GUIDE: Controlling the 2.2" 18-bit color TFT LCD display breakout board by Adafruit with the BeagleBone Green/Black.

By Ben Smus, Art Yang, Rithik Ramamurthy, and Connor Goudie-Groat

The goal of this guide is to write a C program for BeagleBone that can draw on the LCD display breakout board (henceforth known as "breakout").

Vocab

- Breakout board: Breadboard-ready hardware that makes an electrical component easier to use.

- TFT: A variant of a liquid-crystal display that uses thin-film-transistor technology (TFT). We don't need to know anything about TFT in order to control the board.

Background

This breakout makes the TM022HDH26 component by Tianma Microelectronics easier to use. The TM022HDH26 has the ILI9340C (henceforth known as "driver chip") inside. The driver chip has a certain command set that it uses to draw (i.e. output signals that the TM022HDH26's LCD display can understand).

The role of our C program is to send commands to the driver chip. To do this, we have to use a slightly modified version of Serial Peripheral Interface (SPI). SPI and I2C are both serial protocols which allow one controller (a.k.a master) chip to communicate with multiple target (a.k.a slave) chips, which is a bus architecture.

This guide will reference the driver chip's datasheet, which can be found here: https://cdn-shop.adafruit.com/datasheets/ILI9340.pdf

The Arduino tutorial for the breakout board is sometimes applicable, and can be found here: https://learn.adafruit.com/2-2-tft-display/

Links to useful resources can be found on the Adafruit product page for the breakout: https://www.adafruit.com/product/1480

Please look at the SPI guide for BeagleBone before proceeding: (NOT OUR WORK, written by Ian Cruikshank and his teammates) https://opencoursehub.cs.sfu.ca/bfraser/grav-cms/cmpt433/links/files/2022-student-howtos-ensc351/SPI-On-BBG.pdf

Wiring up the breakout to the BeagleBone

Let's split the pins on the breakout into three categories:

1. Power: VIN (voltage in), GND (ground).
2. SPI pins: CS (chip select), SCLK (serial clock), MOSI (master-out, slave-in), MISO (master-in, slave-out).
3. Other: RST (reset), D/C (data/command), SDCS (SD card chip select), BL (backlight PWM).

We can ignore SDCS since we will not be using SD card related functionality.

We can ignore BL. When nothing is connected to the BL pin, it is pulled high, which sets the backlight to max brightness and looks great. You can send a PWM signal at any frequency to make the backlight dimmer. Connecting BL to ground will turn off the backlight.

We will need to wire up all the pins besides SDCS and BL to get the display to work.

VIN can be wired to 3.3V or 5V. GND should connect to one of the BeagleBone's ground pins.

Wire up the SPI pins on the breakout to the correct pins on the BeagleBone as described by the SPI guide, except for the CS pin. Remember to configure those pins to SPI mode with the `config-pin` command. We will need to manually control the CS pin. Connect the CS, D/C, and RST pins on the breakout to any available GPIO pins on the BeagleBone.

Communicating with the driver chip

The driver chip supports three types of communication:

1. Sending write commands to the chip. Write commands are a type of command where a command is sent to the chip and no response is sent back from the chip.
2. Sending read commands to the chip. Read commands are a type of command where a command is sent to the chip and the chip sends back a response.
3. Sending pixel stream data to the chip. This is done after a "memory write" (a write command) command is sent to the chip.

