(https://vivonomicon.com/)

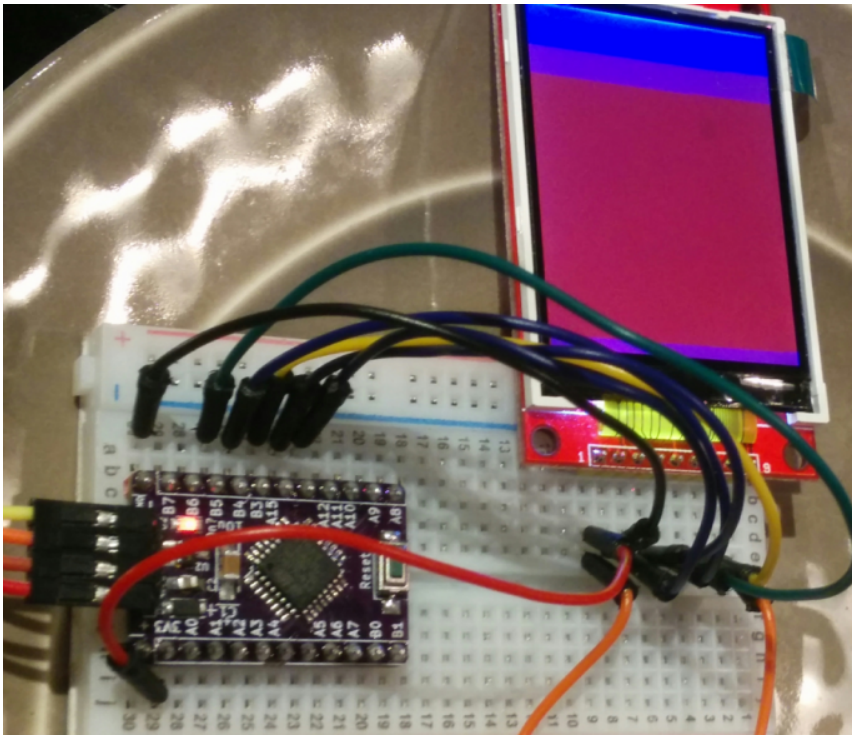
Blog for my various projects, experiments, and learnings

JUNE 17, 2018 (HTTPS://VIVONOMICON.COM/2018/06/17/DRAWING-TO-A-SMALL-TFT-DISPLAY-THE-ILI9341-AND-STM32/)

Drawing to a Small TFT Display: the ILI9341 and STM32

STM32 Baremetal Examples (https://vivonomicon.com/category/stm32_baremetal_examples/), Talking to Hardware (https://vivonomicon.com/category/talking_to_hardware/)

As you learn about more of your microcontroller's peripherals and start to work with more types of sensors and actuators, you will probably want to add small displays to your projects. Previously, I wrote about creating a simple program to draw data to an SSD1331 OLED display (https://vivonomicon.com/2018/04/25/stm32-software-spi-ssd1331-sketch/), but while they look great, the small size and low resolution can be limiting. Fortunately, the larger (and slightly cheaper) ILI9341 TFT (https://en.wikipedia.org/wiki/Thin-film-transistor_liquid-crystal_display) display module uses a nearly-identical SPI communication protocol, so this tutorial will build on that previous post by going over how to draw to a 2.2″ ILI9341 module (https://www.adafruit.com/product/1480) using the STM32's hardware SPI peripheral.



An ILI9341 display being driven by an STM32F0 chip. Technically this isn't a 'Nucleo' board, but the code is the same.

We'll cover the basic steps of setting up the required GPIO pins, initializing the SPI peripheral, starting the display, and then finally drawing pixel colors to it. This tutorial won't read any data from the display, so we can use the hardware peripheral's MISO pin for other purposes and leave the TFT's MISO pin disconnected. And as with my previous STM32 posts, example code will be provided (https://github.com/WRansohoff/STM32_ILI9341_HWSPI) for both the STM32F031K6 (https://www.digikey.com/product-detail/en/stmicroelectronics/NUCLEO-F031K6/497-15979-ND/5428803) and STM32L031K6 (https://www.digikey.com/product-detail/en/stmicroelectronics/NUCLEO-L031K6/497-16283-ND/5806780) 'Nucleo' boards.

Step 0: RCC Setup

As with most STM32 projects, the first thing we should do is enable the peripherals that we will use. In this case, that's just `GPIOA`, `GPIOB`, and `SPI1`. As in previous STM32 posts, I will use the device header files provided by ST for basic peripheral variable definitions, and determine the target chip from definitions passed in from the Makefile (https://en.wikipedia.org/wiki/Make_(software)):

```
#ifdef VVC_F0
  RCC->AHBENR   |= RCC_AHBENR_GPIOAEN;
  RCC->AHBENR   |= RCC_AHBENR_GPIOBEN;
#elif  VVC_L0
  RCC->IOPENR   |= RCC_IOPENR_IOPAEN;
  RCC->IOPENR   |= RCC_IOPENR_IOPBEN;
```

```
#endif
RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
```

Step 1: GPIO Setup

With the peripherals powered on, we need to set up the GPIO pins used for communicating with the screen. To power the `SSD1331` display in my previous tutorial, we configured all of the pins as ordinary push-pull outputs; as a quick refresher, we'll use the `SCK` (Clock), `MOSI` (Data output), `CS` (Chip Select), `D/C` (Data or Command?), and `RST` (Reset) pins. The only difference with using the hardware peripheral is that we should configure the `MOSI` and `SCK` pins as 'alternate function' with high-speed output.

