SLCD Application Note AN-110

Sample Programs for an Arduino Mega

Reach Technology, Inc.

07/27/2012

Reach Technology Inc.

www.reachtech.com

Sales: 510-770-1417 x112
sales@reachtech.com

Technical Support: 503-675-6464
techsupport@reachtech.com

1 Overview

This application note describes a sample application 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 basic function of this application is to provide an analog pulse with a user-selectable pulse profile (pulse duration and amplitude). The user selects from one of several buttons on the main screen to choose a pre-defined pulse profile, or the user may use two sliders on an alternate screen to fine-tune the pulse profile. A trigger button on either screen (or a 'switch-to-ground' input to the Arduino) tells the Arduino to start the pulse. An LED on the Arduino is used to show the duration of the pulse. An RC filter on the PWM output from the Arduino converts the signal into an analog pulse.

The sample application contains two approaches to working with the SLCD controller. The first approach, called the Low-Level approach, illustrates how to send individual commands to the SLCD to draw buttons, sliders and text, and how to field responses from the SLCD. This approach is simple to implement but makes the application dependent on the format of the SLCD.

The second approach, called the Macro Approach, takes advantage of the SLCD controller's macro capability to separate the format of the SLCD from the application, which allows the application to work with several different SLCD formats without change. The macros loaded into each SLCD controller handle the format-specific differences and only send messages to the Arduino when it needs to take action.

The user interface for both approaches is setup for the 4.3" LCD Display Module. Other SLCD modules will work as well, but the button and slider placement will only occupy the space used by the 4.3" display module.

The code for this application note is written for an Arduino Mega, using the Arduino IDE version 1.0. Other Arduino boards may be able to be used, but might require changes in the handling of the timer2 output. Likewise, the serial port used to talk to the SLCD controller might need to change if a different Arduino board is used.

This application note assumes that the reader has a copy of the SLCD Software Reference Manual and the reference manual for the SLCD controller to be used.

2 SLCD Communications Interface

The serial interface to the SLCD is full duplex. Commands can be sent independently of return acknowledgements, and once a button or slider is defined on the screen, button press and slider change responses can be received from the SLCD at any time.

This example implements a simple receive buffer which collects characters from the Arduino's internal serial buffer. These characters are parsed for SLCD command prompts, button press messages and slider change messages. The characters are read using the Arduino Serial1 function, which is defined as `#define DISPLAY_SIO Serial1` to allow for changing serial ports easily. Likewise, a debug serial port is defined as `#define DEBUG_SIO Serial` and is used for debug messages that can be monitored with the Arduino serial monitor (in the IDE).

Both approaches implement the function serialEvent() for the Arduino environment to use to tell the application that serial data has been received. Note that this function is called by the Arduino environment after each iteration of the loop() function, so if a section of code in loop() is waiting for serial data, that section must call serialEvent() manually. serialEvent() is defined as `#define DISPLAY_SERIALEVENT serialEvent1` to reflect the use of serial port 1 on the Arduino Mega.

There is also a function wait_for_display() that is used to monitor the number of outstanding commands that have been sent to the SLCD controller. It will stall if the number of outstanding commands is more than 2 to prevent from overrunning the SLCD's input buffer (in lieu of implementing XON/XOFF handling in the application). This function calls serialEvent() on a regular basis to receive replies from the SLCD controller.

The SLCD supports software flow control (XON/XOFF), and instead of limiting the number of commands outstanding, the more general approach would be to implement true software flow control in the application. This was not done in this application note in order to reduce complexity.

3 Program Structure

There are two different versions of the application. The first version implements the Low-Level approach, and is contained in the file `ArduinoExampleLL\ArduinoExampleLL.ino`. This version depends on bitmap files that are located in the folder `BMP and Macro`.

The second version implements the Marco Approach and is contained in the folder `ArduinoExampleMacros\ArduinoExampleMacros.ino`. This version depends on bitmap files and a macro definition file, all located in the folder `BMP and Macro`.

3.1 Low-Level Approach

The application in the Low-Level approach does all of the work. It uses individual button define commands to draw each button on each screen, individual slider define messages to draw the sliders on the alternate screen and text display messages to draw the slider labels and the pulse profile information. It also receives and handles button press notifications for each button that is pressed and slider change notifications each time a slider is moved.

The setup() function is called by the Arduino environment to setup the hardware for the application and to do any other one-time initialization that the application needs.

The loop() function is the main work loop of the application. It is called repeatedly by the Arduino environment. Each iteration of loop() starts by making sure the proper screen is displayed on the SLCD, and then it checks for any incoming messages from the SLCD. If any button press notifications or slider change notifications have been received, the appropriate handler is called to decode the message and translate the values received into new pulse profile information. If the TRIG button message is received the trigger flag is set. The footswitch input is then checked for a change to a low state, and if active after a debounce time, a trigger flag is set. Finally, if the trigger flag is set, a pulse is started using the current pulse profile and the pulse profile information is printed on the SLCD screen.

Various 'draw' functions are defined to draw the main screen, the alternate screen, and buttons and sliders on the two screens. Other functions are defined to help print pulse profile information on the SLCD screen, to decode button press notifications and slider change notifications, to issue the pulse itself, and the wait_for_display() and DISPLAY_SERIAL_EVENT() functions mentioned previously.

3.2 Macro Approach

The application in the Macro approach does only a portion of the work. It tells the SLCD to run the `power_on` macro to initialize itself, and then waits for messages from the SLCD macros with new pulse profile information, screen change information or trigger requests. Macros on the SLCD are used to translate button presses into pulse profile information, handle screen ID changes and slider changes, and to generate trigger requests. Another macro is used by the application to print pulse profile information. The application doesn't know how many buttons are defined or how they are interpreted by the macros.

The setup() function is called by the Arduino environment to setup the hardware for the application and to do any other one-time initialization that the application needs. This function also tells the SLCD to run the `power_on` macro to initialize itself.

The loop() function is the main work loop of the application. It is called repeatedly by the Arduino environment. Each iteration of loop() starts by checking for any incoming messages from the SLCD.

The messages from the SLCD macros all start with a ‘@’ character, and then have a message identifier character as follows:

1. @p – pulse profile message

2. @t – trigger request message

3. @s – screen ID message

If a pulse profile message has been received, the handler is called to decode the message (based on active screen) and translate the values received into new pulse profile information. If a screen ID message has been received, the application makes note of the new active screen for use in decoding pulse profile messages. If a trigger request message is received, a trigger flag is set to trigger a new pulse. The footswitch input is then checked for a change to a low state, and if active after a debounce time, the trigger flag is set. Finally, if the trigger flag is set, a pulse is started using the current pulse profile and the pulse profile information is printed on the SLCD screen using the `display_pulse_profile` macro.