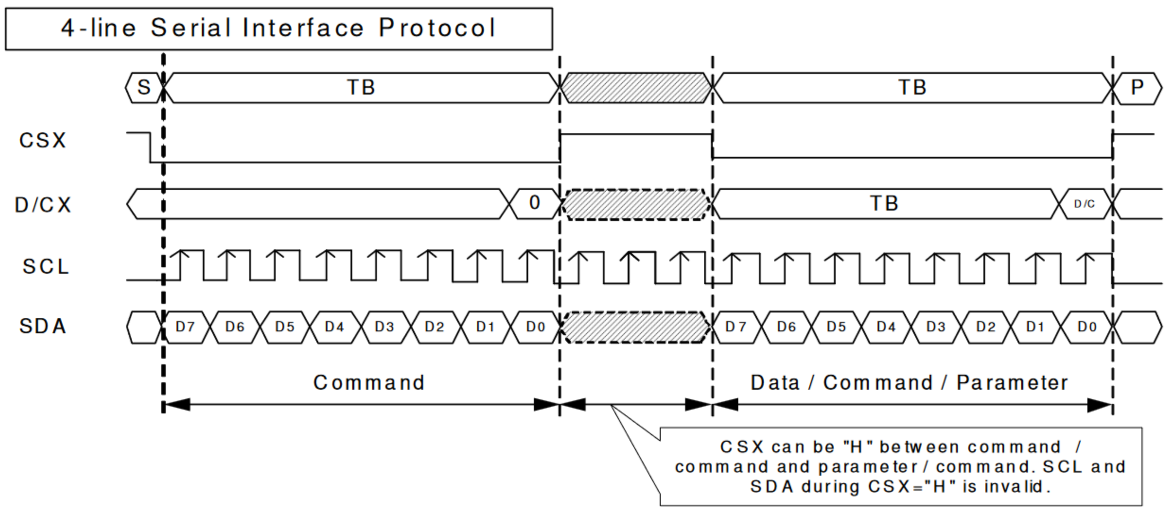
Communication is done by sending signals to 5 of the driver chips' pins. These pins are visible on the breakout, but are labeled differently than in the driver chip datasheet:

Breakout labeling	**Driver chip labeling**
CS	CSX
D/C	D/CX
SCLK	SCL
MOSI	SDA/SDI
MISO	SDO

So, for example, when you see SCL in the driver chip datasheet that means SCLK on the breakout.

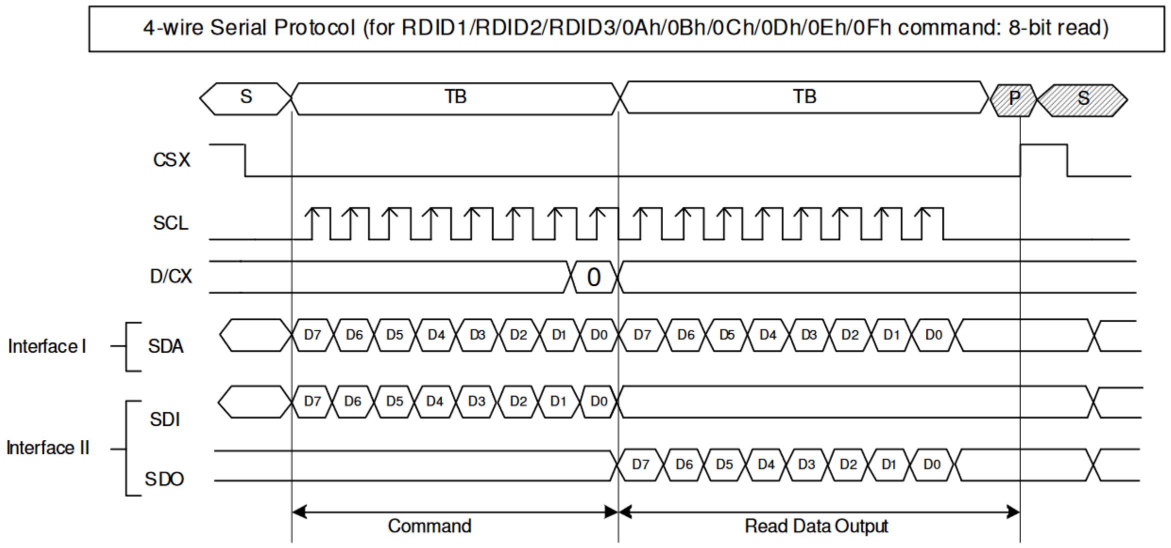
Here are the timing diagrams for write commands and 8-bit read commands. These two types of commands are sufficient for our purposes (the driver chip also has 24-bit and 32-bit read commands). A timing diagram shows what values the signals have to be on the y-axis, and time on the x-axis.

"Write" command from page 35 of the driver chip datasheet:


4-line Serial Interface Protocol
S
TB
TB
P
CSX
D/CX
0
TB
D/C
SCL
SDA
D7 D6 D5 D4 D3 D2 D1 D0
D7 D6 D5 D4 D3 D2 D1 D0
Command
Data / Command / Parameter
CSX can be "H" between command / command and parameter / command. SCL and SDA during CSX="H" is invalid.