There are actually multiple sets of pins mapped to the `SPI1` peripheral, even on the 32-pin STM32xKx chips. I'll use pin `B3` for `SCK` and pin `B5` for `MOSI`. Pin `B4` is mapped to `MISO`, but I'll use it as a general-purpose output to drive the `D/C` pin on the TFT. As long as the `MISO` pin is not configured as 'alternate function', the peripheral will ignore it and we can use pin `B4` as a normal GPIO pin. Finally, pins `A12` and `A15` are mapped to `CS` and `RST` respectively:

```
// Define GPIOB pin mappings for software '4-wire' SPI.
#define PB_MOSI (5)
#define PB_SCK  (3)
#define PB_DC   (4)
#define PA_CS   (12)
#define PA_RST  (15)

GPIOB->MODER   &= ~((0x3 << (PB_MOSI * 2)) |
                    (0x3 << (PB_SCK  * 2)) |
                    (0x3 << (PB_DC   * 2)));
// Set the MOSI and SCK pins to alternate function mode 0.
// Set D/C to normal output.
#ifdef VVC_F0
  GPIOB->AFR[0]  &= ~(GPIO_AFRL_AFSEL3 |
                      GPIO_AFRL_AFSEL5);
#elif  VVC_L0
  GPIOB->AFR[0]  &= ~(GPIO_AFRL_AFRL3 |
                      GPIO_AFRL_AFRL5);
#endif
GPIOB->MODER   |=  ((0x2 << (PB_MOSI * 2)) |
                    (0x2 << (PB_SCK  * 2)) |
                    (0x1 << (PB_DC   * 2)));
```

```
// Use pull-down resistors for the SPI peripheral?
// Or no pulling resistors?
GPIOB->PUPDR   &= ~((0x3 << (PB_MOSI * 2)) |
                    (0x3 << (PB_SCK  * 2)) |
                    (0x3 << (PB_DC   * 2)));
GPIOB->PUPDR  |=   ((0x1 << (PB_MOSI * 2)) |
                    (0x1 << (PB_SCK  * 2)));
// Output type: Push-pull
GPIOB->OTYPER  &= ~((0x1 << PB_MOSI) |
                    (0x1 << PB_SCK)  |
                    (0x1 << PB_DC));
// High-speed - 50MHz maximum
// (Setting all '1's, so no need to clear bits first.)
GPIOB->OSPEEDR |=  ((0x3 << (PB_MOSI * 2)) |
                    (0x3 << (PB_SCK  * 2)) |
                    (0x3 << (PB_DC   * 2)));
// Initialize the GPIOA pins; ditto.
GPIOA->MODER   &= ~((0x3 << (PA_CS   * 2)) |
                    (0x3 << (PA_RST  * 2)));
GPIOA->MODER   |=  ((0x1 << (PA_CS   * 2)) |
                    (0x1 << (PA_RST  * 2)));
GPIOA->OTYPER  &= ~((0x1 << PA_CS) |
                    (0x1 << PA_RST));
GPIOA->PUPDR   &= ~((0x3 << (PA_CS  * 2)) |
                    (0x3 << (PA_RST * 2)));
```

With the pins set up, it is also a good idea to set them all to a known starting state, and tell the ILI9341 display to reset by pulling the ‘Reset’ pin low/high with a delay to give the display time to perform its reset sequence:

```
// Set initial pin values.
//   (The 'Chip Select' pin tells the display if it
//    should be listening. '0' means 'hey, listen!', and
//    '1' means 'ignore the SCK/MOSI/DC pins'.)
GPIOA->ODR |=  (1 << PA_CS);
//   (See the 'sspi_cmd' method for 'DC' pin info.)
GPIOB->ODR |=  (1 << PB_DC);
// Set SCK high to start
GPIOB->ODR |=  (1 << PB_SCK);
// Reset the display by pulling the reset pin low,
// delaying a bit, then pulling it high.
GPIOA->ODR &= ~(1 << PA_RST);
// Delay at least 100ms; meh, call it 2 million no-ops.
delay_cycles(2000000);
```

```
GPIOA->ODR |=  (1 << PA_RST);
delay_cycles(2000000);
```

I don't want to complicate things by covering timers (https://vivonomicon.com/2018/05/20/bare-metal-stm32-programming-part-5-timer-peripherals-and-the-system-clock/) or precisely-timed assembly code in this tutorial, so I'm using a simple (but inaccurate) 'delay_cycles' method to give the display plenty of time to reset itself. If you want to use a better time-based delay, try waiting for about 100-150 milliseconds.

```
// Simple delay method, with instructions not to optimize.
// It doesn't accurately delay a precise # of cycles,
// it's just a rough scale.
void __attribute__((optimize("O0"))) delay_cycles(uint32_t cyc) {
  uint32_t d_i;
  for (d_i = 0; d_i < cyc; ++d_i) {
    asm("NOP");
  }
}
```

Step 2: Initializing the SPI Peripheral

The STM32's SPI peripheral resets to a convenient state for simple communication, but there are still a few options that we need to configure. First up is the clock 'polarity' and 'phase' – the `SCK` clock pin will toggle up and down as data is sent, and these two bits tell the peripheral when the data pins should be written and read. The 'clock polarity' defines the clock pin's resting state when data is not being transferred, and the 'clock phase' defines whether the devices should read data on the 'falling' or 'rising' edge of the clock signal. The ILI9341 seems to like a polarity and phase of either 1 and 1 or 0 and 0; you can inspect the timing diagram in its datasheet (https://cdn-shop.adafruit.com/datasheets/ILI9341.pdf), or just try and see what works best.

```
// Make sure that the peripheral is off, and reset it.
SPI1->CR1 &= ~(SPI_CR1_SPE);
RCC->APB2RSTR |=  (RCC_APB2RSTR_SPI1RST);
RCC->APB2RSTR &= ~(RCC_APB2RSTR_SPI1RST);
// Set clock polarity and phase.
SPI1->CR1 |=  (SPI_CR1_CPOL |
               SPI_CR1_CPHA);
```

Next, we need to tell the peripheral that the STM32 will be the one initiating communications by setting its `MSTR` flag. And to avoid unnecessary complexity, it is also a good idea to tell the STM32's SPI peripheral not to use its hardware `CS` pin – just like the ILI9341 has a `CS` ('Chip Select') pin which tells it whether it should listen to the clock/data lines, I think that the STM32 has a similar `CS` signal which tells it whether to read/write, called `NSS` in the datasheets. Fortunately, we can ignore all of that by using a software 'Chip Select' signal (the `SSM` flag) and leaving it 'high' to permanently enable communication (the `SSI` flag):

```
// Set the STM32 to act as a host device.
SPI1->CR1 |=  (SPI_CR1_MSTR);
// Set software 'Chip Select' pin.
SPI1->CR1 |=  (SPI_CR1_SSM);
// (Set the internal 'Chip Select' signal.)
SPI1->CR1 |=  (SPI_CR1_SSI);
```

Then all we have to do is set the `PE` (Peripheral Enable) flag to start communications. The ILI9341 expects its data to be sent with the MSB (Most-Significant Bit) first with 8 bits per data frame, but those are the default reset settings on the STM32's SPI peripheral so we don't need to change them:

```
// Enable the peripheral.
SPI1->CR1 |=  (SPI_CR1_SPE);
```

If you run into issues and want to get a better look at the signals on an oscilloscope or logic analyzer, you can slow the peripheral down by setting the three `BR` (Baud Rate) bits; they default to zero, and the peripheral's clock speed is divided by two to the power of their value plus one. So for example, you can slow things down by a factor of 256 by setting all of those bits before enabling the peripheral:

```
SPI1->CR1 |=  (0x7 << SPI_CR1_BR_Pos);
```

Step 3: Sending Data

With the peripheral initialized, it is pretty easy to send data – but there are a few important 'gotchas' which make things a little bit more complicated than simply writing bytes to the `SPI1->DR` 'Data Register'.

