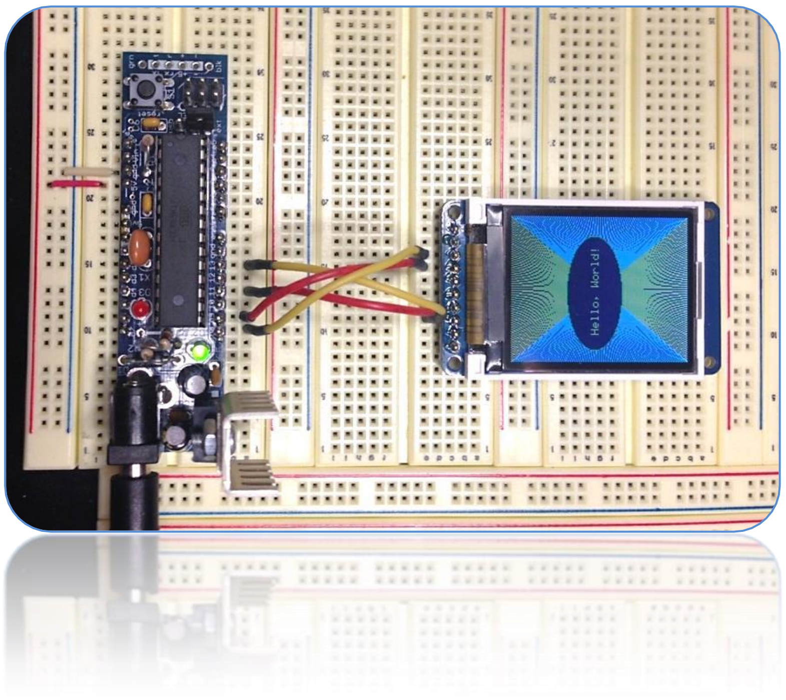


Add a TFT Display to your AVR Micro

Bruce E. Hall, W8BH

Objective: control a 128x160 pixel TFT LCD module from your AVR microcontroller, using C.

1) INTRODUCTION

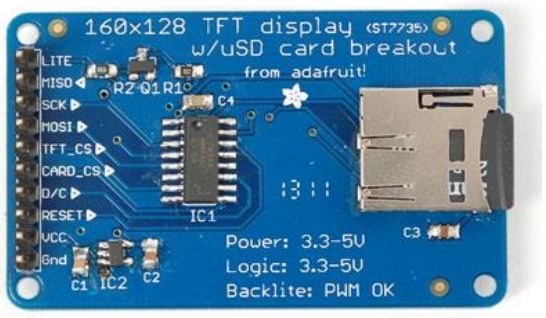
I am a big fan of 2x16 character-based LCD displays. They are simple, cheap, and readily available. Send ASCII character data to them, and they display the characters. Nice.

But sometimes characters are not enough. Colors, images, graphs, even text with fonts all require something more. The 1.8" TFT module from Adafruit (and others) gives you this option for the reasonable price of $25. Just buy one and have some fun learning what you can do with it.

This TFT module is a 128x160 LCD matrix, controlled by the onboard Sitronix ST7735 controller. You send data to it serially using the Serial-Peripheral Interface (SPI) protocol. Simple, right? Unfortunately, it's not as simple as writing to a character mode display.

I started with two documents: the Siltronix datasheet and the Adafruit library. Take a look at both of them. For me, both are bit complicated. Even the initialization code looks like a programming nightmare. I like to start simple, and build as I go. Here is my approach.

2) MAKE THE CONNECTIONS



Let's connect the hardware first. The Adafruit module has 10 pins. On the bottom of the module each pin is labeled, from pin 1 'Lite' to pin 10 'Gnd'. Mount the display module on a breadboard and connect the pins to your

AVR micro. Male to male prototyping wires are very handy for making these point-to-point connections.

First, apply 5V power to the backlight, pin 1 and ground to pin 10. You should see the glow of the backlight when you do this. If not, check your power connections before proceeding further.

Next, connect Vcc to 5V and the TFT select line to Gnd. Finally, hook up the four data lines: SCLK, MOSI, DC, and Reset. On Arduino-compatible boards, these connect to the digital 13, 11, 9, and 8 lines. On other AVR controllers, these lines may be labelled by their position on Port B: PB5, PB3, PB1, and PB0.

TFT pin	Function	AVR micro
1	Backlight	5V
2	MISO	
3	SCK	D13 (PB5)
4	MOSI	D11 (PB3)
5	TFT select	Gnd
6	SD select	
7	D/C	D9 (PB1)
8	Reset	D8 (PB0)
9	Vcc	5V
10	Gnd	Gnd

3) SERIAL PERIPHERAL INTERFACE (SPI)

Finding a good starting point is sometimes the hardest part! I chose the SPI protocol, since any data transfer to the TFT module would require this. There is a good overview of SPI using AVR micros at avrbeginners.net. The Arduino, and most AVR microcontrollers, have a fast, built-in hardware SPI interface which is easy to use.

At its core, the SPI algorithm is very straightforward:

- Put a data bit on the serial data line.
- Pulse the clock line.
- Repeat for all the bits you want to send, usually 8 bits at a time.

You must set the microcontroller's SPI control register (SPCR) to enable SPI communication. This is an eight-bit register that contains the following bits:

SPCR = 0x50:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
SPIE	**SPE**	DORD	**MSTR**	CPOL	CPHA	SPR1	SPR0
0	**1**	0	**1**	0	0	0	0

The first bit on the left, SPIE, enables the SPI interrupt and is not needed for this application. The SPE bit enables SPI. DORD determines the data direction: when 0, the most-significant bit is sent & received first. MSTR determines if the micro acts as a master (1) or slave (0) device. CPOL and CPHA together determine the transfer mode. Our TFT display works well with Mode 0, in which both bits are zero. Finally, SPR1 and SPR0 determine the transfer speed, as a fraction of the microcontroller's oscillator. When both are 0, the SPI transfer speed is osc/4, which on my 16 MHz micro is 16/4 = 4 MHz. When both bits are 1, the transfer speed is osc/256 = 62.5 kHz.

Using an SPCR value of 0x50, SPI is enabled as Master, in Mode 0 at 4 MHz. The code to open SPI communication can be as simple as the following:

```
void OpenSPI()
{
    SPCR = 0x50;                       // SPI enabled as Master, Mode0 at 4 MHz
}
```

To close SPI, just set the SPE bit to 0. This will stop SPI and return the four dedicated SPI lines (MOSI, MISO, SCLK, SS) to the general purpose I/O functions:

```
void CloseSPI()
{
    SPCR = 0x00;                       // clear SPI enable bit
}
```

Only one more routine is needed: the SPI transfer routine. SPI is a bidirectional protocol, with two separate data lines. The data is transmitted over MOSI and received over MISO *at the same time*. Even if we only want to send, we are always going to receive. And vice versa. If you aren't expecting any received data, just ignore what is returned to you.

The data transfer register is SPDR. Load this register with a value, and the data transfer will start automatically. A bit in SPSR, the status register, will tell us when the transfer is complete. As the data bits are serially shifted out of the transfer register, the received bits are shifted in. When the transfer completes, SPDR will hold the received data:

```
byte Xfer(byte data)                   // you can use uint8_t for byte
{
    SPDR = data;                       // initiate transfer
    while (!(SPSR & 0x80));            // wait for transfer to complete
    return SPDR;
}
```

The three routines above are all we need for SPI. Let's make sure they work by doing a serial loop-back test. In this test, the output data on MOSI is looped-back as the input on MISO. Whatever value we put into the data register should come right back in.