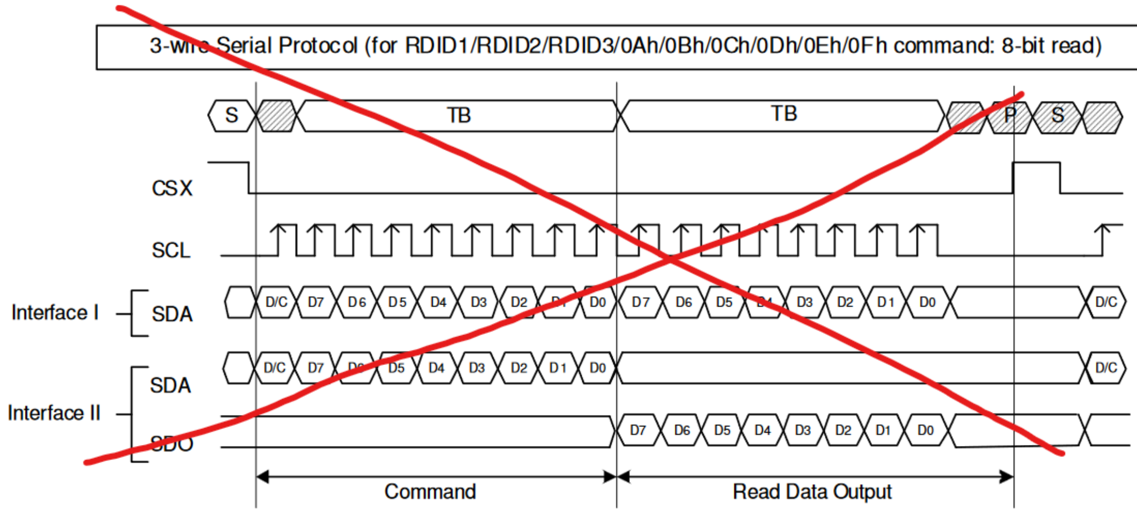
The timing diagram shows the command (all commands are one byte) being sent first, with D/C being set to LOW. Then, if command parameters are sent, D/C must be set to high.

8-bit "read" command from page 38 of the driver chip datasheet:



The timing diagram shows CSX being low during the entire communication sequence. After the command is sent, D/CX switches from low to high. Notice interface I and interface II: the breakout uses interface II, with SDI being exposed on MOSI and SDO being exposed on MISO.

Note: The driver chip also supports 3-wire protocol, which doesn't use the D/CX pin, however the breakout uses 4-line protocol. This means that you can ignore all of the 3-wire diagrams in the datasheet, such as this one on page 36 (notice the absence of the D/CX pin):



3-wire Serial Protocol (for RDID1/RDID2/RDID3/0Ah/0Bh/0Ch/0Dh/0Eh/0Fh command: 8-bit read)
S
TB
TB
P
S
CSX
SCL
Interface I
SDA
D/C D7 D6 D5 D4 D3 D2 D1 D0 D7 D6 D5 D4 D3 D2 D1 D0 D/C
Interface II
SDA
D/C D7 D6 D5 D4 D3 D2 D1 D0 D/C
SDO
D7 D6 D5 D4 D3 D2 D1 D0
Command
Read Data Output

