_TIMER >= t )
1098         {
1099            if( MS_TIMER > t+delay )
1100               wait = 0;
1101         }
1102         else
1103         {
1104            if( MS_TIMER > delay )
1105               wait = 0;
1106         }
1107      }
1108   }
1109
1110   void bmp(uint bitmap, uint x, uint y)
1111   {
1112      char buf[20];
1113      sprintf(buf,"xi %d %d %d",bitmap,x,y);
1114      slcd(buf);
1115   }
```

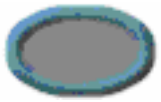
Appendix A - Bitmaps used in the program

The RCM3720_SLCD_demo.c program requires a specific set of graphical bitmaps to be loaded into the SLCD unit before it can run. This appendix shows what these bitmaps look like.

21_MainScrn.bmp

22_round_button.bmp

23_round_button_dn.bmp

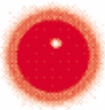


24_check_box.bmp

25_check_box_click.bmp

26_LEDOFF.bmp

27_LEDON.bmp



28_board_sw_off.BMP

29_board_sw_on.BMP

30_big_button.bmp

31_big_button_dn.bmp

32_Bezel.BMP

33_volts_on.BMP 

34_volts_off.BMP 

35_mA_on.BMP 

36_mA_off.BMP 

37_button_on.BMP 

38_button_off.BMP 

39_button_up.BMP 

40_button_dn.BMP