3.3 Common Functions

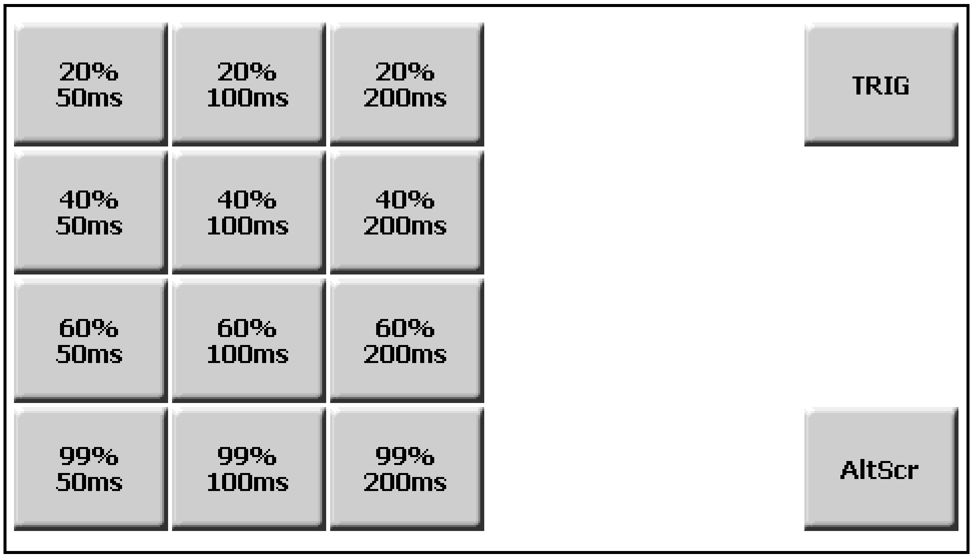
The two application approaches have the same functions for:

- do_pulse
- handle_footswitch
- wait_for_display
- DISPLAY_SERIALEVENT

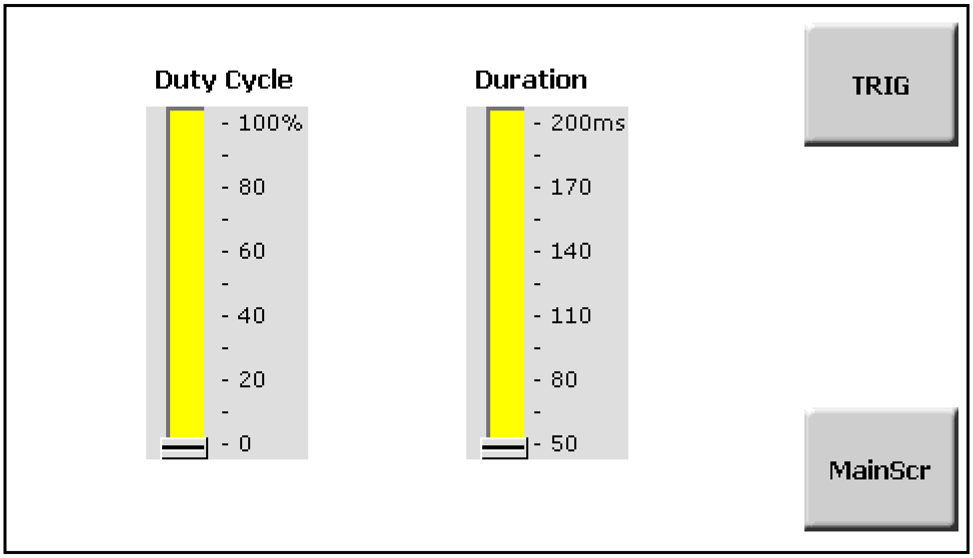
Both approaches also have a function or macro which is used to send pulse profile information to the display, but in the Low-Level approach, this is a function called `print_pulse_profile()` and it uses a text display command for each value, whereas in the Macro approach uses the macro `display_pulse_profile` to print both values at once.

4 Application Screens

The main screen contains several buttons to select pre-set pulse profiles, as well as a TRIG button and an AltScr button



The alternate screen contains two sliders manually set the pulse profile (duty cycle and duration), as well as a TRIG button and a MainScr button:



5 Bitmaps Used

Some of the bitmap files used in these two application approaches are taken from the standard bitmaps provided in the demo application that comes with the SLCD Development Kits. They are extracted to make this application note standalone. The two slider background bitmaps are derived from the demo application slider background but have been changed to suit this application.

01_big_button.bmp
02_big_button_dn.bmp
03_DutyCycleSliderBackground.bmp
04_DurationSliderBackground.bmp
05_slider2Knob.unc.bmp

6 Bitmap and Macro Installation

The same set of bitmap files are used for both application approaches. They are located in the folder `BMP and Macro`. This folder also contains the macro definitions used in the Macro approach. Both application approaches use the BMPload program to load the bitmaps and macros into the SLCD. The bitmaps and macros only need to be downloaded once, or after downloading other bitmaps and macros.

Low-Level approach:

- start BMPload
- click on “Load BMP List”
- browse to the `BMP and Macro` folder,
- select the file `ArduinoBMP.LST` and click on “Open”
- make sure the following “Extra Settings” options is checked:
    - High Color
- make sure the following “Extra Settings” options are not checked:
    - Set Power On Macro
    - Set Splash Screen
    - Set Control Port
- click on the “Store into SLCD” button to download the bitmaps into the SLCD

Macro approach:

- start BMPload
- click on “Load BMP List”
- browse to the `BMP and Macro` folder
- select the file `ArduinoBMP.LST` and click on “Open”
- click on the “Add Macro File” button
- select the file `ArduinoMacros.txt` and click on “Open”
- make sure the following “Extra Settings” options is checked:
    - High Color
- make sure the following “Extra Settings” options are not checked:
    - Set Power On Macro
    - Set Splash Screen
    - Set Control Port
- click on the “Store into SLCD” button to download the bitmaps and macros into the SLCD