First, the STM32 has a small queue which it can use to store a few bytes of data while it is busy sending, and we shouldn't try to send data if that queue is full. The peripheral sets a `TXE` ('Transmit Buffer Empty') flag when it is ready for new data to be written. This means that our 'write 8 bits' function should wait for that flag to be set before continuing:

```
inline void hspi_w8(SPI_TypeDef *SPIx, uint8_t dat) {
  // Wait for TXE.
  while (!(SPIx->SR & SPI_SR_TXE)) {};
  // Send the byte.
  *(uint8_t*)&(SPIx->DR) = dat;
}
```

The second important thing to note in that function is that the Data Register is cast (https://www.cprogramming.com/tutorial/lesson11.html?sb=tutmap) to a pointer to an 8-bit integer. The peripheral behaves differently depending on how many bits are set in the register. If you simply write to the register – even with an expression like `SPI1->DR = (uint8_t)(dat & 0xFF);` – the peripheral will send a full 16 bits of data, and the extra byte of zeros will definitely confuse the ILI9341.

We will be sending 16 bits of color data per pixel though, so it is also useful to have a 'write 16 bits' function to use the full data register. But here we run into another quirk – ARM cores are Little-Endian (https://en.wikipedia.org/wiki/Endianness). So we need to reverse the order of the bytes to get the result that most people would expect – namely, having `hspi_w16(0x1234);` send the same data as `hspi_w8(0x12); hspi_w8(0x34);` That is simple with bit-shifting operations:

```
inline void hspi_w16(SPI_TypeDef *SPIx, uint16_t dat) {
#ifdef VVC_F0
  // Wait for TXE.
  while (!(SPIx->SR & SPI_SR_TXE)) {};
  // Send the data.
  // (Flip the bytes for the little-endian ARM core.)
  dat = (((dat & 0x00FF) << 8) | ((dat & 0xFF00) >> 8));
  *(uint16_t*)&(SPIx->DR) = dat;
#elif  VVC_L0
  hspi_w8(SPIx, (uint8_t)(dat >> 8));
  hspi_w8(SPIx, (uint8_t)(dat & 0xFF));
#endif
}
```

I actually couldn't get the STM32L0 line to write the full 16 bits this way – I think they have a slightly different peripheral configuration. Anyways, for the ILI9341's "4-Wire" SPI protocol, we also need to write a 'send command byte' method which pulls the `D/C` pin low during communication. Since we don't want to change the `D/C` pin while the peripheral is still sending data in its transmit queue, we should wait for the peripheral's `BSY` (Busy) flag to be cleared before changing the state of the `D/C` GPIO pin:

```
/*
 * Send a 'command' byte over hardware SPI.
 * Pull the 'D/C' pin low, send the byte, then pull the pin high.
 * Wait for the transmission to finish before changing the
 * 'D/C' pin value.
 */
inline void hspi_cmd(SPI_TypeDef *SPIx, uint8_t cmd) {
  while ((SPIx->SR & SPI_SR_BSY)) {};
  GPIOB->ODR &= ~(1 << PB_DC);
  hspi_w8(SPIx, cmd);
  while ((SPIx->SR & SPI_SR_BSY)) {};
  GPIOB->ODR |=  (1 << PB_DC);
}
```

That's all there is to it – now we just have to send some meaningful data to the display.

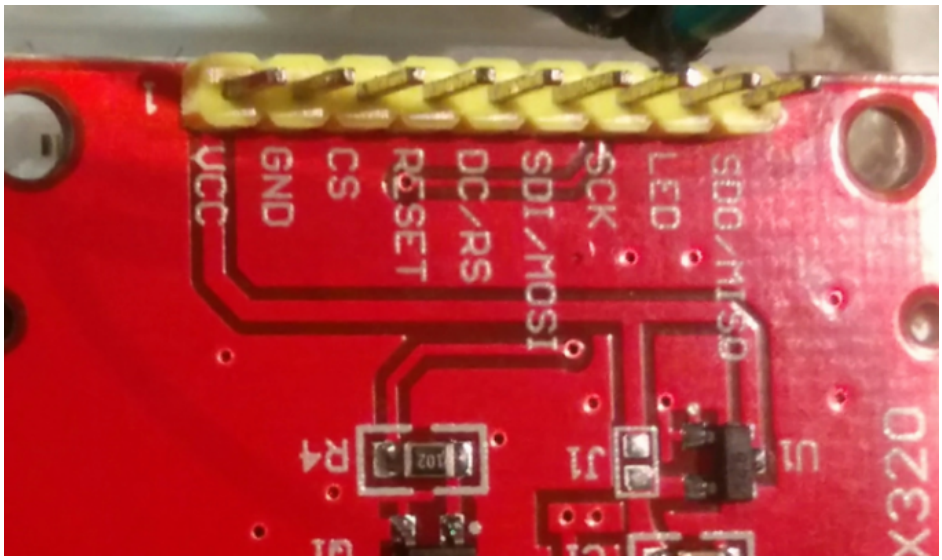
Step 4: ILI9341 Initialization

Connecting the Display

Initializing the display and drawing to it isn't too difficult, but if the previous steps aren't done properly, it can be frustrating to debug the communications. If you run into problems, you can also use the same 'software SPI' methods presented in the previous SSD1331 tutorial – just don't forget to set the `SCK` and `MOSI` pins as 'output' instead of 'alternate function' if you decide to try that.

The pins on your ILI9341 module should be labeled, although if you are using a generic module the labels might be on the back side of the board. The connections are about what you would expect; plug the `VCC` and `LED` pins into your board's `+3.3V` supply, and connect `GND` to Ground. The `CS`, `RESET`, `DC/RS`, `SDI/MOSI`, and `SCK` pins should connect to the corresponding microcontroller pins, and the `SDO/MISO` pin can be left unconnected. `DC/RS` is a different acronym for our `D/C` 'Data/Command' pin,