Without a working display, we need a way to verify the data. You might want to use your fancy debugger, or send the value to a monitor via UART, but here is something even simpler: flash the LED on your controller board. Most AVR boards have a connected LED. On many AVR boards, including the Arduino, the status LED is on PB5. Here is a routine to flash it:

```
void FlashLED(byte count)
// flash the on-board LED at ~ 2 Hz
{
    DDRB |= _BV(DDB5);                   // Set PB5 as output
    for (;count>0;count--)
    {
         PORTB |= _BV(PORTB5);           // turn LED on
         _delay_ms(250);                 // wait
         PORTB &= ~_BV(PORTB5);          // turn LED off
         _delay_ms(250);                 // wait
    }
}
```

Now, disconnect the microcontroller’s MOSI (digital 11, PB3) from the TFT display, and connect it to the microcontroller’s MISO line (digital 12, PB4). Run the following code:

```
int main()
{
    OpenSPI();                          // start communication to TFT
    char i = Xfer(5);                   // MISO to MOSI -> returns 5
                                        // MISO to +5V  -> returns 255
                                        // MISO to Gnd  -> returns 0
    CloseSPI();                         // return portB lines to general use
    FlashLED(i+1);                      // flash (returned value + 1)
}
```

What happens? If all goes well, the LED will flash 6 times. The value 5 is sent out the MOSI line, comes back in on the MISO line, and is returned from the SPI xfer routine.

You may wonder if Xfer worked at all. Maybe nothing was transferred: the value 5 could have stayed in the transfer register ‘untouched’. How can we know for sure?

For the doubters out there like me, take your wire on the MISO line and put to ground (logic 0). Now, all bits shifted-in will be 0, and the value returned should be 0x00000000 = 0. If you run the program now, the LED should flash only once. To further convince you, connect MISO to +5V. Now, all bits shifted-in will be one, and the value returned will always be 0x11111111 = 255. The LED should not flash at all, since 255+1 = 256 = 0, for byte-sized variables.

4) SENDING COMMANDS & DATA

Serial information sent to the display module can be either commands or data. For commands, the D/C (data/command) input must be 0; for data, the input must be 1. We use a third microcontroller pin, D9/PB1, to supply this information. Here are the two routines:

```
#define ClearBit(x,y) x &= ~_BV(y)      // equivalent to cbi(x,y)
#define SetBit(x,y) x |= _BV(y)         // equivalent to sbi(x,y)

void WriteData (byte b)
{
    Xfer(b);                            // assumes DC (PB1) is high
}

void WriteCmd (byte cmd)
{
    ClearBit(PORTB,1);                  // 0=command, 1=data
    Xfer(cmd);
    SetBit(PORTB,1);                    // return DC high
}
```

I use two handy macros, ClearBit and SetBit, for clearing & setting individual bits. The WriteData routine just calls Xfer; it’s nice to have as an abstraction from the SPI layer, but… I don’t use it. We want to send data as fast as possible, and calling this routine instead of Xfer just adds time (and stack use). WriteCmd, on the other hand, makes sure that the Data/Command line is appropriately cleared before the command and set afterwards.

5) ST7735 TFT CONTROLLER

The Sitronix ST7735 is a single-chip driver/controller for 128x160 pixel TFT-LCD displays. It can accept both serial and 8/9/16/18 bit parallel interfaces. On my module, and many others like it, only the serial interface over SPI is supported. The Sitronix datasheet refers to this interface as the "four-wire" SPI protocol.

The ST7735 supports over 50 different commands. Many of these commands fine-tune the power output and color intensity settings, allowing you to correct for LCD display variations. In this tutorial we need only six of those commands.

6) INITIALIZING THE DISPLAY

You should initialize the display before sending pixel data. I found a few code samples online, but they are a bit confusing. My favorite library, the Adafruit-ST7735-Library on GitHub, calls 19 different commands with over 60 parameters! Let's find an easier solution.

The first initialization step is to reset the controller, either by hardware or software. A hardware reset requires an additional GPIO line to pulse the controller's reset pin. A software reset is a byte-sized command sent to the controller. I chose the hardware reset because of its reliability. The reset function initializes the controller registers to their default values. See the reset table in datasheet section 9.14.2 for more information. To do a hardware reset, take the reset line briefly low, and wait enough time for the reset to complete:

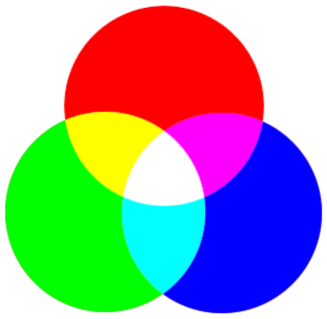
```
void HardwareReset()
{
    ClearBit(PORTB,0);                  // pull PB0 (digital 8) briefly low
    msDelay(1);                         // 1 mS is enough
    SetBit(PORTB,0);                    // return PB0 high
    msDelay(200);                       // wait 200 mS for reset to finish
}
```

After the reset, the controller enters a low-power sleep mode. We wake the controller and turn on its TFT driver circuits with the sleep out SLPOUT command. Next, we set the controller to accept 16-bit pixel data. More on that below. Finally, after turning on the driver circuits, we need to enable display output with the DISPON (display on) command. Here is the code for our simplified initialization routine:

```
void InitDisplay()
{
    HardwareReset();                    // initialize display controller
    WriteCmd(SLPOUT);                   // take display out of sleep mode
    msDelay(150);                       // wait 150mS for TFT driver circuits
    WriteCmd(COLMOD);                   // select color mode:
    WriteByte(0x05);                    // mode 5 = 16bit pixels (RGB565)
    WriteCmd(DISPON);                   // turn display on!
}
```

Now is a good time to check your hardware connections and run a program that calls InitDisplay(). Your display should briefly blank, and then show a screen full of tiny, random color pixels. If it does, you have successfully initialized the display. If not, check your hardware connections. It's easy to get a couple I/O pins reversed, or forget a power/ground connection. Once you get it working, it's time to send some real data.

7) DISPLAY COLOR MODES



The default color mode for this controller is RGB666. Pixel colors are a combination of red, green, and blue color values. Each subcolor has 6 bits (64 different levels) of intensity. Equal amounts of red, green, and blue light produce white. Equal amounts of blue and green produce cyan. Red and green make yellow. Since each color component is specified by 6 bits, the final color value is 18 bits in length. The number of possible color combinations in RGB666 colorspace is 2^18 = 262,144 or 256K.

We represent these color combinations as an 18 bit binary number. The 6 red bits are first, followed by 6 green bits, followed by 6 blue bits. Our controller wants to see data in byte-sized chucks, however. For every pixel we must send 24 bits (3 bytes), arranged as follows:

r	r	r	r	r	r	-	-	g	g	g	g	g	g	-	-	b	b	b	b	b	b	-	-

The lowest two bits of each red, green, and blue byte are ignored; only the 6 upper bits of each byte are used. That's a bit wasteful, isn't it? It takes time to send those empty bits.