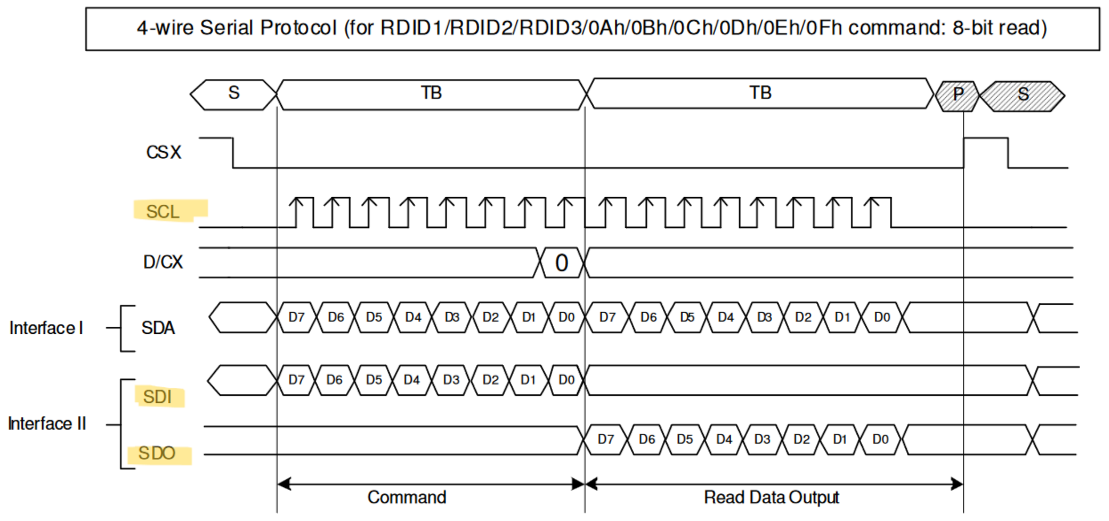
The 4-line serial interface protocol can be seen as an extension of SPI. It has the same clock, chip select, MOSI, and MISO signals, but adds the D/C signal to indicate how the MOSI signal should be interpreted.

Recall the SPI_transfer function from the SPI guide:

```
static void SPI_transfer(uint8_t *sendBuf, uint8_t *receiveBuf, int length) {
    struct spi_ioc_transfer transfer = {
        .tx_buf = (unsigned long) sendBuf,
        .rx_buf = (unsigned long) receiveBuf,
        .len = length
    };

    const int NUM_TRANSFERS = 1;
    int status = ioctl(spiFileDesc, SPI_IOC_MESSAGE(NUM_TRANSFERS), &transfer);
    if (status < 0) {
        perror("Error: SPI Transfer failed");
    }
}
```

We will be using the SPI_transfer function to control the highlighted signals:


4-wire Serial Protocol (for RDID1/RDID2/RDID3/0Ah/0Bh/0Ch/0Dh/0Eh/0Fh command: 8-bit read)
S
TB
TB
P
S
CSX
SCL
D/CX
0
Interface I
SDA
D7 D6 D5 D4 D3 D2 D1 D0 D7 D6 D5 D4 D3 D2 D1 D0
Interface II
SDI
D7 D6 D5 D4 D3 D2 D1 D0
SDO
D7 D6 D5 D4 D3 D2 D1 D0
Command
Read Data Output

The SPI_transfer function allows us to not worry about generating a clock pulse and synchronizing SDI/SDO with the clock. If we were to control CSX through the SPI_transfer function as well, we would not be able to change D/CX while CSX stays low, which is why we need to control CSX through GPIO.

For all of the GPIO signals (D/CX, CS, and RST), have a GPIO initialization function which sets them high. Any functions in your code that set the GPIO signals need to set them back to their initial state by the time that they exit.

Let's write the partial C code for two functions, Display_writeCommand and Display_read8Command. Comments with exclamation points need to be implemented.

```c
// according to 4-line serial interface timing diagram on page 35 of the datasheet
void Display_writeCommand(uint8_t command, uint8_t *params, size_t param_count) {
    // ! set the CS to low through GPIO
    // ! set the D/C to low through GPIO
    SPI_transfer(&command, NULL, 1);
    // ! set the D/C to high through GPIO
    SPI_transfer(params, NULL, param_count);
    // ! set the CS to high through GPIO
}
```

```c
// according to the 4-line serial interface "8-bit read" timing diagram on page 38
of the datasheet
uint8_t Display_read8Command(uint8_t command) {
    // ! set the CS to low through GPIO
    // ! set the D/C to low through GPIO
    SPI_transfer(&command, NULL, 1);
    // ! set the D/C to high through GPIO
    uint8_t readVal;
    SPI_transfer(NULL, &readVal, 1);
    // ! set the CS to high through GPIO
    return readVal;
}
```

A table of driver chip commands can be found on page 83 of the driver chip datasheet. All of the read commands start with the word "read", for all the other commands you can use Display_writeCommand to send them.
The hex command code for every command is highlighted on the table, and is written like "<number1><number2>h", which would be "0x<number1><number2>" in C.