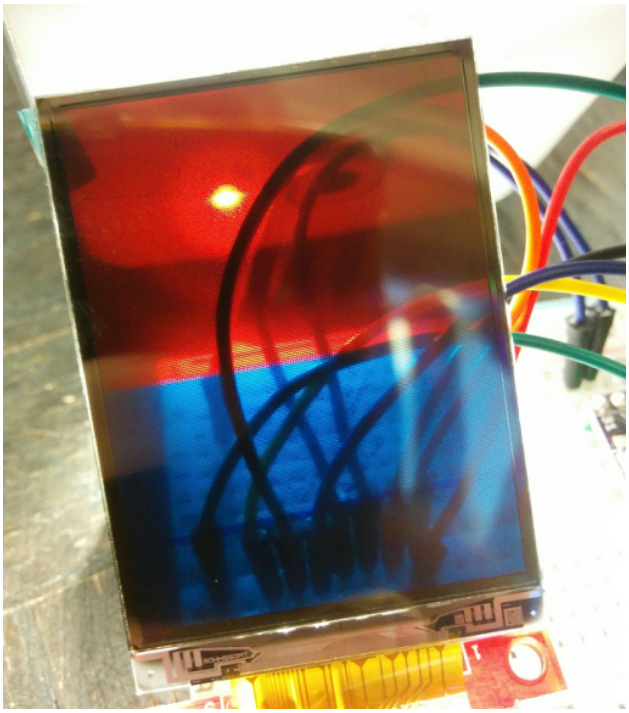
and `SDO/SDI` are starting to become popular labels on SPI boards – they stand for 'Serial Data Out/In', so `SDI` on the listening device corresponds to the `MOSI` SPI line and `SDO` is not needed since we won't be listening to the display.



ILI9341 Pins

Programming the Display

When the ILI9341 first powers on it should show a uniform bright white color, but that's just the backlight LEDs. The display will not try to show anything at all until it is initialized. Be aware that a broken display might still show a bright white screen when power is applied, but these modules are fairly sturdy. I've gone so far as to pry them apart and remove the backlights, and the panels worked even after being bluntly removed from the case.



This voids the warranty.

So short of taking a hammer to the screen, you shouldn't be able to damage them too much by bumping them around or dropping them from a tabletop. Anyways, to start the display and put it into a state where it can draw things, we need to send it a series of startup commands. Like with the SSD1331 display, most commands are followed by one or more 'option' bytes, but unlike the SSD1331, those 'option' bytes should be sent with the `D/C` pin held high, not low. You can see all of the commands in the ILI9341 datasheet (https://cdn-shop.adafruit.com/datasheets/ILI9341.pdf), but some commands appear to be undocumented, so it is a good idea to look at an existing library for a starting sequence that should work for most purposes.

Since Adafruit is awesome, they provide an ILI9341 library (https://github.com/adafruit/Adafruit_ILI9341) which is compatible with the Arduino IDE and devices which are supported by that – take a look at the `.cpp` file's `void Adafruit_ILI9341::begin(...)` method. The command macros such as `ILI9341_PWCTR1` are defined in the library's `.h` file. The `writeCommand` method is similar to our `hspi_cmd` one, and `spiWrite` is used to write a byte over the SPI protocol, like our `hspi_w8` method. So, our startup sequence can look something like this:

```
void ili9341_hspi_init(SPI_TypeDef *SPIx) {
  // (Display off)
  //hspi_cmd(SPIx, 0x28);

  // Issue a series of initialization commands from the
  // Adafruit library for a simple 'known good' test.
  // (TODO: Add named macro definitions for these hex values.)
  hspi_cmd(SPIx, 0xEF);
```

```
    hspi_w8(SPIx, 0x03);
    hspi_w8(SPIx, 0x80);
    hspi_w8(SPIx, 0x02);
    hspi_cmd(SPIx, 0xCF);
    hspi_w8(SPIx, 0x00);
    hspi_w8(SPIx, 0xC1);
    hspi_w8(SPIx, 0x30);
    hspi_cmd(SPIx, 0xED);
    hspi_w8(SPIx, 0x64);
    hspi_w8(SPIx, 0x03);
    hspi_w8(SPIx, 0x12);
    hspi_w8(SPIx, 0x81);
    hspi_cmd(SPIx, 0xE8);
    hspi_w8(SPIx, 0x85);
    hspi_w8(SPIx, 0x00);
    hspi_w8(SPIx, 0x78);
    hspi_cmd(SPIx, 0xCB);
    hspi_w8(SPIx, 0x39);
    hspi_w8(SPIx, 0x2C);
    hspi_w8(SPIx, 0x00);
    hspi_w8(SPIx, 0x34);
    hspi_w8(SPIx, 0x02);
    hspi_cmd(SPIx, 0xF7);
    hspi_w8(SPIx, 0x20);
    hspi_cmd(SPIx, 0xEA);
    hspi_w8(SPIx, 0x00);
    hspi_w8(SPIx, 0x00);
    // PWCTR1
    hspi_cmd(SPIx, 0xC0);
    hspi_w8(SPIx, 0x23);
    // PWCTR2
    hspi_cmd(SPIx, 0xC1);
    hspi_w8(SPIx, 0x10);
    // VMCTR1
    hspi_cmd(SPIx, 0xC5);
    hspi_w8(SPIx, 0x3E);
    hspi_w8(SPIx, 0x28);
    // VMCTR2
    hspi_cmd(SPIx, 0xC7);
    hspi_w8(SPIx, 0x86);
    // MADCTL
    hspi_cmd(SPIx, 0x36);
    hspi_w8(SPIx, 0x48);
    // VSCRSADD
    hspi_cmd(SPIx, 0x37);
    hspi_w8(SPIx, 0x00);
```

```
    // PIXFMT
    hspi_cmd(SPIx, 0x3A);
    hspi_w8(SPIx, 0x55);
    // FRMCTR1
    hspi_cmd(SPIx, 0xB1);
    hspi_w8(SPIx, 0x00);
    hspi_w8(SPIx, 0x18);
    // DFUNCTR
    hspi_cmd(SPIx, 0xB6);
    hspi_w8(SPIx, 0x08);
    hspi_w8(SPIx, 0x82);
    hspi_w8(SPIx, 0x27);
    hspi_cmd(SPIx, 0xF2);
    hspi_w8(SPIx, 0x00);
    // GAMMASET
    hspi_cmd(SPIx, 0x26);
    hspi_w8(SPIx, 0x01);
    // (Actual gamma settings)
    hspi_cmd(SPIx, 0xE0);
    hspi_w8(SPIx, 0x0F);
    hspi_w8(SPIx, 0x31);
    hspi_w8(SPIx, 0x2B);
    hspi_w8(SPIx, 0x0C);
    hspi_w8(SPIx, 0x0E);
    hspi_w8(SPIx, 0x08);
    hspi_w8(SPIx, 0x4E);
    hspi_w8(SPIx, 0xF1);
    hspi_w8(SPIx, 0x37);
    hspi_w8(SPIx, 0x07);
    hspi_w8(SPIx, 0x10);
    hspi_w8(SPIx, 0x03);
    hspi_w8(SPIx, 0x0E);
    hspi_w8(SPIx, 0x09);
    hspi_w8(SPIx, 0x00);
    hspi_cmd(SPIx, 0xE1);
    hspi_w8(SPIx, 0x00);
    hspi_w8(SPIx, 0x0E);
    hspi_w8(SPIx, 0x14);
    hspi_w8(SPIx, 0x03);
    hspi_w8(SPIx, 0x11);
    hspi_w8(SPIx, 0x07);
    hspi_w8(SPIx, 0x31);
    hspi_w8(SPIx, 0xC1);
    hspi_w8(SPIx, 0x48);
    hspi_w8(SPIx, 0x08);
    hspi_w8(SPIx, 0x0F);
```

```
  hspi_w8(SPIx, 0x0C);
  hspi_w8(SPIx, 0x31);
  hspi_w8(SPIx, 0x36);
  hspi_w8(SPIx, 0x0F);

  // Exit sleep mode.
  hspi_cmd(SPIx, 0x11);
  delay_cycles(2000000);
  // Display on.
  hspi_cmd(SPIx, 0x29);
  delay_cycles(2000000);
  // 'Normal' display mode.
  hspi_cmd(SPIx, 0x13);
}
```