The TFT controller supports three different color depths. Here they are:

COLOR MODE	RGB SPACE	Bits per Pixel	Unique Colors
3	RGB444	12	16x16x16 = 4K
5	RGB565	16	32x64x32 = 64K
6 (default)	RGB666	18	64x64x64 = 256K

Mode 6 is the default, requiring three bytes per pixel. Notice that Mode 5 is 16 bits in size, which is exactly 2 bytes in length. No waste! And it still provides for plenty of colors. If we use mode 5 instead of mode 6, each pixel can be sent in 2 bytes instead of 3. Data transmission will be faster, at the cost of less color depth. If 65,536 different colors are enough for you, this is a good tradeoff. Let's use it. Here is how the red, green, and blue bits are packed into a 2-byte word:

r	r	r	r	r	g	g	g	g	g	g	b	b	b	b	b

8) PIXELS

Sending pixel data is as simple as sending both bytes of our 16-bit color, MSB first. We often want to send more than one pixel of the same color, so it can be helpful to add a counter loop:

```
void Write565 (int data, unsigned int count)
{
    for (;count>0;count--)
```

```
        {
            WriteByte (data >> 8);            // write hi byte
            WriteByte (data & 0xFF);          // write lo byte
        }
    }
```

To save CPU cycles, you may call Xfer directly, instead of WriteByte. Even faster, you can inline the Xfer code for each WriteByte call:

```
    void Write565 (int data, unsigned int count)
    // note: inlined spi xfer for optimization
    {
        for (;count>0;count--)
        {
            SPDR = (data >> 8);            // write hi byte
            while (!(SPSR & 0x80));        // wait for transfer to complete
            SPDR = (data & 0xFF);          // write lo byte
            while (!(SPSR & 0x80));        // wait for transfer to complete
        }
    }
```

We must give screen coordinates before sending pixel data to the controller. The coordinates are not a single (x,y) location, but a rectangular region. To specify the region we need the controller commands CASET and RASET. The Column Address Set command sets the column boundaries, or x coordinates. The Row Address Set sets the row boundaries, or y coordinates. The two together set the display region where new data will be written.

```
    void SetAddrWindow(byte x0, byte y0, byte x1, byte y1)
    {
        WriteCmd(CASET);                   // set column range (x0,x1)
        WriteWord(x0);
        WriteWord(x1);
        WriteCmd(RASET);                   // set row range (y0,y1)
        WriteWord(y0);
        WriteWord(y1);
    }
```

We need to specify an active region, whether we’re filling a large rectangle or just a single pixel. First, specify the region; next, issue a RAMWR (memory write) command; and finally, send the raw pixel data. Notice how the following routines are similar:

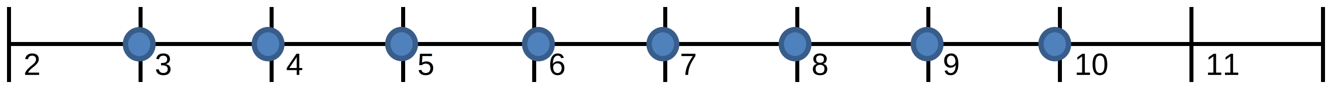
```
    void DrawPixel (byte x, byte y, int color)
    {
        SetAddrWindow(x,y,x,y);            // set active region = 1 pixel
        WriteCmd(RAMWR);                   // memory write
        Write565(color,1);                 // send color for this pixel
    }

    void FillRect (byte x0, byte y0, byte x1, byte y1, int color)
    {
        byte width = x1-x0+1;              // rectangle width
        byte height = y1-y0+1;             // rectangle height
        SetAddrWindow(x0,y0,x1,y1);        // set active region
        WriteCmd(RAMWR);                   // memory write
        Write565(color,width*height);      // set color data for all pixels
    }
```

Both routines set the window, issue a memory write command, and then send the color data. For DrawPixel, the active window is a single pixel and only a single color value is sent. For FillRect, the active window is the entire rectangle, and the color value is sent width*height times. The RAMWR is always called before Write565, so from now on it will be issued from inside the Write565 routine.

9) LINES

The simplest line to draw is a horizontal line along the x axis. For example, from x=3 to x=10. The length of this line is the difference, 10-3 = 7. Add one pixel so that the start-pixel and end-pixel are both included. The total length is 8 pixels.



To code it, set up a display window between the two positions on the x axis:

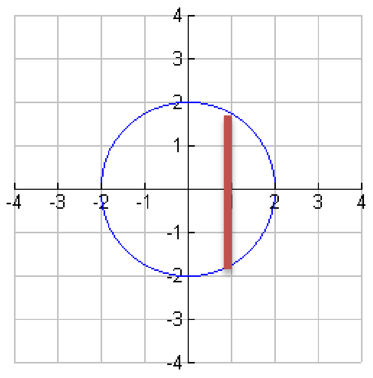
```
void HLine (byte x0, byte x1, byte y, int color)
{
    byte width = x1-x0+1;
    SetAddrWindow(x0,y,x1,y);
    Write565(color,width);
}
```

Vertical lines are exactly the same, swapping the x and y axes.

```
void VLine (byte x, byte y0, byte y1, int color)
{
    byte height = y1-y0+1;
    SetAddrWindow(x,y0,x,y1);
    Write565(color,height);
}
```

Other lines can be plotted by using the general linear equation *y=mx+b*. To use this method, iterate over each x from x0 to x1, calculate y, and draw the (x,y) pixel. The problem is that the slope of the line, m, is a floating-point number. Our 8-bit microcontroller cannot efficiently handles "floats". A line-drawing method that uses integers will be much faster and take less code space. The integer-only line drawing algorithm was first described by Jack Bresenham in 1962. The source code contains a Line routine based on the Bresenham algorithm.

10) CIRCLES AND ELLIPSES



The algebraic equation for a circle is $x^2 + y^2 = r^2$, where r is the radius. It's easiest to think of the center of circle at the (0,0) origin. For example, here is the graph of $x^2 + y^2 = 4$. The radius, the distance from the origin, is 2.

To create a filled circle, let's do the right half of the circle first, starting from x=0 and ending with x=2. For each x value we will draw a vertical line from +y to –y, shown as a red line on the graph, where y is calculated from the circle equation:

```
long r2 = radius * radius;
for (int x=0; x<=radius; x++)
{
    byte y = intsqrt(r2-x*x);
    VLine(x,y,-y,color);
}
```

This code creates the right half of the circle. The left half of the circle is a mirror image. All we need to add is a second vertical line at –x for each +x:

```
long r2 = radius * radius;
for (int x=0; x<=radius; x++)
{
    byte y = intsqrt(r2-x*x);
    VLine(x,y,-y,color);
    VLine(-x,y,-y,color);
}
```

Finally, we must translate this circle from (0,0) to any arbitrary xPos,yPos position. Substitute (xPos+x) for x and (yPos+y) for y in the VLine calls. See the source code for the completed FillCircle routine.

To create non-filled circles, use DrawPixel instead of VLine, drawing only the 'ends' of the line:

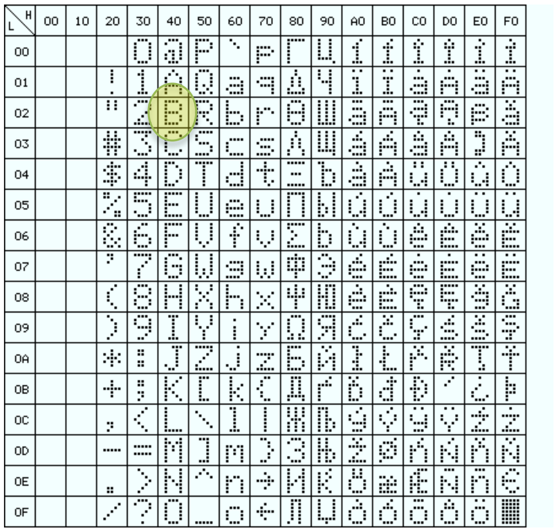
```
long r2 = radius * radius;
for (int x=0; x<=radius; x++)
{
    byte y = intsqrt(r2-x*x);
    DrawPixel(x,y,color);       // VLine(x,y,-y,color)
    DrawPixel(x,-y,color);
    DrawPixel(-x,y,color);      // VLine(-x,y,-y,color);
    DrawPixel(-x,-y,color);
}
```

You can halve the number of square root calculations, exploiting the symmetry between the top and bottom halves of each quadrant. See the source code for the completed Circle() routine.

Ellipses can be constructed in a similar way, using the formula $x^2/ a^2 + y^2/ b^2 = 1$. Instead, I chose the mid-point circle algorithm. Obviously, you can use this algorithm for circles too!

11) TEXT

The TFT controller does not have any command for 'Draw the letter B'. Unlike the 16x2 LCD character display modules, we cannot send it ASCII code 0x42 and expect to see the corresponding 'B' appear on the display. Instead, we must send each pixel dot that forms the B character.



