Embedded Systems Interfacing

Lecture one

Digital Input Output Part 1

TFT (Thin Film Transistor)

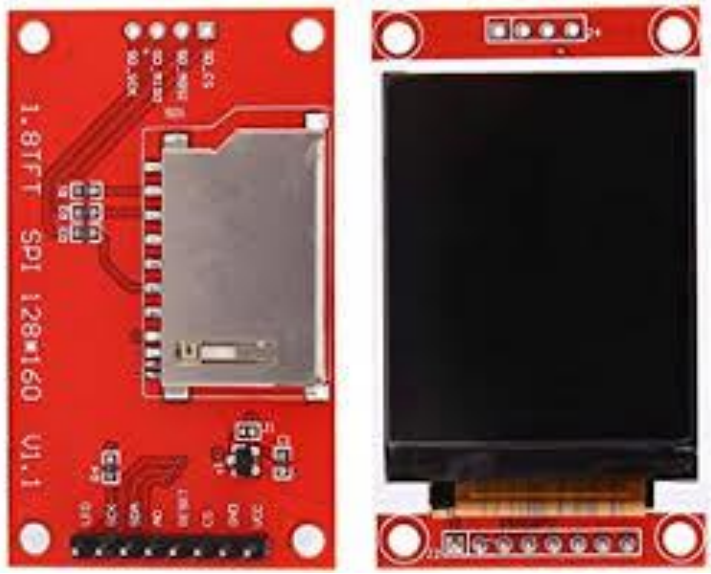
This TFT module is a **128x160** LCD matrix, controlled by the onboard Sitronix **ST7735** controller.

Thin film transistor liquid crystal display **(TFT-LCD)** is a variant of liquid crystal display (LCD) which uses thin-film transistor **(TFT)** technology to improve image quality.

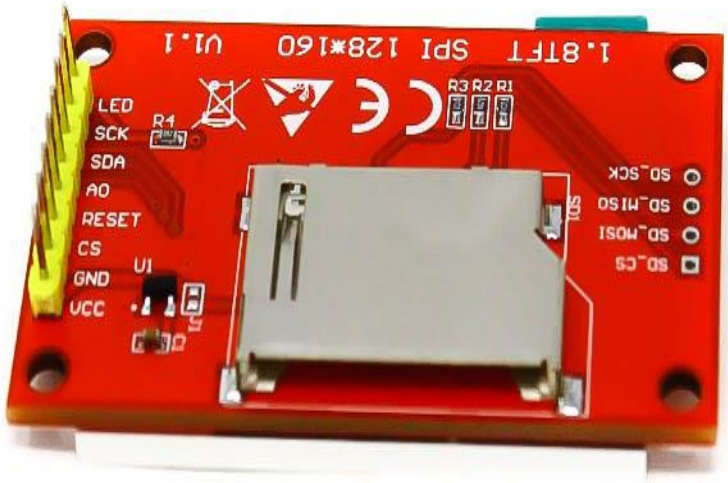
One of the most powerful ways of presenting data and interacting with users is through **color displays**.

You send data to it serially using the Serial-Peripheral Interface **(SPI)** protocol. Unfortunately, it's not as simple as writing to a character mode display.

The **ST7735S** is a single-chip controller/driver for 262K-color, graphic type TFT-LCD.



TFT (Pin Description)



Micro Controller	1.8”TFT”
5V	VCC
GND	GND
GND	CS
GPIO pin	RESET
GPIO pin	A0(D/C)
MOSI pin	SDA
SCK pin	SCK
3.3V	LED

SPI (Interfacing)

We chose the SPI protocol, since any data transfer to the TFT module would require this.

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 (SPI_CR1) to enable SPI communication.

This is an 16-bit register that contains the following bits:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	*LSB FIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1

The Hex Value of **SPI_CR1** = **0x0047**

SPI (Interfacing)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	*LSB FIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1

Bit0 & Bit1 → **CPHA , CPOL** :- Set the clock polarity and clock phase of the SPI to determine the transfer mode.

Bit2 → **MSTR** :- determines if the micro acts as a master (1) or slave (0) device.

Bit3 & Bit4 & Bit5 → **BR[2:0]**:-determine the transfer speed, as a fraction of the microcontroller's oscillator. When all are 0, the SPI transfer speed is OSC/2, which on my 8MHz micro is 8/2 = 4 MHz

Bit6 → **SPE** :- The SPE bit enables SPI

Bit7 → **LSBFIRST** :- determines the data direction: when 0, the most-significant bit is sent & received first.

Bit9 → **SSM** :- the NSS pin managed by hardware.