After the display is reset and those commands are sent, the display should change to a flickering grey color. That tells you that the display is all set up and ready to go, but it has not received any pixel data yet so it is not showing any colors.

To draw to the display, we go through a similar process as we did with the SSD1331; send commands to say which rectangular area we want to draw to, then send one 16-bit color for each pixel in that rectangular area.

To refresh the entire 240 x 320 display, we can set the drawing area to be between (0, 0) and (239, 319) and then draw `320 * 240 = 76,800` pixels of data. That's a lot of data – even at one bit per pixel, the small chips used in this example would not have enough RAM to store a full framebuffer. You'd need over 600KB of RAM to store a full 16 bits of color per pixel, so we'll only draw some solid colors in this tutorial.

That sequence of commands looks like this, including a main loop:

```
// Main loop - empty the screen as a test.
int tft_iter = 0;
int tft_on = 0;
// Set column range.
hspi_cmd(SPI1, 0x2A);
hspi_w16(SPI1, 0x0000);
hspi_w16(SPI1, (uint16_t)(239));
// Set row range.
hspi_cmd(SPI1, 0x2B);
hspi_w16(SPI1, 0x0000);
hspi_w16(SPI1, (uint16_t)(319));
```

```
// Set 'write to RAM'
hspi_cmd(SPI1, 0x2C);
while (1) {
  // Write 320 * 240 pixels.
  for (tft_iter = 0; tft_iter < (320*240); ++tft_iter) {
    // Write a 16-bit color.
    if (tft_on) {
      hspi_w16(SPI1, 0xF800);
    }
    else {
      hspi_w16(SPI1, 0x001F);
    }
  }
  tft_on = !tft_on;
}
```

And that's all there is to it! As the program runs, your display should cycle between a red and blue color as fast as the chip can send data. This could be made even faster by using hardware interrupts and/or DMA transfers (https://en.wikipedia.org/wiki/Direct_memory_access), but that is a topic for a future tutorial.

Conclusions

The ILI9341 is a good display driver to know how to use. Screens using it come in sizes from about 2.2″ – 3.2″ with a resolution of 240 x 320 pixels, and they are very affordable. Their contrast is not as good as the SSD1331 OLED displays, but they get you a lot more pixels on a hobbyist's budget.

So all in all, they're nice choices for small applications which need an easy-to-read display. I've seen them used in devices ranging from handheld oscilloscopes to CNC machines. Chime in if you wind up making anything with one!

As usual, a project demonstrating this code is available on Github. (https://github.com/WRansohoff/STM32_ILI9341_HWSPI)

Comments (18):

Yesenia Poppleton (https://extraproxies.com/extract-files-to-sites-all-libraries/)
December 1, 2018 at 6:28 pm

When I originally commented I clicked the “Notify me when new comments are added” checkbox and now each time a comment is added I get three emails with the same comment. Is there any way you can remove people from that service? Appreciate it!

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=89#respond)

Vivonomicon

December 2, 2018 at 2:50 pm

Hm, sorry about that – I’m pretty new to this whole ‘blog’ thing. I hadn’t even realized there was a comment subscription option, but since you mentioned it I installed a plugin to hopefully make those more user-friendly.

I couldn’t find any existing subscriptions from within that plugin, but I think any emails that go out should have an unsubscribe or ‘manage subscriptions’ link which you can use. I also turned on an option to send a confirmation email before enabling these subscriptions, but let me know if you keep seeing issues.

Sorry!

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=90#respond)

kratatau

February 11, 2019 at 2:24 pm

Hi Vivonomicon,
Thank You for this excellent blog. Although I’m just a beginner, thanks to your explanations I was able to start with STM32 and moved your code to Bluepill (F103). Now, after a little tuning my ILI9341 works like a charm. Please continue with your enlightenment.
Greetings from Czechia.
kratatau

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=196#respond)

Vivonomicon

February 12, 2019 at 7:06 pm

I’m glad to hear this was helpful – the F103 cores definitely seem like reliable workhorses. Good luck, I hope your projects work well!

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=199#respond)

Massimo M.; Italy

February 12, 2019 at 2:54 am

I adapted this to a F072RB. Thanks a lot for yur help!
Now I'm trying to use DMA and a scalable fonts/other utilities for this display. Unfortunately I find too much example but extremely confused…
If someone have a good example like this why don't share the link?

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=197#respond)

Vivonomicon

February 12, 2019 at 7:02 pm

That's great to hear, congratulations! I would also like to learn more about DMA on the STM32, but unfortunately I haven't had time to look into it yet. You might look at the examples in the 'STCube' package distributed by ST, though, and I have also seen some blog posts about using DMA to drive 'neopixel' LEDs, like this one:

http://www.martinhubacek.cz/arm/improved-stm32-ws2812b-library (http://www.martinhubacek.cz/arm/improved-stm32-ws2812b-library)

It seems like that might be a good place to start, especially since many of those examples also use the SPI peripheral.

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=198#respond)

David

March 5, 2019 at 10:19 am

Hi, could please help me how do i display text or variables on screen?

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=238#respond)

Vivonomicon

March 30, 2019 at 12:35 pm

For that sort of thing, you would probably want to use an existing library – most of them will make a 'framebuffer' object which is just a big array with the current pixel colors. For a 16-bit TFT, you'd probably make it an array of `uint16_t`s. To refresh the display, you send the whole framebuffer one color at a time. And to 'draw' to it, you change the color values in the array.