Figuring out each character dot would be tedious work. Fortunately it was done many years ago! Tables like the one shown map out each character on pixel grids of various sizes. The granddaddy of them all is the 5x7 ASCII font, in which each character is (at most) 5 pixels wide and 7 pixels tall.

So how do we make a 'B'?

1	1	1	1	
1				1
1				1
1	1	1	1	
1				1
1				1
1	1	1	1	

Here is a 5x7 matrix for an uppercase B. Each matrix pixel will be represented by a bit. We can consider the whole character as a series of 35 individual bits. For simplicity, these bits are usually packaged in 8-bit bytes, with each row (or column) represented by 1 or more bytes.

There are two basic formats for font data. Row major order means that the bits for the first row come first, then the second row, etc. Using Row major order, each 5x7 character is a 7 element list of row bytes, with 5 active bits in each byte and 3 bits unused. Each character requires 7 bytes.

The second format is Column major order. The bits of the first column are represented first, then the second column, etc. Using column-major order, a 5x7 character is a 5 element list of column bytes, with 7 active bits in each byte and 1 bit unused. Most representations of 5x7 characters use column major order, since it requires only 5 bytes to represent each character.

Our B character is represented by 5 columns, left-most column first. Consider the third column, shown here horizontally, with the top pixel on the left:

1	0	0	1	0	0	1

There are 3 active pixels in blue. Substituting 1 for blue and 0 for white, the binary value is 0x1001001 or hexadecimal 0x49. The entire character is represented by an array of the five columns, or {0x7F, 0x49, 0x49, 0x49, 0x36}.

I created a array of 96 characters from the standard 5x7 ASCII character set above and saved them in FONT_CHAR[96][5]. This array contains 96*5 = 480 bytes of information, which would use up much of the available data memory. For this reason, I stored it in program memory instead, using routines from <pgmspace.h> to access the data.

12) DRAW A CHARACTER

Here is the pseudocode for drawing a character:

```
• Set the display window to a 5x7 pixel region
• For each row & column bit,
    o If it's a 0, set the pixel color to the background color
    o If it's a 1, set the pixel color to the foreground color
    o Write the pixel to the display
```

That is fairly straightforward. Here is the code:

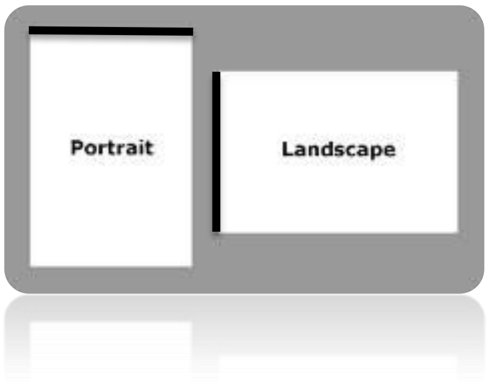
```
void PutCh (char ch, byte x, byte y, int color)
// write ch to display X,Y coordinates using ASCII 5x7 font
{
    int  pixel;
    byte row, col, bit, data, mask = 0x01;
    SetAddrWindow(x,y,x+4,y+6);
    WriteCmd(RAMWR);
```

```
        for (row=0; row<7;row++)
        {
            for (col=0; col<5;col++)
            {
                data = pgm_read_byte(&(FONT_CHARS[ch-32][col]));
                bit = data & mask;
                if (bit==0) pixel=BLACK;
                else        pixel=color;
                WriteWord(pixel);
            }
            mask <<= 1;
        }
    }
```

The font data is stored as an array of 96 characters. Since the characters are stored in ASCII order, starting with the space character (ASCII 32), the index to each character in the list is: ord(ch)-32. The character information returned into data is a byte containing the 7 pixels in the current column.

In my font table, the columns are encoded with the *least significant bit* at the top, so we must lay down row pixel data starting with bit0, then row data for bit1, etc. This can be accomplished by creating a mask byte with bit0 initially set (mask = 0x01), then moving the bit leftward for each successive row (mask <<= 1). Other tables may put the msb at the top instead. In such tables you would start with a mask on bit 7 (mask = 0x80) and move the bit rightward for each successive row (mask >>= 1).

13) PORTRAIT MODE



Finally, it is time to fill our screen with text! How many characters will fit on the display?

The default layout is portrait, with the display connectors along the top of the module. The display width is 128 pixels. Each character is 5 pixels wide. Allowing for one pixel space between characters, we can fit 128/6 = 21 characters per row.

The display height is 160 pixels. Each character is 7 pixels tall. Allowing for one pixel space between rows, we can fit 160/8 = 20 rows of characters.

The total number of characters per screen is therefore 21 chars/row x 20 rows = 420 characters. Here is a routine to display 420 characters on the screen:

```
    void PortraitChars()
    {
        for (int i=420;i>0;i--)
        {
            byte x= i % 21;
            byte y= i / 21;
            char ascii = (i % 96)+32;
            PutCh(ascii,x*6,y*8,CYAN);
        }
    }
```