7 Source Code for Low-Level Approach

```
/*
 * ArduinoExampleLL:
 *
 * Arduino sketch to present a user interface on a Reach Technology
 * SLCD43 Development Kit display, receive button press and slider
 * position notifications, and then drive an analog pulse with a
 * settable pulse profile (PWM duty cycle and duration).
 *
 * This sketch is an example of a low-level control application
 * that manages the display completely, drawing objects on the
 * screen as required and handling all button/slider notifications
 * directly.
 *
 * The main display presents several buttons to select pre-set
 * pulse profiles.
 *
 * An AltScr button selects an alternate display screen that
 * presents two sliders to manually select the pulse profile
 * parameters.
 *
 * A TRIG button on either screen starts the pulse, as will
 * activating a short-to-ground switch connected to digital
 * pin 2.
 *
 * Timer2 on the Arduio is used to drive a PWM signal to create
 * the output pulse.  The PWM output from timer2 on OC2B (pin 9
 * on Mega2560) is filtered with an RC network to convert the
 * output into an analog signal.  The duty cycle is varied to
 * adjust the pulse amplitude.  The pulse duty cycle and duration
 * are based on information received from the display.
 *
 * On both screens, the low-level pulse profile settings are
 * displayed each time the pulse is triggered (either by the
 * TRIG button or by the footswitch).
 *
 * An LED with 330 ohm current limiting resistor connected to
 * pin 12 will turn on for the selected pulse time each time a
 * pulse is triggered.
 */

// compiler-provided definitions
// __AVR_ATmega328P__ - Uno
// __AVR_ATmega1280__ - Mega
// __AVR_ATmega2560__ - Mega 2560


// button image size literals, based on bitmap image sizes
#define  BUTTON_30_WIDTH    77    // width of 30_big_button BMP
#define  BUTTON_30_HEIGHT   62    // height of 30_big_button BMP
#define  SLIDER_44_WIDTH    81    // width of 44_slider2Back BMP
#define  SLIDER_44_HEIGHT   176   // height of 44_slider2Back BMP
```

```
// button placement literals based on button dimensions
#define  BUTTON_ROW_OFFSET  8
#define  BUTTON_COL_OFFSET  4
#define  BUTTON_SPACING     2
#define  BUTTON_ROW_1       (BUTTON_ROW_OFFSET)
#define  BUTTON_ROW_2       (BUTTON_ROW_1+BUTTON_30_HEIGHT+BUTTON_SPACING)
#define  BUTTON_ROW_3       (BUTTON_ROW_2+BUTTON_30_HEIGHT+BUTTON_SPACING)
#define  BUTTON_ROW_4       (BUTTON_ROW_3+BUTTON_30_HEIGHT+BUTTON_SPACING)
#define  BUTTON_COL_1       (BUTTON_COL_OFFSET)
#define  BUTTON_COL_2       (BUTTON_COL_1+BUTTON_30_WIDTH+BUTTON_SPACING)
#define  BUTTON_COL_3       (BUTTON_COL_2+BUTTON_30_WIDTH+BUTTON_SPACING)
#define  BUTTON_COL_4       (BUTTON_COL_3+BUTTON_30_WIDTH+BUTTON_SPACING)
#define  BUTTON_COL_5       (BUTTON_COL_4+BUTTON_30_WIDTH+BUTTON_SPACING)
#define  BUTTON_COL_6       (BUTTON_COL_5+BUTTON_30_WIDTH+BUTTON_SPACING)

// index value assignments for 'special' buttons and sliders
#define  MAIN_SCREEN_BUTTON 20
#define  ALT_SCREEN_BUTTON  21
#define  TRIGGER_BUTTON     50
#define  DUTYCYCLE_SLIDER   128
#define  DURATION_SLIDER    129

// slider image size literals
#define  SLIDER_ROW                  50
#define  SLIDER_LABEL_ROW            (SLIDER_ROW-20)
#define  DUTYCYCLE_SLIDER_COL        70
#define  DUTYCYCLE_SLIDER_LABEL_COL  (DUTYCYCLE_SLIDER_COL+5)
#define  DURATION_SLIDER_COL         230
#define  DURATION_SLIDER_LABEL_COL   (DURATION_SLIDER_COL+5)

// notifications from display, see SLCDx Software Reference
#define  BUTTON_PRESS      'x'
#define  SLIDER_ACTION     'l'

// maximum number of commands sent without getting a reply
#define  MAX_CMD_OUTSTANDING  2

// duty cycle definitions assuming 'fast PWM' config
#define  DUTY_CYCLE_MAX     255
#define  DUTY_CYCLE_20PCNT  ((DUTY_CYCLE_MAX*20)/100)
#define  DUTY_CYCLE_40PCNT  ((DUTY_CYCLE_MAX*40)/100)
#define  DUTY_CYCLE_60PCNT  ((DUTY_CYCLE_MAX*60)/100)
#define  DUTY_CYCLE_99PCNT  ((DUTY_CYCLE_MAX*99)/100)
#define  MIN_DUTY_CYCLE     DUTY_CYCLE_20PCNT

// duration literals
#define  MIN_DURATION      50      // ms
#define  MAX_DURATION      200     // ms

// length of serial input buffer
#define  BUFF_SIZE        100

// pin definitions
#define  FOOTSWITCH_IN    2
#define  PWM_OUT          9  // timer2 OC2B output
#define  DEBUG_LED        12
```

```
// serial IO port mapping literals
#define  DISPLAY_SIO          Serial1
#define  DISPLAY_SERIALEVENT  serialEvent1
#define  DEBUG_SIO            Serial    // to use IDE's Serial Monitor

// debug support
#undef  SHOW_DEBUG
#ifdef  SHOW_DEBUG
// these are undefined if SHOW_DEBUG is undefined,
// otherwise may be defined or not as needed
#define  BUTTON_PRESS_DEBUG
#define  SLIDER_DEBUG
#define  SLIDER_VAL_DEBUG
#undef   DEBUG_CMD_OVERLAP
#endif

// misc
#define  FOOTSWITCH_DEBOUNCE  50  // time in milliseconds

// variables and flags
boolean  main_screen;
boolean main_screen_active;
boolean alt_screen_active;
byte footswitch_state;
byte last_footswitch_state;
boolean trigger_set;
boolean trigger_holdoff;
byte pulse_duty_cycle;
int pulse_duration;
char buffer[BUFF_SIZE];
int  buffer_index;
char  replies_req;
boolean message_ready;
boolean serial_error;

// function prototypes
void draw_screen(void);
void draw_main_screen(void);
void draw_alt_screen(void);
void clear_screen(void);
void draw_button(int index, int button_x, int button_y, char *text);
void draw_slider_label(int label_x, int label_y, char *text);
void draw_slider(int index, int slider_x, int slider_y);
int decode_ASCII(char buffer[], int *index, char delim);
void print_pulse_profile(void);
void handle_button(void);
void handle_slider(void);
void do_pulse(void);
void handle_footswitch(void);
void wait_for_display(void);
void DISPLAY_SERIALEVENT(void);


/*
 * setup:
 *
 * the required Arduino sketch setup() function.
 */
 void setup()
{

  // timer2 output pin
  pinMode(PWM_OUT, OUTPUT);

  // footswitch input, with internal 20k pull-up
  pinMode(FOOTSWITCH_IN, INPUT);
  digitalWrite(FOOTSWITCH_IN, HIGH);

  // debug LED
  pinMode(DEBUG_LED, OUTPUT);

  // init the serial port and send a few <return>s
  // to get the display's attention
  DISPLAY_SIO.begin(115200);
  DISPLAY_SIO.print('\r');
  DISPLAY_SIO.print('\r');
  DISPLAY_SIO.print('\r');

#ifdef  SHOW_DEBUG
  DEBUG_SIO.begin(9600);
  DEBUG_SIO.println("Showing debug");
#endif

  // set our initial state to the main screen
  main_screen = true;
  main_screen_active = false;
  alt_screen_active = false;
  footswitch_state = digitalRead(FOOTSWITCH_IN);
  last_footswitch_state = footswitch_state;
  trigger_set = false;
  trigger_holdoff = false;
  pulse_duty_cycle = MIN_DUTY_CYCLE;
  pulse_duration = MIN_DURATION;
  buffer_index = 0;
  replies_req = 0;
  message_ready = false;
  serial_error = false;

  // set the font
  DISPLAY_SIO.println("f 13B");
  replies_req++;
  wait_for_display();

  // soften the button beep
  DISPLAY_SIO.println("bvs 25");
  replies_req++;
  wait_for_display();

}


/*
 * loop:
 *
 * the required Arduino loop() function.
 */
void loop()
{
  int  i, num_chars;

  draw_screen();

  // check for button and slider notifications
  if (message_ready)
  {
//    DEBUG_SIO.println("Handling a message");
    message_ready = false;
    if (buffer[0] == BUTTON_PRESS)
    {
      handle_button();
    }  /* button press */
    else if (buffer[0] == SLIDER_ACTION)
    {