It looks like most libraries implement helper methods for drawing shapes, text, etc. For an example that is designed for monochrome displays, maybe check out U8G2?

https://github.com/olikraus/u8g2 (https://github.com/olikraus/u8g2)

I was thinking of doing that in this tutorial, but a framebuffer for 320×240 pixels at 16 bits per pixel is 150KB, and not many cheap microcontrollers have that much RAM.

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=300#respond)

kratatau

April 10, 2019 at 5:11 am

Hi, I did it pretty much the way Vivonomicon described. You just need a ASCII font in the form of array bytes, where each byte describes a column. Then say 5 of these columns alongside make one character, – bits '1' with different color than '0'.
You can even magnify the font with a simple routine.
I used just a part of ASCII table to save the memory.
If you are interested, I can send you my code

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=324#respond)

Yitong Dai

March 18, 2019 at 7:19 pm

Really appreciate the tutorial. I am trying to use this display to implement a simple game. I wonder if SPI bandwidth will be the bottleneck. Also I try to modify Adafruit graphics library in order to use it wiht my STM32 board. One thing I noticed is that in Adafruit graphics library, startWrite() and endWrite() will be called before sending color data to the display. I know it has something to do with SPI, but I can't find the corresponding part in your tutorial. I am guessing in your tutorial you just use one transaction? If yes, is it necessary to have multiple transactions if I want to make a game?

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=279#respond)

Vivonomicon

March 30, 2019 at 12:42 pm

It might be, but I think that you can use clock rates as fast as 20-40MHz. There are also versions of these displays which support 8, 16, and 18-bit parallel interfaces if that is too slow; they're described in the datasheet, but I haven't figured out how they work yet.

I'm not sure what the 'start/endWrite' methods do without looking at the library, but SPI devices usually have a 'Chip Select' (CS) pin which tells the device when it is being addressed. You can have multiple devices share the same clock/data lines, and any chips without their CS pins asserted will ignore the data. With most devices, pulling the CS pin to V++ will put the device to 'sleep', while pulling it to Ground will activate it.

A lot of devices have the CS pin connected to V++ through a pull-up resistor, and some devices will misbehave if it is permanently pulled to Ground, so that can be a good place to look if you have trouble communicating with the display over SPI. For this tutorial, I think that 'startWrite' is probably the same as pulling the CS pin to Ground, while 'endWrite' equates to pulling it high.

Good luck, I hope your game turns out well!

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=302#respond)

Yitong Dai

March 19, 2019 at 8:45 pm

I wonder if SPI bandwidth will be the bottleneck if I want to use what's cover in the tutorial to have some interactive graphics.

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=283#respond)

Maunik Patel

March 26, 2019 at 12:16 am

Hi,

Thanks for this very helpful and well-described post.

I am trying to interface ILI9341V display with STM32F030C6, using 3-wire, 9-bit, Software SPI.
Here, I am not using D/C pin. Instead, I am appending an additional D/C 'bit' before sending Data/command.
The display initialization process is the same as the one you mentioned in this post.
I cross-checked SCK, SDA, and CS pins using CRO and seems like every pin is working fine.
Still, am seeing a pure white display, that doesn't flicker after initialization as you have mentioned in this post.

Can you help me to solve this unknown issue?

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=291#respond)

Vivonomicon

March 30, 2019 at 12:46 pm

Hm, I haven't used the 3-wire SPI mode before, but looking at the datasheet it looks like the D/C bit should be the first bit sent, followed by the byte MSB->LSB. Are you sure that it's being sent in that order? Also, have you confirmed that the display works with an existing library to rule out hardware or power supply issues?

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=303#respond)

Pythno

April 7, 2019 at 1:08 pm

Thank you very much for the writeup. I was trying today setting up the display on my "blue pill" but failed. So I was googling and found this. A few questions: The AFR and MODE Registers are not available for me. Also, the datasheet for the STM32F103c8x doesn't mention those registers at all. Do you know why? I have a book about the STM32 and it also uses MODE and AFR registers. A bit lost there 😉

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=316#respond)

Vivonomicon

April 8, 2019 at 11:38 am

Different ‘major versions’ of STM32s can have different peripheral layouts. Usually the registers and commands look fairly similar, but the “F1” series which the STM32F103 belongs to is one of the older lines, so a lot of its peripherals look different from the newer chips.

The GPIO peripheral is one example – take a look at the ‘GPIOs and AFIOs’ chapter (currently chapter 9) in the reference manual (PDF) (https://www.st.com/resource/en/reference_manual/cd00171190.pdf). The register descriptions are at the end of the chapter, and you can see that each GPIO bank has two ‘port configuration registers’ called ‘CRL’ and ‘CRH’. ‘CRL’ controls pins 0-7, and ‘CRH’ controls pins 8-15.

You can read the register descriptions for more information, but basically each pin gets a 2-bit ‘MODE’ setting and a 2-bit ‘CNF’ setting. The ‘MODE’ bits are similar to the ‘MODER’ and ‘OSPEEDR’ registers, in that they choose between input/output mode, and select the maximum speed for output mode. The ‘CNF’ bits choose between push-pull/open-drain/alternate-function modes if the pin is in output mode, or analog/floating/pull-up/pull-down settings if it is in input mode.

Also, if you configure a pin as ‘input with pull-up or pull-down’, I think that you might need to *write* a ‘1’ or ‘0’ to the pin to set the pulling direction.

And I don’t think I’ve looked too closely at the SPI peripheral on the STM32F1 series, but it might also look different from what I described in these posts – sorry about that, but good luck!

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=320#respond)

pythong

April 10, 2019 at 6:00 pm

thanks! I got it to work to be initialized, apparently. I get the grayish screen. It seems like alternating darker lines from top to bottom. However, I cannot get pixels on the screen. I tried several things. The byte order didn’t seem to do anything. By the way…. why don’t you do the CS before you transmit data? It seems to miss in your code snippets?

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=325#respond)

Vivonomicon

May 13, 2019 at 9:10 am

Well, in my experience that sort of grey screen with alternating bright/dark lines usually means that the display was initialized properly, but I’ve never used the 3-wire SPI mode before. I think it might use an extra bit instead of the ‘Data/Command’ pin, so maybe it could be interpreting your pixel data as commands if that isn’t being set correctly? Sorry, I’m not sure.

And the CS pin should be pulled low when a SPI transaction is happening, like with the software SPI tutorial – I think that should be in the full example code, but I might have omitted it in the snippets, sorry.

reply (/2018/06/17/drawing-to-a-small-tft-display-the-ili9341-and-stm32/?replytocom=387#respond)

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment*

Name* | Email* | Website

Notify me of followup comments via e-mail. You can also subscribe (https://vivonomicon.com/comment-subscriptions/?srp=457&srk=1bc58eb5a73ec3f3692a2bc3e7e5cf7a&sra=s&srsrc=f) without commenting.