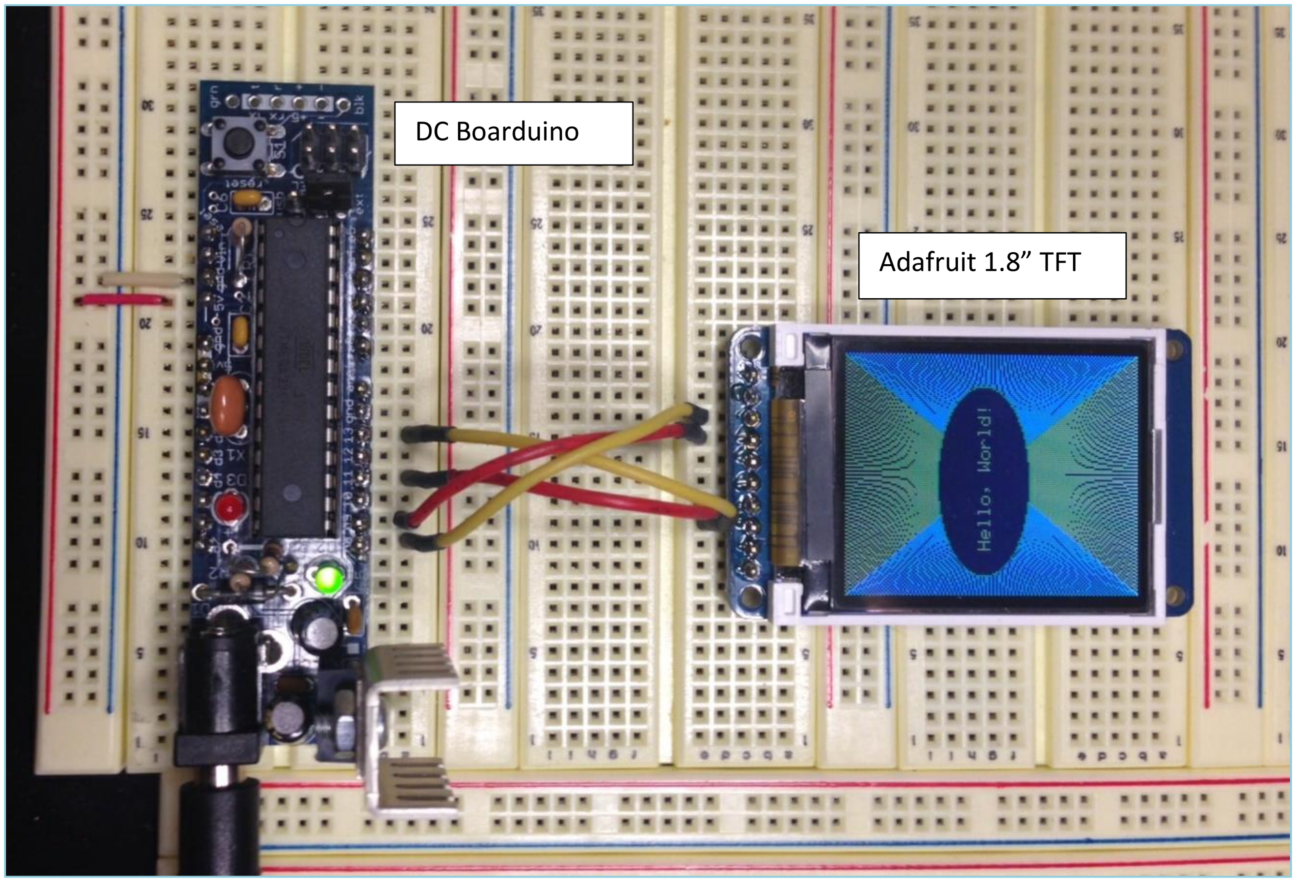
14) LANDSCAPE MODE

In the default portrait mode (0°), the display connectors are on the top and the long edges are vertical. In landscape mode (90°), the display connectors are on the left, and the long edges are horizontal. In fact, using the MADCTL command, it is equally easy to have upright text with the display connectors up, down, left or right. Additional "mirror image" displays are also possible, but not quite as useful.

Orientation	Mode Byte
0°	0x00
90°	0x60
180°	0xC0
270°	0xA0

```
void SetOrientation(int degrees)
{
    byte arg;
    switch (degrees)
    {
        case  90: arg = 0x60; break;
        case 180: arg = 0xC0; break;
        case 270: arg = 0xA0; break;
        default:  arg = 0x00; break;
    }
    WriteCmd(MADCTL);
    WriteByte(arg);
}
```

In landscape mode, the display width and height are switched. With 160 pixels per row, the number of characters per row increases from 21 to (160/8) = 26. The number of rows deceases from 20 to (128/8) = 16. The total number of characters per screen is 26x16 = 416.



15) MORE FUN

In the source code I include a small demo that runs some of the text and graphic routines. Would you like to do more with this display, like use a wider variety of colors, create larger fonts, adjust the display gamma, or display bitmap files? Check out the series of articles I wrote for the raspberry pi.