      handle_slider();
    }  /* slider value */
    else
    {
      // unknown message - ignore
    }
  }

  // check for footswitch trigger
  handle_footswitch();

  // trigger pulse if flag is set
  if (trigger_set)
  {
    print_pulse_profile();
    trigger_set = false;
    do_pulse();
  }

}

/*
 * draw_screen:
 *
 * called to draw the proper screen based
 * on the main_screen state
 */
void draw_screen(void)
{

  if (main_screen)
  {
    // but only draw the screen if it is not yet active
    if (!main_screen_active)
    {
      draw_main_screen();
      // track the active screen
      main_screen_active = true;
      alt_screen_active = false;
    }
  }
  else
  {
    if (!alt_screen_active)
    {
      draw_alt_screen();
      alt_screen_active = true;
      main_screen_active = false;
    }
  }

}


/*
 * draw_main_screen:
 *
 * draws the buttons for the main screen.
 */
void draw_main_screen(void)
{

  clear_screen();

  // draw buttons with centered text
  // NOTE: button number assignments are also 'known'
  // in handle_button() switch statement
  draw_button(1, BUTTON_COL_1, BUTTON_ROW_1, "20%\n50ms");
  draw_button(2, BUTTON_COL_2, BUTTON_ROW_1, "20%\n100ms");
  draw_button(3, BUTTON_COL_3, BUTTON_ROW_1, "20%\n200ms");
  draw_button(4, BUTTON_COL_1, BUTTON_ROW_2, "40%\n50ms");
  draw_button(5, BUTTON_COL_2, BUTTON_ROW_2, "40%\n100ms");
  draw_button(6, BUTTON_COL_3, BUTTON_ROW_2, "40%\n200ms");
  draw_button(7, BUTTON_COL_1, BUTTON_ROW_3, "60%\n50ms");
  draw_button(8, BUTTON_COL_2, BUTTON_ROW_3, "60%\n100ms");
  draw_button(9, BUTTON_COL_3, BUTTON_ROW_3, "60%\n200ms");
  draw_button(10, BUTTON_COL_1, BUTTON_ROW_4, "99%\n50ms");
  draw_button(11, BUTTON_COL_2, BUTTON_ROW_4, "99%\n100ms");
  draw_button(12, BUTTON_COL_3, BUTTON_ROW_4, "99%\n200ms");

  draw_button(TRIGGER_BUTTON, BUTTON_COL_6, BUTTON_ROW_1, "TRIG");
  draw_button(ALT_SCREEN_BUTTON, BUTTON_COL_6, BUTTON_ROW_4, "AltScr");

}


/*
 * draw_alt_screen:
 *
 * draws the sliders and buttons for the alternate screen.
 */
void draw_alt_screen(void)
{

  clear_screen();
  draw_slider_label(DUTYCYCLE_SLIDER_LABEL_COL,
                    SLIDER_LABEL_ROW,
                    "Duty Cycle");
  draw_dutycycle_slider(DUTYCYCLE_SLIDER,
                    DUTYCYCLE_SLIDER_COL, SLIDER_ROW);

  draw_slider_label(DURATION_SLIDER_LABEL_COL,
                    SLIDER_LABEL_ROW,
                    "Duration");
  draw_duration_slider(DURATION_SLIDER,
                    DURATION_SLIDER_COL, SLIDER_ROW);

  draw_button(TRIGGER_BUTTON, BUTTON_COL_6, BUTTON_ROW_1, "TRIG");
  draw_button(MAIN_SCREEN_BUTTON, BUTTON_COL_6, BUTTON_ROW_4, "MainScr");

}


/*
 * clear_screen:
 *
 * called to clear the screen
 */
void clear_screen(void)
{

  wait_for_display();
  DISPLAY_SIO.println("z");
  replies_req++;

}


/*
 * draw_button:
 *
 * draws a 'big_button' at button_x,button_y with the label
 * 'text' centered in the button.  button is configured as
 * a Type 1 button (notify on press only).
 */
void draw_button(int index, int button_x, int button_y, char *text)
{

  wait_for_display();
  DISPLAY_SIO.print("bdc ");        // button with centered text
  DISPLAY_SIO.print(index, DEC);
  DISPLAY_SIO.print(" ");
  DISPLAY_SIO.print(button_x, DEC);
  DISPLAY_SIO.print(" ");
  DISPLAY_SIO.print(button_y, DEC);
  DISPLAY_SIO.print(" 1 \"");      // type 1 - notify on press only
  DISPLAY_SIO.print(text);
  DISPLAY_SIO.println("\" 1 2");   // 'big_button', formerly bitmaps 30 and
31
  replies_req++;
```

```
}


/*
 * draw_slider_label:
 *
 * called to draw a label for a slider
 */
void draw_slider_label(int label_x, int label_y, char *text)
{

  wait_for_display();
  DISPLAY_SIO.print("t \"");
  DISPLAY_SIO.print(text);
  DISPLAY_SIO.print("\" ");
  DISPLAY_SIO.print(label_x, DEC);
  DISPLAY_SIO.print(" ");
  DISPLAY_SIO.println(label_y, DEC);
  replies_req++;

}


/*
 * draw_dutycycle_slider:
 *
 * called to draw a DutyCycleSlider at slider_x
 * and slider_y.  knob is offset 5 pixels from edge.
 * lo/hi values are set to 1 and 99 to avoid 0 (and
 * needing to check for 0 in calc's).  The 'lo' vaue
 * is at the bottom of the slider.
 */
void draw_dutycycle_slider(int index, int slider_x, int slider_y)
{

  wait_for_display();
  DISPLAY_SIO.print("sl ");
  DISPLAY_SIO.print(index, DEC);
  DISPLAY_SIO.print(" 3 ");
  DISPLAY_SIO.print(slider_x, DEC);
  DISPLAY_SIO.print(" ");
  DISPLAY_SIO.print(slider_y, DEC);
  DISPLAY_SIO.println(" 5 7 0 1 1 99 0");
  replies_req++;

}


/*
 * draw_duration_slider:
 *
 * called to draw a DurationSlider at slider_x
 * and slider_y.  knob is offset 5 pixels from edge.
 * lo/hi values are set to 50 and 200.  The 'lo'
 * vaue is at the bottom of the slider.
 */
void draw_duration_slider(int index, int slider_x, int slider_y)
```

```
{

  wait_for_display();
  DISPLAY_SIO.print("sl ");
  DISPLAY_SIO.print(index, DEC);
  DISPLAY_SIO.print(" 4 ");
  DISPLAY_SIO.print(slider_x, DEC);
  DISPLAY_SIO.print(" ");
  DISPLAY_SIO.print(slider_y, DEC);
  DISPLAY_SIO.println(" 5 7 0 1 1 200 50");
  replies_req++;

}


/*
 * decode_ASCII:
 *
 * called to decod the ASCII numeric value at buffer[index].
 * returns the decoded value, and leaves index set to the
 * delimiter or the zero terminator.
 *
 * assumes that there are no more than three ASCII characters.
 */
int decode_ASCII(char buffer[], int *index, char delim)
{
  int  val, l_index;

  l_index = *index;
  val = buffer[l_index++] - '0';
  if ((buffer[l_index] != delim) && (buffer[l_index] != 0))
  {
    val = (val * 10) + (buffer[l_index++] - '0');
  }
  if ((buffer[l_index] != delim) && (buffer[l_index] != 0))
  {
    val = (val * 10) + (buffer[l_index++] - '0');
  }

#ifdef  PULSE_PROFILE_DEBUG
  DEBUG_SIO.print("decode_ASCII: ");
  DEBUG_SIO.print(val, DEC);
  DEBUG_SIO.print(" ending index: ");
  DEBUG_SIO.println(l_index, DEC);
#endif

  *index = l_index;
  return(val);

}


/*
 * print_pulse_profile:
 *
 * called to print the current low-level pulse profile
 */
```

```
void print_pulse_profile(void)
{

  wait_for_display();
  DISPLAY_SIO.print("t \"dc: ");
  DISPLAY_SIO.print(pulse_duty_cycle, DEC);
  DISPLAY_SIO.print("     \" ");  // extra spaces clear old text
  DISPLAY_SIO.print(BUTTON_COL_6, DEC);
  DISPLAY_SIO.print(" ");
  DISPLAY_SIO.println(BUTTON_ROW_3-20, DEC);
  replies_req++;

  wait_for_display();
  DISPLAY_SIO.print("t \"time: ");
  DISPLAY_SIO.print(pulse_duration, DEC);
  DISPLAY_SIO.print("     \" ");  // extra spaces clear old text
  DISPLAY_SIO.print(BUTTON_COL_6, DEC);
  DISPLAY_SIO.print(" ");
  DISPLAY_SIO.println(BUTTON_ROW_3, DEC);
  replies_req++;

}


/*
 * handle_button:
 *
 * handles button notifications for main screen and for
 * alt screen.  decodes button value into one of several
 * pre-set pulse profile settings, or into one of a few
 * special actions.
 */
void handle_button(void)
{
  int  index, button_num;

  // convert button number from ASCII
  index = 1;
  button_num = decode_ASCII(buffer, &index, ',');
#ifdef  BUTTON_PRESS_DEBUG
  DEBUG_SIO.print("button ");
  DEBUG_SIO.print(button_num, DEC);
  DEBUG_SIO.println(" pressed");
#endif

  switch (button_num)
  {
     case 1:
       pulse_duty_cycle = DUTY_CYCLE_20PCNT;
       pulse_duration = 50;   // ms
       break;
     case 2:
       pulse_duty_cycle = DUTY_CYCLE_20PCNT;