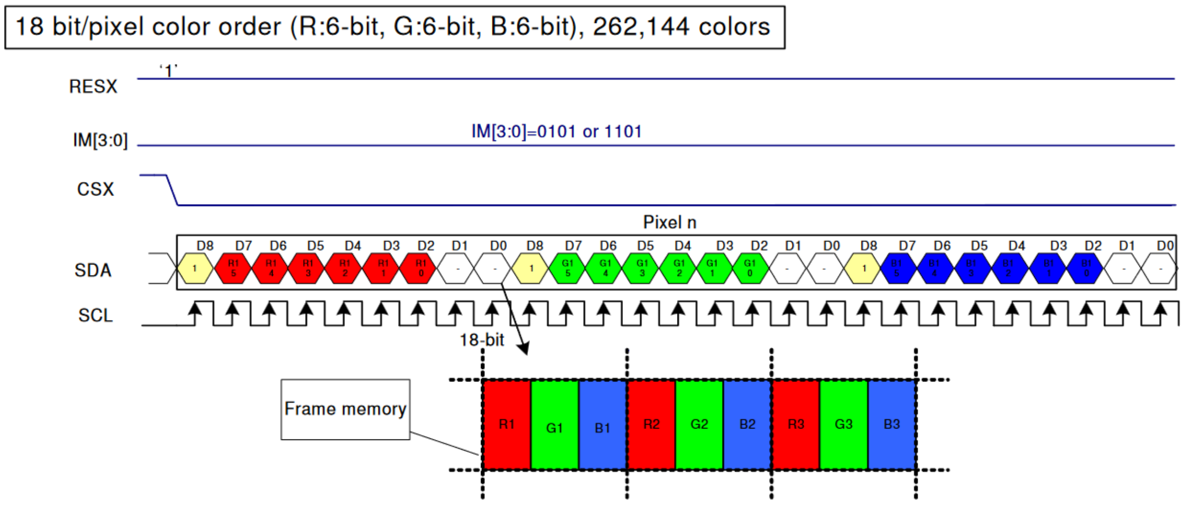
As part of the display initialization, and to ensure that you start from a 'blank slate' when running your program, you should implement a hardware reset function (finally using the RST GPIO signal!):

```c
static void hardwareReset() {
    // ! wait for 100 ms
    // ! set RST to low
    // ! wait for 10 ms
    // ! set RST to high
    // ! wait for 20 ms
}
```

Now that we have the nitty-gritty of the actual signals that we have to send figured out, here is what you need to do to actually draw something on the display.

1. Hardware reset.
2. Turn off sleep mode by sending the "sleep OUT" command and then waiting 60 ms for the display to wake up.
3. Sending the "display ON" command.
4. Make sure that everything is working as expected, by seeing that the display sends back the correct power state. Send the "read power mode" command using Display_read8Command: we are expecting it to return 0x9C, which means that the display is not asleep and some other stuff.

5. Specify where you want to draw on the display. Use the "column address set" and "page address set" commands. "column address set" specifies the minimum and maximum columns, and "page address set" specifies the minimum and maximum rows. This creates a rectangular area inside the display that will be where the pixels that you send get drawn.
6. Send the "memory write" command.
7. Send the pixel bytes. The format is shown in the image below. You should make a function for this that calls SPI_transfer multiple times. At the start of the function, it should set CSX to low, and set CSX to high when it is done with all the SPI_transfers.


18 bit/pixel color order (R:6-bit, G:6-bit, B:6-bit), 262,144 colors
RESX
'1'
IM[3:0]
IM[3:0]=0101 or 1101
CSX
Pixel n
SDA
SCL
D8 D7 D6 D5 D4 D3 D2 D1 D0 D8 D7 D6 D5 D4 D3 D2 D1 D0 D8 D7 D6 D5 D4 D3 D2 D1 D0
18-bit
Frame memory
R1 G1 B1 R2 G2 B2 R3 G3 B3

8. Send the "noop" command. This means that you are done sending pixel data.
9. Repeat steps 5-8 for as long as desired!