POST COMMENT

RELATED POSTS:

(https://vivonomicon.com/2019/05/23/hello-rust-blinking-leds-in-a-new-language/)

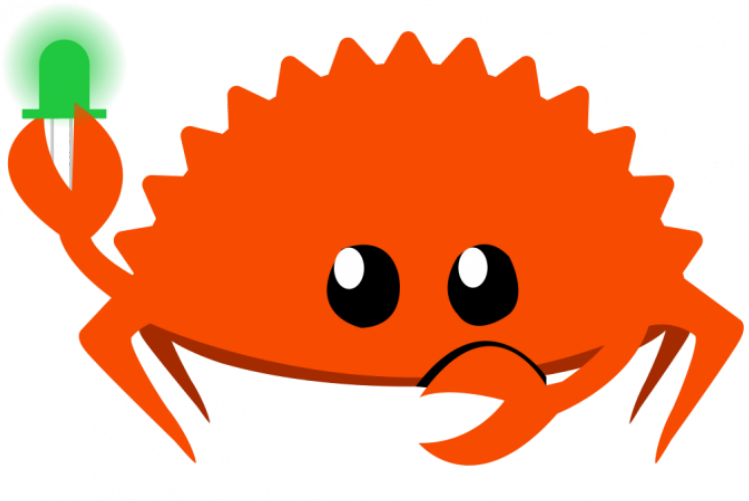
MAY 23, 2019 (HTTPS://VIVONOMICON.COM/2019/05/23/HELLO-RUST-BLINKING-LEDS-IN-A-NEW-LANGUAGE/) Random (https://vivonomicon.com/category/random/), STM32 Baremetal Examples (https://vivonomicon.com/category/stm32_baremetal_examples/)

Hello, Rust: Blinking LEDs in a New Language (https://vivonomicon.com/2019/05/23/hello-rust-blinking-leds-in-a-new-language/)

Rust (https://www.rust-lang.org/) is a fairly new language that has gotten to be very popular in recent years. And as the language matures, it has started to support a wider set of features, including compilation and linking for bare-metal targets. There is an excellent "Embedded Rust" ebook (https://rust-embedded.github.io/book/) being written which covers the concepts that I'll talk about here, but it's still in-progress and there aren't many turn-key code examples after the first couple of chapters.

The Rust language is less than 10 years old and still evolving, so some features which might change in the future are only available on the `nightly` branch at the time of writing; this post is written for `rustc` version 1.36. And the language's documentation is very good, but it can also be a little bit scattered in these early days. For example, after I had written most of this post I found a more comprehensive "Discovery ebook" (https://docs.rust-embedded.org/discovery/) which covers hardware examples for an `STM32F3` "Discovery Kit" board. That looks like a terrific resource if you want to learn how to use the bare-metal Rust libraries from someone who actually knows what they're talking about.

As a new Rustacean (https://rustacean.net/), I'll admit that the syntax feels little bit frustrating at times. But that's normal when you learn a new language, and Rust is definitely growing on me as I learn more about its aspirations for embedded development. Cargo looks promising for distributing things like register definitions, HALs, and BSPs. And there's an automated `svd2rust` utility for generating your own register access libraries from vendor-supplied SVD (http://www.keil.com/pack/doc/CMSIS/SVD/html/index.html) files, which is useful in a language that hasn't had time to build up an extensive set of well-proven libraries. So in this post I'll talk about how to generate a "Peripheral Access Crate" for a simple `STM32L031K6` chip, and how to use that crate to blink an LED.



It's kind of fun when languages have mascots, especially when they're CC0-licensed.

The target hardware will be an STM32L031K6 Nucleo-32 (https://www.digikey.com/short/p4bv41) board, but this should work with any `STM32L0x1` chip. I also tried the same process with an `STM32F042` board and the

`STM32F0x2` SVD file, which worked fine. It's amazing how easy it is to get started with a new chip compared to C, although you do still need to read the reference manuals to figure out which registers and bits you need to modify. This post will assume that you know a little bit about how to use Rust, but only the very basics – I'm still new to the language and I don't think I would do a good job of explaining anything. The free Rust ebook is an excellent introduction (https://doc.rust-lang.org/book/), if you need a quick introduction.

READ MORE (HTTPS://VIVONOMICON.COM/2019/05/23/HELLO-RUST-BLINKING-LEDS-IN-A-NEW-LANGUAGE/)

(https://vivonomicon.com/2019/01/13/keeping-up-with-the-moorses-learning-to-use-stm32g0-chips/)

JANUARY 13, 2019 (HTTPS://VIVONOMICON.COM/2019/01/13/KEEPING-UP-WITH-THE-MOORSES-LEARNING-TO-USE-STM32G0-CHIPS/) Circuitry (https://vivonomicon.com/category/circuitry/), STM32 Baremetal Examples (https://vivonomicon.com/category/stm32_baremetal_examples/)

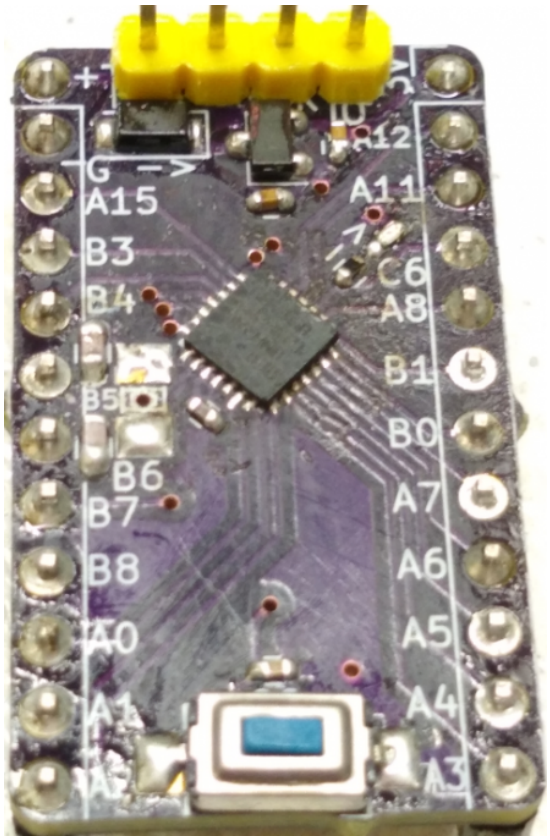
Keeping Up With the Moorses: Learning to Use STM32G0 Chips (https://vivonomicon.com/2019/01/13/keeping-up-with-the-moorses-learning-to-use-stm32g0-chips/)

Microcontrollers are just like any other kind of semiconductor product. As manufacturers learn from customer feedback and fabrication processes continue to advance, the products get better. One of the most visible metrics for gauging a chip's general performance – and the basis of Moore's Law (https://en.wikipedia.org/wiki/Moore%27s_law) – is how large each transistor is. Usually this is measured in nanometers, and as we enter 2019 the newest chips being made by companies like Samsung and Intel are optimistically billed as 7nm.