       pulse_duration = 100;  // ms
       break;
     case 3:
       pulse_duty_cycle = DUTY_CYCLE_20PCNT;
```

```
        pulse_duration = 200;  // ms
        break;
      case 4:
        pulse_duty_cycle = DUTY_CYCLE_40PCNT;
        pulse_duration = 50;   // ms
        break;
      case 5:
        pulse_duty_cycle = DUTY_CYCLE_40PCNT;
        pulse_duration = 100;  // ms
        break;
      case 6:
        pulse_duty_cycle = DUTY_CYCLE_40PCNT;
        pulse_duration = 200;  // ms
        break;
      case 7:
        pulse_duty_cycle = DUTY_CYCLE_60PCNT;
        pulse_duration = 50;   // ms
        break;
      case 8:
        pulse_duty_cycle = DUTY_CYCLE_60PCNT;
        pulse_duration = 100;  // ms
        break;
      case 9:
        pulse_duty_cycle = DUTY_CYCLE_60PCNT;
        pulse_duration = 200;  // ms
        break;
      case 10:
        pulse_duty_cycle = DUTY_CYCLE_99PCNT;
        pulse_duration = 50;   // ms
        break;
      case 11:
        pulse_duty_cycle = DUTY_CYCLE_99PCNT;
        pulse_duration = 100;  // ms
        break;
      case 12:
        pulse_duty_cycle = DUTY_CYCLE_99PCNT;
        pulse_duration = 200;  // ms
        break;

      case MAIN_SCREEN_BUTTON:
        main_screen = true;
        break;
      case ALT_SCREEN_BUTTON:
        main_screen = false;
        break;
      case TRIGGER_BUTTON:    // same on both screens
        trigger_set = true;
        break;

      default:
#ifdef  BUTTON_PRESS_DEBUG
        DEBUG_SIO.print("OOPS - unknown button ");
        DEBUG_SIO.print(button_num, DEC);
        DEBUG_SIO.println(" pressed");
#endif
        pulse_duty_cycle = MIN_DUTY_CYCLE;
        pulse_duration = MIN_DURATION;
```

```
       break;
  }

}


/*
 * handle_slider:
 *
 * handles slider notifications for the alt screen.  decodes
 * slider number and value, then maps slider value into the
 * appropriate range for the slider function.
 */
void handle_slider(void)
{
  int  index, slider_num, slider_val;

  // convert slider number from ASCII
  index = 1;
  slider_num = decode_ASCII(buffer, &index, ':');
  index++;    // skip delimiter
  slider_val = decode_ASCII(buffer, &index, ':');
#ifdef  SLIDER_VAL_DEBUG
  // NOTE: slow response with this enabled.
  DEBUG_SIO.print("Slider: ");
  DEBUG_SIO.print(slider_num, DEC);
  DEBUG_SIO.print(" val: ");
  DEBUG_SIO.println(slider_val, DEC);
#endif
  if (slider_num == DUTYCYCLE_SLIDER)
  {
    // the dutycycle slider sends a percentage 1-99
    pulse_duty_cycle = map(slider_val,
                           0, 100,
                           MIN_DUTY_CYCLE, DUTY_CYCLE_MAX-1);
  }
  else if (slider_num == DURATION_SLIDER)
  {
    // the duraction slider sends a time in milliseconds
    if (slider_val < MIN_DURATION)
    {
      pulse_duration = MIN_DURATION;
    }
    else if (slider_val > MAX_DURATION)
    {
      pulse_duration = MAX_DURATION;
    }
    else pulse_duration = slider_val;  // OK to use directly
  }
  else
  {
    pulse_duty_cycle = MIN_DUTY_CYCLE;
    pulse_duration = MIN_DURATION;
#ifdef  SLIDER_DEBUG
    DEBUG_SIO.print("Unknown slider: ");
    DEBUG_SIO.println(slider_num, DEC);
#endif
```

```
  }

}


/*
 * do_pulse:
 *
 * called to issue a pulse as specified by the current
 * pulse profile settings.
 *
 * NOTE: The direct use of processor registers makes this code
 * specific to a processor function (output OSC2B) instead of
 * an Arduino function (e.g. analogWrite on pin 3) and this
 * processor function may appear on different Arduino pins
 * depending on which processor is used.  The ifdefs at the
 * beginning of this file allow this same code to be used on a
 * Uno and on a Mega 2560.
 */
void do_pulse(void)
{

  digitalWrite(DEBUG_LED, 1);

  // set timer2 for fast PWM, start timer
  TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
  TCCR2B = _BV(CS20);  // prescler = 1

  // the pulse
  OCR2B = pulse_duty_cycle;
  delay(pulse_duration);

  OCR2B = 1;          // minimum duty cycle
  TCCR2A = _BV(0);    // stop timer

  digitalWrite(DEBUG_LED, 0);

}


/*
 * handle_footswitch:
 *
 * called to monitor and debounce the footswitch input.
 * sets trigger_set when a valid footswitch action is
 * detected (falling edge).  waits for rising edge to
 * reenable footswitch action.
 */
void handle_footswitch(void)
{

  footswitch_state = digitalRead(FOOTSWITCH_IN);
  if (footswitch_state != last_footswitch_state)
  {
    if ((footswitch_state == LOW)  && (!trigger_holdoff))
    {
      delay(FOOTSWITCH_DEBOUNCE);
      footswitch_state = digitalRead(FOOTSWITCH_IN);
      // if footswitch is still low, OK to trigger pulse
      if (footswitch_state == LOW)
      {
        trigger_set = true;
        trigger_holdoff = true;
      }
    }
    last_footswitch_state = footswitch_state;
  }
  else
  {
    // fotswitch has been released, allow new trigger
    if (footswitch_state == HIGH)
    {
      trigger_holdoff = false;
    }
  }

}


/*
 * wait_for_display:
 *
 * called to wait for display to reply to previous commands.
 * used in lieu of XON/XOFF flow control.  application is able
 * to send up to MAX_CMD_OUTSTANDING overlapping commands before
 * needing to wait for display to reply.
 */
void wait_for_display(void)
{

  while (replies_req >= MAX_CMD_OUTSTANDING)
  {
#ifdef  DEBUG_CMD_OVERLAP
    DEBUG_SIO.print("Stalled: ");
    DEBUG_SIO.println(replies_req, DEC);
#endif
    // wait until display catches up with commands
    DISPLAY_SERIALEVENT();  // must call manually
    delay(10);
  }

}


/*
 * DISPLAY_SERIALEVENT:
 *
 * called when serial data is available.  reads serial data into
 * the buffer and sets the line_ready flag when a <return> char
 * has been received.
 *
 * NOTE: This function is called by the Arduino environment each
 * time loop() is executed, after loop() is called.  If loop()
 * is waiting for serial data it will need to call this function
 * manually.
 */
void DISPLAY_SERIALEVENT(void)
{
  char  serial_char;

  while (DISPLAY_SIO.available())
  {
    serial_char = DISPLAY_SIO.read();

#undef  DEBUG_SERIAL
#ifdef  DEBUG_SERIAL
    // send same data to debug serial port
    DEBUG_SIO.print(serial_char);
    if (serial_char == '\r')
    {
      DEBUG_SIO.print("<return>");
      DEBUG_SIO.print("\n");
    }
#endif

    if (serial_char == '>')
    {
      if (replies_req)
      {
        replies_req--;
      }
    }
    // lines starting with:
    //   '!' indicates command failure
    //   '^' indicates display buffer full
    //   '?' indicates transmission line error
    //   '#' indicates CRC error
    //   ':' indicates human-readable information
    else if ((buffer_index == 0) &&
              ((serial_char == '!') ||
               (serial_char == '^') ||
               (serial_char == '?') ||
               (serial_char == '#') ||
               (serial_char == ':')))
    {
      serial_error = true;
      // additional handling depends on user application
    }
    else if (serial_char == '\r')
    {
      // end of line
      buffer[buffer_index] = '\0';  // string terminator
      if (buffer_index > 0)
      {
#ifdef  DEBUG_SERIAL
        DEBUG_SIO.println("Got a message");
#endif
        message_ready = true;
        buffer_index = 0;
      }
    }
    else
    {
      // store character
      buffer[buffer_index] = serial_char;
      buffer_index++;
    }

  }  /* while */

}
```