SPI(Interfacing)

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 SPI_DR . Load this register with a value, and the data transfer will start automatically.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

A bit7 in SPI_SR, 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, SPI_DR will hold the received data:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								BSY	OVR	MODF	CRC ERR	UDR	CHSID E	TXE	RXNE
								r	r	r	rc_w0	r	r	r	r

SPI(Interfacing)

```
void SPI1_voidInit(void)
{
    /* NSS Managed by HW..
    SPI1->CR1 = 0x0047;

}

u8    SPI1_u8SendDataU8(u8 Copy_u8Data)
{

    /* Send Data */
    SPI1->DR = Copy_u8Data;

    /* wait until finish */
    while ( (GET_BIT(SPI1->SR, 7)) == 1 );

    return SPI1->DR;

}
```

TFT(Interfacing)

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 GPIO microcontroller pin, to supply this information.

Here are the two routines:

```
static void WriteData    (u8 data)
{
     /* DC is 1 for data */
     DIO_SetPinVal(TFT_DC_PORT,TFT_DC_PIN,1);

     /* Send Data */
     SPI1_u8SendDataU8(data);
}

static void WriteCommand(u8 cmd)
{
     /* DC is 0 for data */
     DIO_SetPinVal(TFT_DC_PORT,TFT_DC_PIN,0);

     /* Send Command */
     SPI1_u8SendDataU8(cmd);
}
```

Initializing The Display

You should initialize the display before sending pixel data.

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.

To do a hardware reset, take the reset line briefly low, and wait enough time for the reset to complete:

```
/* Reset Puls To Initializing All registers*/
DIO_SetPinVal(TFT_RST_PORT,TFT_RST_PIN,0);
STK_BusyDelay(1000);
DIO_SetPinVal(TFT_RST_PORT,TFT_RST_PIN,1);
STK_BusyDelay(150000);
```

Initializing The Display

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.

```
WriteCommand(SLPOUT) ; // take display out of sleep mode
STK_BusyDelay(150000); // wait 150mS for TFT driver circuits
WriteCommand(COLMOD) ; // select color mode:
WriteData(0x05)      ; // mode 5 = 16bit pixels (RGB565)
WriteCommand(DISPON) ; // turn display on!
```

Initializing The Display

```
void TFT_voidInit      (void)
{
    /* Reset Puls To Initializing All registers*/
    DIO_SetPinVal(TFT_RST_PORT,TFT_RST_PIN,0);
    STK_BusyDelay(1000);
    DIO_SetPinVal(TFT_RST_PORT,TFT_RST_PIN,1);
    STK_BusyDelay(150000);


    WriteCommand(SLPOUT) ; // take display out of sleep mode
    STK_BusyDelay(150000); // wait 150mS for TFT driver circuits
    WriteCommand(COLMOD) ; // select color mode:
    WriteData(0x05)      ; // mode 5 = 16bit pixels (RGB565)
    WriteCommand(DISPON) ; // turn display on!
}
```

Display Color Mode

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 color space 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	-	-

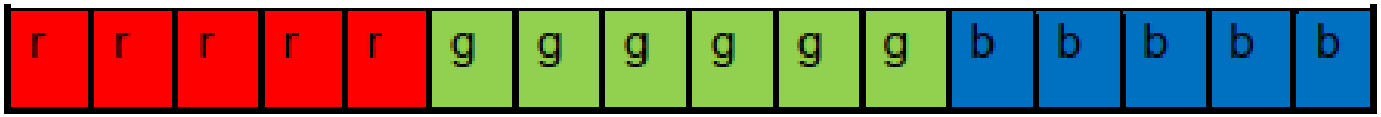
The lowest two bits of each red, green, and blue byte are ignored; only the 6 upper bits of each byte are used. That's a bit wasteful, isn't it? It takes time to send those empty bits.

Display Color Mode

The TFT controller supports three different color depths.

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:



Draw Pixels

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.

```
WriteCommand(CASET); // set column range (x0,x1)

WriteData( Xaxis );
WriteData( Xaxis );

WriteCommand(RASET); // set row range (y0,y1)

WriteData( Yaxis );
WriteData( Yaxis );

WriteCommand(RAMWR); // memory write
```

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.

Draw Pixels

Sending pixel data is as simple as sending both bytes of our 16-bit color, MSB first.

```
WriteData ( Copy_u16Data >> 8   ); // write hi byte
WriteData ( Copy_u16Data & 0xFF ); // write lo byte
```

The final Draw Pixel Function :-

```
void TFT_voidDrawPixels( u8 Xaxis , u8 Yaxis , u16 Copy_u16Data ){

    WriteCommand(CASET); // set column range (x0,x1)

    WriteData( Xaxis );
    WriteData( Xaxis );

    WriteCommand(RASET); // set row range (y0,y1)

    WriteData( Yaxis );
    WriteData( Yaxis );

    WriteCommand(RAMWR); // memory write

    WriteData ( Copy_u16Data >> 8   ); // write hi byte
    WriteData ( Copy_u16Data & 0xFF ); // write lo byte

}
```

The End …

IMT SCHOOL

www.imtschool.com

ww.facebook.com/imaketechnologyschool/