The venerable and popular STM32F1 series is more than a decade old now, and it is produced using a 130nm process. But ST's newer lines of chips like the STM32L4 use a smaller 90nm process. Smaller transistors usually mean that chips can run at lower voltages, be more power-efficient, and run at faster clock speeds. So when ST moved to this smaller process, they introduced two types of new chips: faster 'mainline' chips like the `F4` and `F7` lines which run at about 100-250MHz, and more efficient 'low-power' chips like the `L0` and `L4` lines which have a variety of 'sleep' modes and can comfortably run off of 1.8V. They also have an `H7` line which uses an even smaller 40nm process and can run at 400MHz.

Now as 2018 fades into history, it looks like ST has decided that it's time for a fresh line of 'value-line' chips, and we can order a shiny new STM32G0 (https://blog.st.com/stm32g0-mainstream-90-nm-mcu/) from retailers like Digikey.

At the time of writing there aren't too many options, but it looks like they're hoping to branch out and there are even some 8-pin variants on the table. I could be misreading things, but these look like a mix between the `F0` and `L0` lines, with lower power consumption than `F0` chips and better performance than the `L0` chips. The STM32G071GB (https://www.st.com/en/microcontrollers/stm32g071gb.html) that I made a test board with has 128KB of Flash, 36KB of RAM, and a nice set of communication peripherals.



Simple STM32G0 breakout board

So what's the catch? Well, this is still a fairly new chip, so "Just Google It" may not be an effective problem-solving tool. And it looks like ST made a few changes in this new iteration of chips to provide more GPIO pins in smaller packages, so the hardware design will look similar but slightly different from previous STM32 lines (https://vivonomicon.com/2018/05/05/your-own-hardware-designing-an-stm32-development-board/). Finally, with a new chip comes new challenges in getting an open-source programming and debugging toolchain working. So with all of that said, let's learn how to migrate!

READ MORE (HTTPS://VIVONOMICON.COM/2019/01/13/KEEPING-UP-WITH-THE-MOORSES-LEARNING-TO-USE-STM32G0-CHIPS/)

(https://vivonomicon.com/2018/12/28/bare-metal-stm32-programming-part-8-learn-to-debug-timing-issues-with-neopixels/)

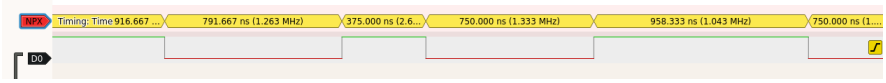
DECEMBER 28, 2018 (HTTPS://VIVONOMICON.COM/2018/12/28/BARE-METAL-STM32-PROGRAMMING-PART-8-LEARN-TO-DEBUG-TIMING-ISSUES-WITH-NEOPIXELS/) STM32 Baremetal Examples (https://vivonomicon.com/category/stm32_baremetal_examples/), Talking to Hardware (https://vivonomicon.com/category/talking_to_hardware/)

"Bare Metal" STM32 Programming (Part 8): Learn to Debug Timing Issues with Neopixels (https://vivonomicon.com/2018/12/28/bare-metal-stm32-programming-part-8-learn-to-debug-timing-issues-with-neopixels/)

I haven't written about STM32 chips in a little while, but I have been learning how to make fun displays and signage using colorful LEDs, specifically the WS2812B (https://www.seeedstudio.com/document/pdf/WS2812B%20Datasheet.pdf) and SK6812 (https://cdn-shop.adafruit.com/product-files/1138/SK6812+LED+datasheet+.pdf) 'Neopixels'. I talked about the single-wire communication standard that these LEDs use in a post about running them from an FPGA (https://vivonomicon.com/2018/12/24/learning-how-to-fpga-with-neopixel-leds/), and I mentioned there that it is a bit more difficult for microcontrollers to run the communication standard. If you don't believe me, take a look at what the poor folks at Adafruit needed to do (https://github.com/adafruit/Adafruit_NeoPixel/blob/f890ac8caaff7767824c9d639f3daed5c15e8d06/Adafruit_NeoPixel.cpp#L204) to get it working on a 16MHz AVR core. Yikes!

When you send colors, the `1` bits are fairly easy to encode but the `0` bits require that you reliably hold a pin high for just 250-400 nanoseconds. Too short and the LED will think that your `0` bit was a blip of noise, too long and it will think that your `0` is a `1`. Using timer peripherals is a reasonable solution, but it requires a faster clock than 16MHz and we won't be able to use interrupts because it takes about 20-30 clock cycles for the STM32 to jump to an interrupt handler. At 72MHz it takes my code about 300-400 nanoseconds to get to an interrupt handler, and that's just not fast enough.

There are ways to make it faster, but this is also a good example of how difficult it can be to calculate how long your C code will take to execute ahead of time. Between compiler optimizations and hardware realities like Flash wait-states (https://en.wikipedia.org/wiki/Wait_state) and pushing/popping functions (https://www.csee.umbc.edu/~chang/cs313.s02/stack.shtml), the easiest way to tell how long your code takes to run is often to simply run it and check.



Pulseview diagram of '101' in Neopixel. I can't be sure, but I think the '0' pulse might be about 375 nanoseconds long.

Which brings us to the topic of this tutorial – we are going to write a simple program which uses an STM32 timer peripheral to draw colors to 'Neopixel' LEDs. Along the way, we will debug the program's timing using Sigrok and Pulseview with an affordable 8-channel digital logic analyzer. These are available for $10-20 from most online retailers; try Sparkfun (https://www.sparkfun.com/products/15033), or Amazon/eBay/Aliexpress/etc. I don't know why Adafruit doesn't sell these; maybe they don't want to carry cheap generics in the same category as Salae analyzers. Anyways, go ahead and install Pulseview (https://learn.sparkfun.com/tutorials/using-the-usb-logic-analyzer-with-sigrok-pulseview/all), brush up a bit on STM32 timers if you need to (https://vivonomicon.com/2018/05/20/bare-metal-stm32-programming-part-5-timer-peripherals-and-the-system-clock/), and let's get started!

READ MORE (HTTPS://VIVONOMICON.COM/2018/12/28/BARE-METAL-STM32-PROGRAMMING-PART-8-LEARN-TO-DEBUG-TIMING-ISSUES-WITH-NEOPIXELS/)