16) SOURCE CODE

```
//-----------------------------------------------------------------------------
//      TFT:  Experiments interfacing ATmega328 to an ST7735 1.8" LCD TFT display
//
//  Author   :  Bruce E. Hall <bhall66@gmail.com>
//  Website  :  w8bh.net
//  Version  :  1.0
//  Date     :  04 May 2014
//  Target   :  ATmega328P microcontroller
//  Language :  C, using AVR studio 6
//  Size     :  3622 bytes
//
//  Fuse settings:  8 MHz osc with 65 ms Delay, SPI enable; *NO* clock/8
//
//  Connections from LCD to DC Boarduino:
//
//  TFT pin 1  (backlight)      +5V
//  TFT pin 2  (MISO)           n/c
//  TFT pin 3  (SCK)            digital13, PB5(SCK)
//  TFT pin 4  (MOSI)           digital11, PB3(MOSI)
//  TFT pin 5  (TFT_Select)     gnd
//  TFT pin 7  (DC)             digital9,  PB1
//  TFT pin 8  (Reset)          digital8,  PB0
//  TFT pin 9  (Vcc)            +5V
//  TFT pin 10 (gnd)            gnd
//

//      ---------------------------------------------------------------------------
//      GLOBAL DEFINES

#define F_CPU       16000000L           // run CPU at 16 MHz
#define LED         5                   // Boarduino LED on PB5
#define ClearBit(x,y) x &= ~_BV(y)      // equivalent to cbi(x,y)
#define SetBit(x,y) x |= _BV(y)         // equivalent to sbi(x,y)

//      ---------------------------------------------------------------------------
//      INCLUDES

#include <avr/io.h>                     // deal with port registers
#include <avr/interrupt.h>              // deal with interrupt calls
#include <avr/pgmspace.h>               // put character data into progmem
#include <util/delay.h>                 // used for _delay_ms function
#include <string.h>                     // string manipulation routines
#include <avr/sleep.h>                  // used for sleep functions
#include <stdlib.h>

//      ---------------------------------------------------------------------------
//      TYPEDEFS

typedef uint8_t byte;                   // I just like byte & sbyte better
typedef int8_t sbyte;

//      ---------------------------------------------------------------------------
//      GLOBAL VARIABLES

const byte FONT_CHARS[96][5] PROGMEM =
{
```

```
{ 0x00, 0x00, 0x00, 0x00, 0x00 }, // (space)
{ 0x00, 0x00, 0x5F, 0x00, 0x00 }, // !
{ 0x00, 0x07, 0x00, 0x07, 0x00 }, // "
{ 0x14, 0x7F, 0x14, 0x7F, 0x14 }, // #
{ 0x24, 0x2A, 0x7F, 0x2A, 0x12 }, // $
{ 0x23, 0x13, 0x08, 0x64, 0x62 }, // %
{ 0x36, 0x49, 0x55, 0x22, 0x50 }, // &
{ 0x00, 0x05, 0x03, 0x00, 0x00 }, // '
{ 0x00, 0x1C, 0x22, 0x41, 0x00 }, // (
{ 0x00, 0x41, 0x22, 0x1C, 0x00 }, // )
{ 0x08, 0x2A, 0x1C, 0x2A, 0x08 }, // *
{ 0x08, 0x08, 0x3E, 0x08, 0x08 }, // +
{ 0x00, 0x50, 0x30, 0x00, 0x00 }, // ,
{ 0x08, 0x08, 0x08, 0x08, 0x08 }, // -
{ 0x00, 0x60, 0x60, 0x00, 0x00 }, // .
{ 0x20, 0x10, 0x08, 0x04, 0x02 }, // /
{ 0x3E, 0x51, 0x49, 0x45, 0x3E }, // 0
{ 0x00, 0x42, 0x7F, 0x40, 0x00 }, // 1
{ 0x42, 0x61, 0x51, 0x49, 0x46 }, // 2
{ 0x21, 0x41, 0x45, 0x4B, 0x31 }, // 3
{ 0x18, 0x14, 0x12, 0x7F, 0x10 }, // 4
{ 0x27, 0x45, 0x45, 0x45, 0x39 }, // 5
{ 0x3C, 0x4A, 0x49, 0x49, 0x30 }, // 6
{ 0x01, 0x71, 0x09, 0x05, 0x03 }, // 7
{ 0x36, 0x49, 0x49, 0x49, 0x36 }, // 8
{ 0x06, 0x49, 0x49, 0x29, 0x1E }, // 9
{ 0x00, 0x36, 0x36, 0x00, 0x00 }, // :
{ 0x00, 0x56, 0x36, 0x00, 0x00 }, // ;
{ 0x00, 0x08, 0x14, 0x22, 0x41 }, // <
{ 0x14, 0x14, 0x14, 0x14, 0x14 }, // =
{ 0x41, 0x22, 0x14, 0x08, 0x00 }, // >
{ 0x02, 0x01, 0x51, 0x09, 0x06 }, // ?
{ 0x32, 0x49, 0x79, 0x41, 0x3E }, // @
{ 0x7E, 0x11, 0x11, 0x11, 0x7E }, // A
{ 0x7F, 0x49, 0x49, 0x49, 0x36 }, // B
{ 0x3E, 0x41, 0x41, 0x41, 0x22 }, // C
{ 0x7F, 0x41, 0x41, 0x22, 0x1C }, // D
{ 0x7F, 0x49, 0x49, 0x49, 0x41 }, // E
{ 0x7F, 0x09, 0x09, 0x01, 0x01 }, // F
{ 0x3E, 0x41, 0x41, 0x51, 0x32 }, // G
{ 0x7F, 0x08, 0x08, 0x08, 0x7F }, // H
{ 0x00, 0x41, 0x7F, 0x41, 0x00 }, // I
{ 0x20, 0x40, 0x41, 0x3F, 0x01 }, // J
{ 0x7F, 0x08, 0x14, 0x22, 0x41 }, // K
{ 0x7F, 0x40, 0x40, 0x40, 0x40 }, // L
{ 0x7F, 0x02, 0x04, 0x02, 0x7F }, // M
{ 0x7F, 0x04, 0x08, 0x10, 0x7F }, // N
{ 0x3E, 0x41, 0x41, 0x41, 0x3E }, // O
{ 0x7F, 0x09, 0x09, 0x09, 0x06 }, // P
{ 0x3E, 0x41, 0x51, 0x21, 0x5E }, // Q
{ 0x7F, 0x09, 0x19, 0x29, 0x46 }, // R
{ 0x46, 0x49, 0x49, 0x49, 0x31 }, // S
{ 0x01, 0x01, 0x7F, 0x01, 0x01 }, // T
{ 0x3F, 0x40, 0x40, 0x40, 0x3F }, // U
{ 0x1F, 0x20, 0x40, 0x20, 0x1F }, // V
{ 0x7F, 0x20, 0x18, 0x20, 0x7F }, // W
{ 0x63, 0x14, 0x08, 0x14, 0x63 }, // X
{ 0x03, 0x04, 0x78, 0x04, 0x03 }, // Y
{ 0x61, 0x51, 0x49, 0x45, 0x43 }, // Z
{ 0x00, 0x00, 0x7F, 0x41, 0x41 }, // [
{ 0x02, 0x04, 0x08, 0x10, 0x20 }, // "\"
{ 0x41, 0x41, 0x7F, 0x00, 0x00 }, // ]
{ 0x04, 0x02, 0x01, 0x02, 0x04 }, // ^
{ 0x40, 0x40, 0x40, 0x40, 0x40 }, // _
{ 0x00, 0x01, 0x02, 0x04, 0x00 }, // `
{ 0x20, 0x54, 0x54, 0x54, 0x78 }, // a
{ 0x7F, 0x48, 0x44, 0x44, 0x38 }, // b
{ 0x38, 0x44, 0x44, 0x44, 0x20 }, // c
{ 0x38, 0x44, 0x44, 0x48, 0x7F }, // d
{ 0x38, 0x54, 0x54, 0x54, 0x18 }, // e
{ 0x08, 0x7E, 0x09, 0x01, 0x02 }, // f
```

```
    { 0x08, 0x14, 0x54, 0x54, 0x3C }, // g
    { 0x7F, 0x08, 0x04, 0x04, 0x78 }, // h
    { 0x00, 0x44, 0x7D, 0x40, 0x00 }, // i
    { 0x20, 0x40, 0x44, 0x3D, 0x00 }, // j
    { 0x00, 0x7F, 0x10, 0x28, 0x44 }, // k
    { 0x00, 0x41, 0x7F, 0x40, 0x00 }, // l
    { 0x7C, 0x04, 0x18, 0x04, 0x78 }, // m
    { 0x7C, 0x08, 0x04, 0x04, 0x78 }, // n
    { 0x38, 0x44, 0x44, 0x44, 0x38 }, // o
    { 0x7C, 0x14, 0x14, 0x14, 0x08 }, // p
    { 0x08, 0x14, 0x14, 0x18, 0x7C }, // q
    { 0x7C, 0x08, 0x04, 0x04, 0x08 }, // r
    { 0x48, 0x54, 0x54, 0x54, 0x20 }, // s
    { 0x04, 0x3F, 0x44, 0x40, 0x20 }, // t
    { 0x3C, 0x40, 0x40, 0x20, 0x7C }, // u
    { 0x1C, 0x20, 0x40, 0x20, 0x1C }, // v
    { 0x3C, 0x40, 0x30, 0x40, 0x3C }, // w
    { 0x44, 0x28, 0x10, 0x28, 0x44 }, // x
    { 0x0C, 0x50, 0x50, 0x50, 0x3C }, // y
    { 0x44, 0x64, 0x54, 0x4C, 0x44 }, // z
    { 0x00, 0x08, 0x36, 0x41, 0x00 }, // {
    { 0x00, 0x00, 0x7F, 0x00, 0x00 }, // |
    { 0x00, 0x41, 0x36, 0x08, 0x00 }, // }
    { 0x08, 0x08, 0x2A, 0x1C, 0x08 }, // ->
    { 0x08, 0x1C, 0x2A, 0x08, 0x08 }, // <-
};

//      ---------------------------------------------------------------------------
//      MISC ROUTINES

void SetupPorts()
{
    DDRB = 0x2F;                       // 0010.1111; set B0-B3, B5 as outputs
    DDRC = 0x00;                       // 0000.0000; set PORTC as inputs
    SetBit(PORTB,0);                   // start with TFT reset line inactive high
}

void msDelay(int delay)                // put into a routine
{                                      // to remove code inlining
    for (int i=0;i<delay;i++)          // at cost of timing accuracy
    _delay_ms(1);
}

void FlashLED(byte count)
// flash the on-board LED at ~ 3 Hz
{
    for (;count>0;count--)
    {
         SetBit(PORTB,LED);            // turn LED on
         msDelay(150);                 // wait
         ClearBit(PORTB,LED);          // turn LED off
         msDelay(150);                 // wait
    }
}

unsigned long intsqrt(unsigned long val)
// calculate integer value of square root
{
    unsigned long mulMask = 0x0008000;
    unsigned long retVal  = 0;
    if (val > 0)
    {
        while (mulMask != 0)
        {
            retVal |= mulMask;
            if ((retVal*retVal) > val)
                retVal &= ~ mulMask;
            mulMask >>= 1;
        }
    }
    return retVal;
```

```c
}

/*
char* itoa(int i, char b[]){
    char const digit[] = "0123456789";
    char* p = b;
    if(i<0){
        *p++ = '-';
        i *= -1;
    }
    int shifter = i;
    do{ //Move to where representation ends
        ++p;
        shifter = shifter/10;
    }while(shifter);
    *p = '\0';
    do{ //Move back, inserting digits as u go
        *--p = digit[i%10];
        i = i/10;
    }while(i);
    return b;
}
*/


//  ---------------------------------------------------------------------------
//  SPI ROUTINES