8 Source Code for Macro Approach

The macros in the Macro approach use the character “@” in messages sent to the application to indicate the start of a user-defined message. The application needs to coordinate with the macros on the definitions of these messages. There are three user-defined messages in this application:

- `@p` – pulse profile – in the form “`@p:<duty cycle>,<duration>\r`”, where `<duty cycle>` is the duty cycle of the pulse profile, in percentage, and where `<duration>` is either a number representing milliseconds when the main screen is active, or is a percentage when the alt screen is active.

- `@t` – trigger – in the form “`@t\r`”

- `@s` – screen ID – in the form “`@s:<new name>\r`”, where `<new name>` is either “`main`” or “`alt`”

8.1 The Macros

```
//
// ArduinoMacros.txt:
//
// macros to implement the user-interface portion of the Arduino
// application note, AN-110.
/
// these macros depend on the following bitmap files being loaded into
// the display, and in the order shown.
//
//     01_big_button.bmp
//     02_big_button_dn.bmp
//     03_DutyCycleSliderBackground.bmp
//     04_DurationSliderBackground.bmp
//     05_slider2Knob.unc.bmp
//

//
// power_on - macro defined first by convention so it will be
// macro #1
//
#define power_on
// draw main screen by calling the draw_screen macro with the
// label "x20", to simulate button 20 (MainScr) being pressed.
m draw_screen:x20
#end

//
// draw_screen - macro to draw the display screens
//
#define draw_screen
//
// common section
//
// always start by clearing the existing screen contents and
// initializing any other items (font, beep volume, etc)
```

```
z
f 13B
bvs 25
//
// the main screen contains buttons to select predefined pulse profiles.
// the main screen is called "x20" because of the button index assigned
// to the MainScr button in the alt screen.
:x20
bdc 1 4 8 1 "20%\n50ms" 1 2
bdc 2 83 8 1 "20%\n100ms" 1 2
bdc 3 162 8 1 "20%\n200ms" 1 2
bdc 4 4 72 1 "40%\n50ms" 1 2
bdc 5 83 72 1 "40%\n100ms" 1 2
bdc 6 162 72 1 "40%\n200ms" 1 2
bdc 7 4 136 1 "60%\n50ms" 1 2
bdc 8 83 136 1 "60%\n100ms" 1 2
bdc 9 162 136 1 "60%\n200ms" 1 2
bdc 10 4 200 1 "99%\n50ms" 1 2
bdc 11 83 200 1 "99%\n100ms" 1 2
bdc 12 162 200 1 "99%\n200ms" 1 2
bdc 50 399 8 1 "TRIG" 1 2
bdc 21 399 200 1 "AltScr" 1 2 //
// assign buttons to macros, execute when button is pressed.
// the parameters passed to profile_button_pressed are the
// new pulse profile duty cycle (in percent) and the new
// pulse duration (in milliseconds)
//
xaq 1 p profile_button_pressed 20 50
xaq 2 p profile_button_pressed 20 100
xaq 3 p profile_button_pressed 20 200
xaq 4 p profile_button_pressed 40 50
xaq 5 p profile_button_pressed 40 100
xaq 6 p profile_button_pressed 40 200
xaq 7 p profile_button_pressed 60 50
xaq 8 p profile_button_pressed 60 100
xaq 9 p profile_button_pressed 60 200
xaq 10 p profile_button_pressed 99 50
xaq 11 p profile_button_pressed 99 100
xaq 12 p profile_button_pressed 99 200
//
xmq 21 draw_screen:x21
xmq 50 trigger_button_pressed
//
m screen_id main
//
// the alt_screen contains two sliders to fine tune the pulse profile.
// sliders are set to notify on release only.
// the main screen is called "x21" because of the button index assigned
// to the AltScr button in the main screen.
:x21
t "Duty Cycle" 75 30
sl 128 3 70 50 5 7 0 1 1 99 1
t "Duration" 235 30
sl 129 4 230 50 5 7 0 1 1 200 50
bdc 50 399 8 1 "TRIG" 1 2
bdc 20 399 200 1 "MainScr" 1 2 //
// assign buttons to macros, execute when button is pressed
```

```
//
xmq 20 draw_screen:x20
xmq 50 trigger_button_pressed
//
// assign sliders to macro to output data in proper format when
// slider is released
//
xaq 128 r slider_released
xaq 129 r slider_released
//
m screen_id alt
#end

//
// screen_id - macro to tell the application which screen is active
// parameter is name of active screen
//
#define screen_id
out "@s:`0`\r"
#end

//
// profile_button_pressed - macro for preset profile buttons
// parameter 0 is duty cycle, parameter 1 is duration
//
#define profile_button_pressed
out "@p:`0`,`1`\r"
#end

//
// trigger_button_pressed - macro for trigger buttons
//
#define trigger_button_pressed
out "@t\r"
#end

//
// slider_released - macro for sliders.  whenever a slider is
// released, the values (in percent) of both sliders are sent
// to the application.  the duty cycle slider is index 128 and
// the duration slider is index 129 (per the assignments when
// the alt screen was drawn)
//
#define slider_released
out "@p:`L128`,`L129`\r"
#end

//
// display_pulse_profile - macro to print pulse profile
// parameter 0 is duty cycle, parameter 1 is duration
//
#define display_pulse_profile
t "dc: `0`     " 399 116
t "time: `1`     " 399 136
#end
```

8.2 The Code

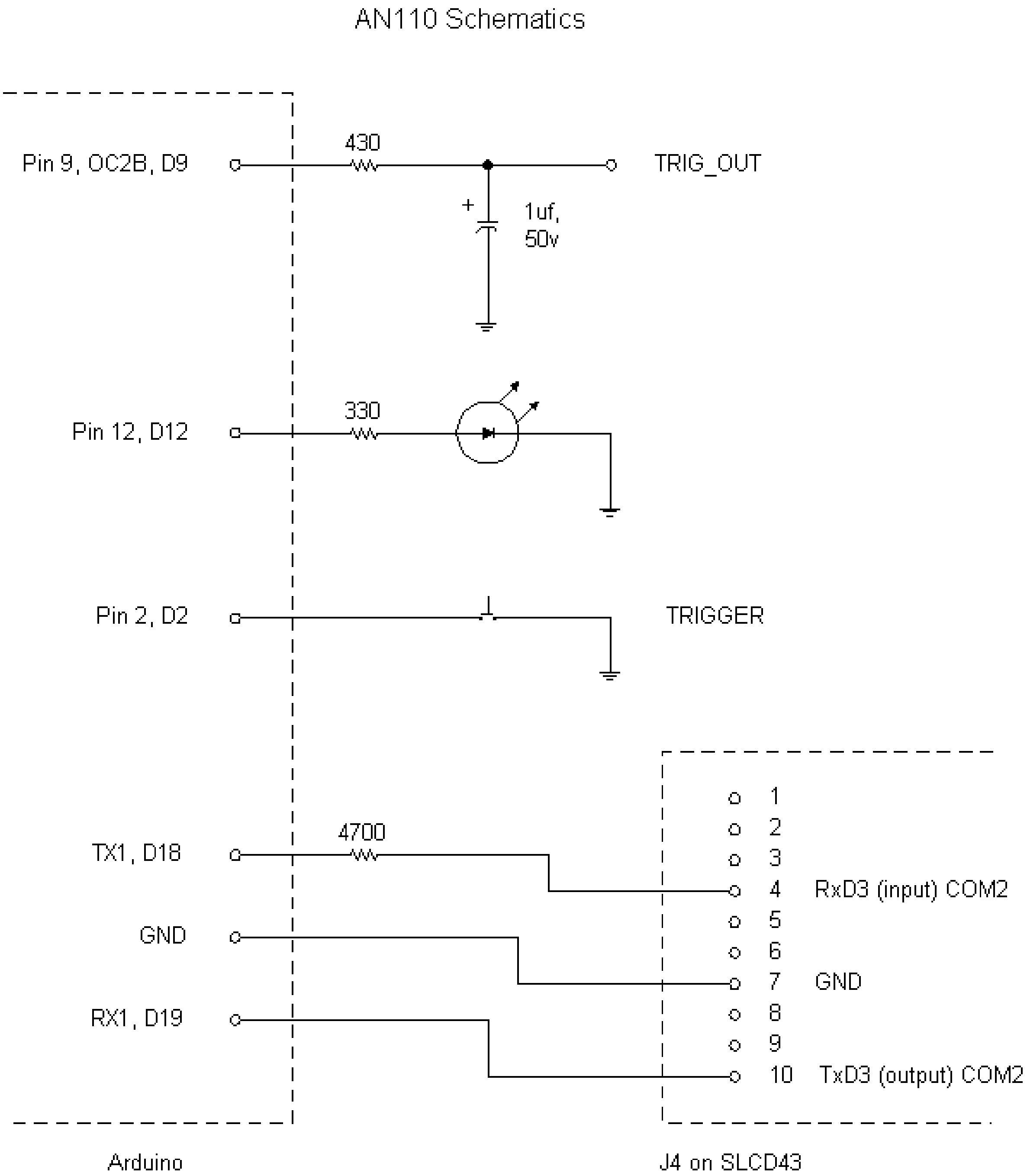
```
/*
 * ArduinoExampleMacros:
 *
 * Arduino sketch that accepts commands from a user interface on
 * a Reach Technology SLCD43 Development Kit display and then
 * drive an analog pulse with a settable pulse profile (PWM duty
 * cycle and duration).
 *
 * This sketch is an example of a high-level control application
 * that receives information from macros running on the display.
 * The information is pulse profile parameters, trigger events
 * and screen change notifications.  The sketch does not manage
 * the buttons or sliders on the display.  The sketch does need
 * to know which screen (main or alt) the display is using so it
 * can interpret the pulse profile information correctly.
 *
 * Timer2 on the Arduio is used to drive a PWM signal to create
 * the output pulse.  The PWM output from timer2 on OC2B (pin 9
 * on Mega2560) is filtered with an RC network to convert the
 * output into an analog signal.  The duty cycle is varied to
 * adjust the pulse amplitude.  The pulse duty cycle and duration
 * are based on information received from the display.
 *
 * The low-level pulse profile settings are displayed on the
 * screen each time the pulse is triggered (either by receiving
 * a trigger event from the display or by the footswitch input).
 *
 * An LED with 330 ohm current limiting resistor connected to
 * pin 12 will turn on for the selected pulse time each time a
 * pulse is triggered.
 */

// compiler-provided definitions
// __AVR_ATmega328P__ - Uno
// __AVR_ATmega1280__ - Mega
// __AVR_ATmega2560__ - Mega 2560


// notifications from display, see SLCDx Software Reference
#define  USER_MSG          '@'  // indicates start of user-defined message
#define  PULSE_PROFILE     'p'
#define  TRIGGER           't'
#define  SCREEN_CHANGE     's'

// maximum number of commands sent without getting a reply
#define  MAX_CMD_OUTSTANDING  2

// duty cycle definitions assuming 'fast PWM' config
#define  DUTY_CYCLE_MAX     255
#define  DUTY_CYCLE_20PCNT  ((DUTY_CYCLE_MAX*20)/100)
//#define  DUTY_CYCLE_40PCNT  ((DUTY_CYCLE_MAX*40)/100)
//#define  DUTY_CYCLE_60PCNT  ((DUTY_CYCLE_MAX*60)/100)
//#define  DUTY_CYCLE_99PCNT  ((DUTY_CYCLE_MAX*99)/100)
#define  MIN_DUTY_CYCLE     DUTY_CYCLE_20PCNT
```

```
// duration literals
#define  MIN_DURATION      50      // ms
#define  MAX_DURATION      200     // ms

// length of serial input buffer
#define  BUFF_SIZE        100

// pin definitions
#define  FOOTSWITCH_IN    2
#define  PWM_OUT          9  // timer2 OC2B output
#define  DEBUG_LED        12

// serial IO port mapping literals
#define  DISPLAY_SIO          Serial1
#define  DISPLAY_SERIALEVENT  serialEvent1
#define  DEBUG_SIO            Serial    // to use IDE's Serial Monitor

// debug support
#undef  SHOW_DEBUG
#ifdef  SHOW_DEBUG
// these are undefined if SHOW_DEBUG is undefined,
// otherwise may be defined or not as needed
#define  USER_MESSAGE_DEBUG
#define  PULSE_PROFILE_DEBUG
#define  DEBUG_SCREEN_NAME
#undef   DEBUG_CMD_OVERLAP
#endif

// misc
#define  FOOTSWITCH_DEBOUNCE  50  // time in milliseconds

// variables and flags
boolean  main_screen;
byte footswitch_state;
byte last_footswitch_state;
boolean trigger_set;
boolean trigger_holdoff;
byte pulse_duty_cycle;
int pulse_duration;
char buffer[BUFF_SIZE];
int  buffer_index;
char  replies_req;
boolean message_ready;
boolean serial_error;

// function prototypes
void handle_pulse_profile(void);
int decode_ASCII(char buffer[], int *index, char delim);
void print_pulse_profile(void);
void do_pulse(void);
void handle_footswitch(void);
void wait_for_display(void);
void DISPLAY_SERIALEVENT(void);


/*
```

```
 * setup:
 *
 * the required Arduino sketch setup() function.
 */
 void setup()
{

  // timer2 output pin
  pinMode(PWM_OUT, OUTPUT);

  // footswitch input, with internal 20k pull-up
  pinMode(FOOTSWITCH_IN, INPUT);
  digitalWrite(FOOTSWITCH_IN, HIGH);

  // debug LED
  pinMode(DEBUG_LED, OUTPUT);

  // init the serial port and send a few <return>s
  // to get the display's attention
  DISPLAY_SIO.begin(115200);
  DISPLAY_SIO.print('\r');
  DISPLAY_SIO.print('\r');
  DISPLAY_SIO.print('\r');

#ifdef  SHOW_DEBUG
  DEBUG_SIO.begin(9600);
  DEBUG_SIO.println("Showing debug");
#endif

  footswitch_state = digitalRead(FOOTSWITCH_IN);
  last_footswitch_state = footswitch_state;
  trigger_set = false;
  trigger_holdoff = false;
  pulse_duty_cycle = MIN_DUTY_CYCLE;
  pulse_duration = MIN_DURATION;
  buffer_index = 0;
  replies_req = 0;
  message_ready = false;
  serial_error = false;

  // tell the screen to initialize itself
  DISPLAY_SIO.println("m power_on");
  replies_req++;
  wait_for_display();

}


/*
 * loop:
 *
 * the required Arduino loop() function.
 */
void loop()
{
  int  i, num_chars, index, pp_dc, pp_dur;
  // check for display notifications
  if (message_ready)
  {
//    DEBUG_SIO.println("Handling a message");
    message_ready = false;
    if (buffer[0] == USER_MSG)
    {
#ifdef  USER_MESSAGE_DEBUG
      DEBUG_SIO.println(buffer);
#endif
      switch (buffer[1])
      {
        case PULSE_PROFILE:
          handle_pulse_profile();
          break;
        case TRIGGER:
          trigger_set = true;
          break;
        case SCREEN_CHANGE:
          // screen change messages are in the form "@s:name\r"
          // where 'name' is "main" or "alt".  we'll take the
          // short-cut of only looking at the first character
          // of the screen name.  then, if needed, code can do
          // something like:
          //    if (main_screen)
          //    {
          //    }
          //    else
          //    {
          //    }

          if (buffer[3] == 'm')
          {
            main_screen = true;
          }
          else if (buffer[3] == 'a')
          {
            main_screen = false;
          }
          else
          {
#ifdef  DEBUG_SCREEN_NAME
            DISPLAY_SIO.println("z");
            DISPLAY_SIO.print("t \"Bad @s command: ");
            DISPLAY_SIO.print(buffer);
            DISPLAY_SIO.println("\" 100 100");
#endif
          }
          break;
        default:
          // ignore unknown user messages
          break;
      }
    }
  }

  // check for footswitch trigger
  handle_footswitch();

  // trigger pulse if flag is set
  if (trigger_set)
  {
    print_pulse_profile();