//
//          b7    b6    b5    b4    b3    b2    b1    b0
//  SPCR:   SPIE  SPE   DORD  MSTR  CPOL  CPHA  SPR1  SPR0
//          0     1     0     1  .  0     0     0     1
//
//  SPIE - enable SPI interrupt
//   SPE - enable SPI
//  DORD - 0=MSB first, 1=LSB first
//  MSTR - 0=slave, 1=master
//  CPOL - 0=clock starts low, 1=clock starts high
//  CPHA - 0=read on rising-edge, 1=read on falling-edge
//  SPRx - 00=osc/4, 01=osc/16, 10=osc/64, 11=osc/128
//
//  SPCR = 0x50:  SPI enabled as Master, mode 0, at 16/4 = 4 MHz

void OpenSPI()
{
    SPCR = 0x50;                        // SPI enabled as Master, Mode0 at 4 MHz
    SetBit(SPSR,SPI2X);                 // double the SPI rate: 4-->8 MHz
}

void CloseSPI()
{
    SPCR = 0x00;                        // clear SPI enable bit
}

byte Xfer(byte data)
{
    SPDR = data;                        // initiate transfer
    while (!(SPSR & 0x80));             // wait for transfer to complete
    return SPDR;
}


//  ---------------------------------------------------------------------------
//  ST7735 ROUTINES

#define SWRESET 0x01                    // software reset
#define SLPOUT  0x11                    // sleep out
#define DISPOFF 0x28                    // display off
#define DISPON  0x29                    // display on
#define CASET   0x2A                    // column address set
#define RASET   0x2B                    // row address set
#define RAMWR   0x2C                    // RAM write
```

```
#define MADCTL  0x36                   // axis control
#define COLMOD  0x3A                   // color mode


// 1.8" TFT display constants
#define XSIZE   128
#define YSIZE   160
#define XMAX    XSIZE-1
#define YMAX    YSIZE-1

// Color constants
#define BLACK   0x0000
#define BLUE    0x001F
#define RED     0xF800
#define GREEN   0x0400
#define LIME    0x07E0
#define CYAN    0x07FF
#define MAGENTA 0xF81F
#define YELLOW  0xFFE0
#define WHITE   0xFFFF


void WriteCmd (byte cmd)
{
    ClearBit(PORTB,1);                 // B1=DC; 0=command, 1=data
    Xfer(cmd);
    SetBit(PORTB,1);                   // return DC high
}

void WriteByte (byte b)
{
    Xfer(b);
}

void WriteWord (int w)
{
    Xfer(w >> 8);                      // write upper 8 bits
    Xfer(w & 0xFF);                    // write lower 8 bits
}

void Write888 (long data, int count)
{
    byte red = data>>16;               // red = upper 8 bits
    byte green = (data>>8) & 0xFF;     // green = middle 8 bits
    byte blue = data & 0xFF;           // blue = lower 8 bits
    for (;count>0;count--)
    {
        WriteByte(red);
        WriteByte(green);
        WriteByte(blue);
    }
}

void Write565 (int data, unsigned int count)
// send 16-bit pixel data to the controller
// note: inlined spi xfer for optimization
{
    WriteCmd(RAMWR);
    for (;count>0;count--)
    {
        SPDR = (data >> 8);            // write hi byte
        while (!(SPSR & 0x80));        // wait for transfer to complete
        SPDR = (data & 0xFF);          // write lo byte
        while (!(SPSR & 0x80));        // wait for transfer to complete
    }
}

void HardwareReset()
{
    ClearBit(PORTB,0);                 // pull PB0 (digital 8) low
    msDelay(1);                        // 1mS is enough
    SetBit(PORTB,0);                   // return PB0 high
    msDelay(150);                      // wait 150mS for reset to finish
}

void InitDisplay()
{
    HardwareReset();                   // initialize display controller
    WriteCmd(SLPOUT);                  // take display out of sleep mode
    msDelay(150);                      // wait 150mS for TFT driver circuits
    WriteCmd(COLMOD);                  // select color mode:
    WriteByte(0x05);                   // mode 5 = 16bit pixels (RGB565)
    WriteCmd(DISPON);                  // turn display on!
}

void SetAddrWindow(byte x0, byte y0, byte x1, byte y1)
{
    WriteCmd(CASET);                   // set column range (x0,x1)
    WriteWord(x0);
    WriteWord(x1);
    WriteCmd(RASET);                   // set row range (y0,y1)
    WriteWord(y0);
    WriteWord(y1);
}

void ClearScreen()
{
    SetAddrWindow(0,0,XMAX,YMAX);       // set window to entire display
    WriteCmd(RAMWR);
    for (unsigned int i=40960;i>0;--i)  // byte count = 128*160*2
    {
        SPDR = 0;                       // initiate transfer of 0x00
        while (!(SPSR & 0x80));         // wait for xfer to finish
    }
}


//  ---------------------------------------------------------------------------
//  SIMPLE GRAPHICS ROUTINES
//
//  note: many routines have byte parameters, to save space,
//  but these can easily be changed to int params for larger displays.

void DrawPixel (byte x, byte y, int color)
{
    SetAddrWindow(x,y,x,y);
    Write565(color,1);
}

void HLine (byte x0, byte x1, byte y, int color)
// draws a horizontal line in given color
{
    byte width = x1-x0+1;
    SetAddrWindow(x0,y,x1,y);
    Write565(color,width);
}

void VLine (byte x, byte y0, byte y1, int color)
// draws a vertical line in given color
{
    byte height = y1-y0+1;
    SetAddrWindow(x,y0,x,y1);
    Write565(color,height);
}

void Line (int x0, int y0, int x1, int y1, int color)
// an elegant implementation of the Bresenham algorithm
{
    int dx = abs(x1-x0), sx = x0<x1 ? 1 : -1;
    int dy = abs(y1-y0), sy = y0<y1 ? 1 : -1;
    int err = (dx>dy ? dx : -dy)/2, e2;
    for(;;)
    {
        DrawPixel(x0,y0,color);
        if (x0==x1 && y0==y1) break;
        e2 = err;
        if (e2 >-dx) { err -= dy; x0 += sx; }
        if (e2 < dy) { err += dx; y0 += sy; }
    }
}

void DrawRect (byte x0, byte y0, byte x1, byte y1, int color)
// draws a rectangle in given color
{
    HLine(x0,x1,y0,color);
    HLine(x0,x1,y1,color);
    VLine(x0,y0,y1,color);
    VLine(x1,y0,y1,color);
}

void FillRect (byte x0, byte y0, byte x1, byte y1, int color)
{
    byte width = x1-x0+1;
    byte height = y1-y0+1;
    SetAddrWindow(x0,y0,x1,y1);
    Write565(color,width*height);
}

void CircleQuadrant (byte xPos, byte yPos, byte radius, byte quad, int color)
// draws circle quadrant(s) centered at x,y with given radius & color
// quad is a bit-encoded representation of which cartesian quadrant(s) to draw.
// Remember that the y axis on our display is 'upside down':
// bit 0: draw quadrant I (lower right)
// bit 1: draw quadrant IV (upper right)
// bit 2: draw quadrant II (lower left)
// bit 3: draw quadrant III (upper left)
{
    int x, xEnd = (707*radius)/1000 + 1;
    for (x=0; x<xEnd; x++)
    {
        byte y = intsqrt(radius*radius - x*x);
        if (quad & 0x01)
        {
            DrawPixel(xPos+x,yPos+y,color);      // lower right
            DrawPixel(xPos+y,yPos+x,color);
        }
        if (quad & 0x02)
        {
            DrawPixel(xPos+x,yPos-y,color);      // upper right
            DrawPixel(xPos+y,yPos-x,color);
        }
        if (quad & 0x04)
        {
            DrawPixel(xPos-x,yPos+y,color);      // lower left
            DrawPixel(xPos-y,yPos+x,color);
        }
        if (quad & 0x08)
        {
            DrawPixel(xPos-x,yPos-y,color);      // upper left
            DrawPixel(xPos-y,yPos-x,color);
        }
    }
}

void Circle (byte xPos, byte yPos, byte radius, int color)
// draws circle at x,y with given radius & color
{
    CircleQuadrant(xPos,yPos,radius,0x0F,color); // do all 4 quadrants
}

void RoundRect (byte x0, byte y0, byte x1, byte y1, byte r, int color)
// draws a rounded rectangle with corner radius r.
// coordinates: top left = x0,y0; bottom right = x1,y1
```

```
{
    HLine(x0+r,x1-r,y0,color);                       // top side
    HLine(x0+r,x1-r,y1,color);                       // bottom side
    VLine(x0,y0+r,y1-r,color);                       // left side
    VLine(x1,y0+r,y1-r,color);                       // right side
    CircleQuadrant(x0+r,y0+r,r,8,color);             // upper left corner