    trigger_set = false;
    do_pulse();
  }

}


/*
 * handle_pulse_profile:
 *
 * called to handle pulse profile user commands.  decodes the two
 * profile values depending on which screen (main or alt) is active.
 */
void handle_pulse_profile(void)
{
  int  index, pp_dc, pp_dur;

  // pulse profile messages are in the form "@p:dc,dur\r"
  // buffer[2] is ':'
  index = 3;
  pp_dc = decode_ASCII(buffer, &index, ',');
  index++;    // skip delimiter
  pp_dur = decode_ASCII(buffer, &index, ',');
  // pp_dc is the duty cycle percentage (1-99)
  // and pp_dur is the duration in ms (50-200)
  pulse_duty_cycle = map(pp_dc,
                         0, 100,
                         MIN_DUTY_CYCLE, DUTY_CYCLE_MAX-1);
  if (pp_dur < MIN_DURATION)
  {
    pulse_duration = MIN_DURATION;
  }
  else if (pp_dur > MAX_DURATION)
  {
    pulse_duration = MAX_DURATION;
  }
  else pulse_duration = pp_dur;  // OK to use directly

}


/*
 * decode_ASCII:
 *
 * called to decod the ASCII numeric value at buffer[index].
 * returns the decoded value, and leaves index set to the
 * delimiter or the zero terminator.
 *
 * assumes that there are no more than three ASCII characters.
 */
int decode_ASCII(char buffer[], int *index, char delim)
```

```
{
  int  val, l_index;

  l_index = *index;
  val = buffer[l_index++] - '0';
  if ((buffer[l_index] != delim) && (buffer[l_index] != 0))
  {
    val = (val * 10) + (buffer[l_index++] - '0');
  }
  if ((buffer[l_index] != delim) && (buffer[l_index] != 0))
  {
    val = (val * 10) + (buffer[l_index++] - '0');
  }

#ifdef  PULSE_PROFILE_DEBUG
  DEBUG_SIO.print("decode_ASCII: ");
  DEBUG_SIO.print(val, DEC);
  DEBUG_SIO.print(" ending index: ");
  DEBUG_SIO.println(l_index, DEC);
#endif

  *index = l_index;
  return(val);

}


/*
 * print_pulse_profile:
 *
 * called to print the current low-level pulse profile
 */
void print_pulse_profile(void)
{

  wait_for_display();
  DISPLAY_SIO.print("m display_pulse_profile ");
  DISPLAY_SIO.print(pulse_duty_cycle, DEC);
  DISPLAY_SIO.print(" ");
  DISPLAY_SIO.println(pulse_duration, DEC);
  replies_req++;

}


/*
 * do_pulse:
 *
 * called to issue a pulse as specified by the current
 * pulse profile settings.
 *
 * NOTE: The direct use of processor registers makes this code
 * specific to a processor function (output OSC2B) instead of
 * an Arduino function (e.g. analogWrite on pin 3) and this
 * processor function may appear on different Arduino pins
 * depending on which processor is used.  The ifdefs at the
 * beginning of this file allow this same code to be used on a
 * Uno and on a Mega 2560.
 */
void do_pulse(void)
{

  digitalWrite(DEBUG_LED, 1);

  // set timer2 for fast PWM, start timer
  TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
  TCCR2B = _BV(CS20);  // prescler = 1

  // the pulse
  OCR2B = pulse_duty_cycle;
  delay(pulse_duration);

  OCR2B = 1;          // minimum duty cycle
  TCCR2A = _BV(0);    // stop timer

  digitalWrite(DEBUG_LED, 0);

}


/*
 * handle_footswitch:
 *
 * called to monitor and debounce the footswitch input.
 * sets trigger_set when a valid footswitch action is
 * detected (falling edge).  waits for rising edge to
 * reenable footswitch action.
 */
void handle_footswitch(void)
{

  footswitch_state = digitalRead(FOOTSWITCH_IN);
  if (footswitch_state != last_footswitch_state)
  {
    if ((footswitch_state == LOW)  && (!trigger_holdoff))
    {
      delay(FOOTSWITCH_DEBOUNCE);
      footswitch_state = digitalRead(FOOTSWITCH_IN);
      // if footswitch is still low, OK to trigger pulse
      if (footswitch_state == LOW)
      {
        trigger_set = true;
        trigger_holdoff = true;
      }
    }
    last_footswitch_state = footswitch_state;
  }
  else
  {
    // fotswitch has been released, allow new trigger
    if (footswitch_state == HIGH)
    {
      trigger_holdoff = false;
    }
  }

}


/*
 * wait_for_display:
 *
 * called to wait for display to reply to previous commands.
 * used in lieu of XON/XOFF flow control.  application is able
 * to send up to MAX_CMD_OUTSTANDING overlapping commands before
 * needing to wait for display to reply.
 */
void wait_for_display(void)
{

  while (replies_req >= MAX_CMD_OUTSTANDING)
  {
#ifdef  DEBUG_CMD_OVERLAP
    DEBUG_SIO.print("Stalled: ");
    DEBUG_SIO.println(replies_req, DEC);
#endif
    // wait until display catches up with commands
    DISPLAY_SERIALEVENT();  // must call manually
    delay(10);
  }

}


/*
 * DISPLAY_SERIALEVENT:
 *
 * called when serial data is available.  reads serial data into
 * the buffer and sets the line_ready flag when a <return> char
 * has been received.
 *
 * NOTE: This function is called by the Arduino environment each
 * time loop() is executed, after loop() is called.  If loop()
 * is waiting for serial data it will need to call this function
 * manually.
 */
void DISPLAY_SERIALEVENT(void)
{
  char  serial_char;

  while (DISPLAY_SIO.available())
  {
    serial_char = DISPLAY_SIO.read();

#undef  DEBUG_SERIAL
#ifdef  DEBUG_SERIAL
    // send same data to debug serial port
    DEBUG_SIO.print(serial_char);
    if (serial_char == '\r')
    {
      DEBUG_SIO.print("<return>");
      DEBUG_SIO.print("\n");
    }
#endif

    if (serial_char == '>')
    {
      if (replies_req)
      {
        replies_req--;
      }
    }
    // lines starting with:
    //   '!' indicates command failure
    //   '^' indicates display buffer full
    //   '?' indicates transmission line error
    //   '#' indicates CRC error
    //   ':' indicates human-readable information
    else if ((buffer_index == 0) &&
              ((serial_char == '!') ||
               (serial_char == '^') ||
               (serial_char == '?') ||
               (serial_char == '#') ||
               (serial_char == ':')))
    {
      serial_error = true;
      // additional handling depends on user application
    }
    else if (serial_char == '\r')
    {
      // end of line
      buffer[buffer_index] = '\0';  // string terminator
      if (buffer_index > 0)
      {
#ifdef  DEBUG_SERIAL
        DEBUG_SIO.println("Got a message");
#endif
        message_ready = true;
        buffer_index = 0;
      }
    }
    else
    {
      // store character
      buffer[buffer_index] = serial_char;
      buffer_index++;
    }

  }  /* while */

}
```

9 Schematics



10 Photo of Adapter/Proto Board