    CircleQuadrant(x1-r,y0+r,r,2,color);             // upper right corner
    CircleQuadrant(x0+r,y1-r,r,4,color);             // lower left corner
    CircleQuadrant(x1-r,y1-r,r,1,color);             // lower right corner
}

void FillCircle (byte xPos, byte yPos, byte radius, int color)
// draws filled circle at x,y with given radius & color
{
    long r2 = radius * radius;
    for (int x=0; x<=radius; x++)
    {
        byte y = intsqrt(r2-x*x);
        byte y0 = yPos-y;
        byte y1 = yPos+y;
        VLine(xPos+x,y0,y1,color);
        VLine(xPos-x,y0,y1,color);
    }
}

void Ellipse (int x0, int y0, int width, int height, int color)
// draws an ellipse of given width & height
// two-part Bresenham method
// note: slight discontinuity between parts on some (narrow) ellipses.
{
    int a=width/2, b=height/2;
    int x = 0, y = b;
    long a2 = (long)a*a*2;
    long b2 = (long)b*b*2;
    long error = (long)a*a*b;
    long stopY=0, stopX = a2*b;
    while (stopY <= stopX)
    {
        DrawPixel(x0+x,y0+y,color);
        DrawPixel(x0+x,y0-y,color);
        DrawPixel(x0-x,y0+y,color);
        DrawPixel(x0-x,y0-y,color);
        x++;
        error -= b2*(x-1);
        stopY += b2;
        if (error < 0)
        {
            error += a2*(y-1);
            y--;
            stopX -= a2;
        }
    }
    x=a; y=0; error = b*b*a;
    stopY = a*b2; stopX = 0;
    while (stopY >= stopX)
    {
        DrawPixel(x0+x,y0+y,color);
        DrawPixel(x0+x,y0-y,color);
        DrawPixel(x0-x,y0+y,color);
        DrawPixel(x0-x,y0-y,color);
        y++;
        error -= a2*(y-1);
        stopX += a2;
        if (error < 0)
        {
            error += b2*(x-1);
            x--;
            stopY -= b2;
        }
    }
}
```

```
void FillEllipse(int xPos,int yPos,int width,int height, int color)
// draws a filled ellipse of given width & height
{
    int a=width/2, b=height/2;         // get x & y radii
    int x1, x0 = a, y = 1, dx = 0;
    long a2 = a*a, b2 = b*b;           // need longs: big numbers!
    long a2b2 = a2 * b2;
    HLine(xPos-a,xPos+a,yPos,color);   // draw centerline
    while (y<=b)                       // draw horizontal lines...
    {
        for (x1= x0-(dx-1); x1>0; x1--)
        if (b2*x1*x1 + a2*y*y <= a2b2)
        break;
        dx = x0-x1;
        x0 = x1;
        HLine(xPos-x0,xPos+x0,yPos+y,color);
        HLine(xPos-x0,xPos+x0,yPos-y,color);
        y += 1;
    }
}


//  ---------------------------------------------------------------------------
//  TEXT ROUTINES
//
//  Each ASCII character is 5x7, with one pixel space between characters
//  So, character width = 6 pixels & character height = 8 pixels.
//
//  In portrait mode:
//    Display width = 128 pixels, so there are 21 chars/row (21x6 = 126).
//    Display height = 160 pixels, so there are 20 rows (20x8 = 160).
//    Total number of characters in portait mode = 21 x 20 = 420.
//
//  In landscape mode:
//    Display width is 160, so there are 26 chars/row (26x6 = 156).
//    Display height is 128, so there are 16 rows (16x8 = 128).
//    Total number of characters in landscape mode = 26x16 = 416.

byte curX,curY;                        // current x & y cursor position

void GotoXY (byte x,byte y)
// position cursor on character x,y grid, where 0<x<20, 0<y<19.
{
    curX = x;
    curY = y;
}

void GotoLine(byte y)
// position character cursor to start of line y, where 0<y<19.
{
    curX = 0;
    curY = y;
}

void AdvanceCursor()
// moves character cursor to next position, assuming portrait orientation
{
    curX++;                            // advance x position
    if (curX>20)                       // beyond right margin?
    {
        curY++;                        // go to next line
        curX = 0;                      // at left margin
    }
    if (curY>19)                       // beyond bottom margin?
        curY = 0;                      // start at top again
}

void SetOrientation(int degrees)
// Set the display orientation to 0,90,180,or 270 degrees
{
    byte arg;
    switch (degrees)
    {
        case  90: arg = 0x60; break;
        case 180: arg = 0xC0; break;
        case 270: arg = 0xA0; break;
        default:  arg = 0x00; break;
    }
    WriteCmd(MADCTL);
    WriteByte(arg);
}

void PutCh (char ch, byte x, byte y, int color)
// write ch to display X,Y coordinates using ASCII 5x7 font
{
    int  pixel;
    byte row, col, bit, data, mask = 0x01;
    SetAddrWindow(x,y,x+4,y+6);
    WriteCmd(RAMWR);
    for (row=0; row<7;row++)
    {
        for (col=0; col<5;col++)
        {
            data = pgm_read_byte(&(FONT_CHARS[ch-32][col]));
            bit = data & mask;
            if (bit==0) pixel=BLACK;
            else        pixel=color;
            WriteWord(pixel);
        }
        mask <<= 1;
    }
}

void WriteChar(char ch, int color)
// writes character to display at current cursor position.
{
    PutCh(ch,curX*6, curY*8, color);
    AdvanceCursor();
}

void WriteString(char *text, int color)
// writes string to display at current cursor position.
{
     for (;*text;text++)               // for all non-nul chars
        WriteChar(*text,color);        // write the char
}

void WriteInt(int i)
// writes integer i at current cursor position
{
    char str[8];                       // buffer for string result
    itoa(i,str,10);                    // convert to string, base 10
    WriteString(str,WHITE);
}

void WriteHex(int i)
// writes hexadecimal value of integer i at current cursor position
{
    char str[8];                       // buffer for string result
    itoa(i,str,16);                    // convert to base 16 (hex)
    WriteString(str,WHITE);
}


//  ---------------------------------------------------------------------------
//  TEST ROUTINES

void PixelTest()
// draws 4000 pixels on the screen
{
    for (int i=4000; i>0; i--)         // do a whole bunch:
```

```
    {
        int x = rand() % XMAX;          // random x coordinate
        int y = rand() % YMAX;          // random y coordinate
        DrawPixel(x,y,YELLOW);          // draw pixel at x,y
    }
}

void LineTest()
// sweeps Line routine through all four quadrants.
{
    ClearScreen();
    int x,y,x0=64,y0=80;
    for (x=0;x<XMAX;x+=2) Line(x0,y0,x,0,YELLOW);
    for (y=0;y<YMAX;y+=2) Line(x0,y0,XMAX,y,CYAN);
    for (x=XMAX;x>0;x-=2) Line(x0,y0,x,YMAX,YELLOW);
    for (y=YMAX;y>0;y-=2) Line(x0,y0,0,y,CYAN);
    msDelay(2000);
}

void CircleTest()
// draw series of concentric circles
{
    for(int radius=6;radius<60;radius+=2)
        Circle(60,80,radius,YELLOW);
}

void PortraitChars()
// Writes 420 characters (5x7) to screen in portrait mode
{
    ClearScreen();
    for (int i=420;i>0;i--)
    {
        byte x= i % 21;
        byte y= i / 21;
        char ascii = (i % 96)+32;
        PutCh(ascii,x*6,y*8,CYAN);
    }
    msDelay(2000);
}


//  ---------------------------------------------------------------------------
//  MAIN PROGRAM

int main()
{
    SetupPorts();                       // use PortB for LCD interface
    FlashLED(1);                        // indicate program start
    OpenSPI();                          // start communication to TFT
    InitDisplay();                      // initialize TFT controller
    PortraitChars();                    // show full screen of ASCII chars
    LineTest();                         // paint background of lines
    FillEllipse(60,75,100,50,BLACK);    // erase an oval in center
    Ellipse(60,75,100,50,LIME);         // outline the oval in green
    char *str = "Hello, World!";        // text to display
    GotoXY(4,9);                        // position text cursor
    WriteString(str,YELLOW);            // display text inside oval
    CloseSPI();                         // close communication with TFT
    FlashLED(3);                        // indicate program end
}
```