Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.115 Microprocessor Project Laboratory Spring 2020

Final Project Report
Image Processing – the Hard Way

Holly Jackson Due: May 12, 2020

1. INTRODUCTION

1.1. Brief Project Overview

In my final project, I created a simple GUI to perform efficient image processing operations – such as blur, edge detection, and inversion – on images captured on the fly. My setup includes only three parts: a PSoC 5LP "Big Board", a color TFT display with 320x240 pixel resolution, and an Arduino compatible color camera.

Using the potentiometer and a button on the PSoC 5LP, the user can select from six menu options displayed on the TFT screen (see **Figure 1**). They can select an image to process from three options: "Take Pic," "Live," and "Sample." Pressing the button on the "Take Pic" options captures an image in near-real-time from the camera and transfers it to the color TFT display. "Live" continuously captures sequential images from the camera (rough every 0.87 seconds) and updates the TFT display until paused by a second button press. "Sample" displays a common image processing test image of peppers.

After displaying an image, the user can then select from three image processing operations: "Gradient," "Blur," and "Invert." "Gradient" takes the 2D gradient of the image by convolving the greyscale intensity of the image with 3x3 Sobel filters in x and y (as defined by **[1]**), which has the effect of edge detection. "Blur" convolves the color image with a 3x3 Gaussian blur kernel. "Invert" takes the bitwise inversion of every pixel in the image. The effect is similar to a photo negative.

These functions can be used individually and also can be applied iteratively. For example, a user could take a photo in "Live" mode (by pressing "Live" and then pausing the livestream to select her desired image) and then apply the "Blur" filter. If she pressed "Blur" again, it would apply the blur filter to the previously blurred image to increase the blur effect. On top of that, she could iteratively apply other functions such as inverting her double-blurred image, or any other combination she desires. However, once she applies a filter, she cannot return the original image (with the sole exception of pressing "Invert" twice consecutively).

1.2. Challenges

Before starting my project, the main challenge I anticipated was storing the images I wished to process in the PSoC, since the PSoC has very limited RAM. I wanted to display 240x240 images on my TFT screen. A 240x240 RGB565 (2 bytes per pixel) image takes up 112.5 kB of memory, and the PSoC only has 64 kB RAM. My original plan was to use the

graphics memory provided inside the TFT display controller (ILI9341) to hold variations of my image captured from the camera. However, when I began my project, I discovered that this solution was not possible and encountered many additional challenges along the way.

First, I realized upon a more thorough reading of the ILI9341 datasheet that the graphics memory only had enough space to hold exactly the image that was being displayed on the screen and no more. As a result, I updated my plan. The new plan was that I would read back the image displayed on the screen in small sections (with the help of the TFT's SPI memory read command and the `MISO` pin), process it using the PSoC, and finally write the processed image back on the screen. Even though this was going to be less efficient than my original solution, it seemed like a solid plan.

However, when I began implementing a read function from the TFT's SPI port, I began having new problems. Reading back from the TFT screen was poorly documented, and the few people who had tried it were unsuccessful. One person even claimed that there was a hardware defect in the screen design leaving the `MISO` pin disconnected (see **[2]**). Most standard projects only write to the screen and would not notice this defect. The MISO pin is unused for a write.

My symptoms matched the trouble other people were having in my internet searches. When I used Creator's "Debug" mode to single step and look into the output of the `MISO` pin I was "reading" from the TFT screen, I would get only black (`0x00`). I suspected the `MISO` pin was disconnected and sitting low. I was able to verify the pin was floating by configuring the input with a weak pull-up, which changed my reads to `0xFF`. Since reading the TFT memory did not work, I had to find an alternative solution. I will discuss my alternate solution (buffering and scaling the image) in the **Section 3**.

A second challenge I faced was the voltage limits of the camera and TFT screen. Both the camera and screen required 3.3 V I/O. The tutorial provided by the 6.115 staff for setting up the TFT screen uses voltage dividers to take the 5 V I/O on the PSoC 5 stick and translate it to 3.3 V at the screen. First off, I did not have resistors, but this was quickly resolved by a last-minute MPJA order. Irregardless of the TFT, the camera required 3.3 V power. This motivated me to change from my original plan of using the PSoC 5LP "Stick" (as detailed in the TFT tutorial) to substituting the PSoC 5LP Big Board, which was already configured for 3.3 V power and I/O.

However, I had to be careful and verify that the power line and each I/O pin were properly outputting 3.3 V. First, I made sure my board jumpers were configured for 3.3 V (see **Figure 2a**). Then, for each pin I planned to use, I raised it to a logic high in Creator and verified that it was outputting 3.3 V with a multimeter (I upgraded to a very nice Fluke digital multimeter I stole from my dad as soon as I got home, one of the few benefits of a stay-at-home order). I show an example of this for one pin in **Figure 2b-e**. After these tests, I was safe to wire my camera and screen to the PSoC 5LP "Big Board."

Finally, the biggest challenge I encountered – which will be discussed thoroughly in the hardware and software sections of this report – was the I^2C initialization and parallel data output timing of the camera. The datasheet for the camera controller (OV7670) was incredibly unclear about how to properly initialize the dozens of control registers. A lot of internet research was required before I figured out how to properly initialize the camera in the mode I wanted (i.e. outputting RGB565 pixel values). In addition, I struggled with finding a balance between the

high clock speed the camera required to accept initialization commands through the I^2C interface and the low clock speed required so that the pixel output rate of the camera was slow enough to read in software with the PSoC. I was able to use the programmable logic of the PSoC to create a two-stage latching system which allowed me to capture four bytes of data from the camera in hardware. This allowed me to raise the clock speed to a high enough rate for the I^2C while relaxing the speed required in software. This system will be discussed in the next section.

2. HARDWARE & FIRMWARE

After a thorough reading of the datasheets for the TFT display controller (ILI9341) and the camera controller (OV7670), I connected my camera, PSoC, and TFT screen as shown in the schematic in **Figure 3.** The real-life setup is shown in **Figure 4**. I assigned all the pins on the camera and screen according to **Figure 6**.

In addition to just wiring everything together. I needed to create "liquid metal hardware" in PSoC Creator in order to make my menu interface (**Figure 1**) and to facilitate communication between the camera, PSoC, and TFT screen. My schematic for this firmware is shown in **Figure 5.**

2.1. Menu Selection

The simplest firmware in my project is the menu selection (see the box labeled MENU SELECTION in **Figure 5**). To select options on the menu, I used a combination of the potentiometer and user button at P6[1] built in to the PSoC 5LP "Big Board." Spinning the potentiometer allows the user to scroll through the six options in the menu, and the button allows her to select which option she wants (in addition to starting and pausing "Live" mode). The firmware for this was mostly based on Exercise 4 from Kovid Lab 2, which used an ADC to digitize the potentiometer output. However, I added a set-reset (SR) flip-flop to "latch" the NOT of the button output (pin 6[1] goes low when the button is pressed). Based on what I had learned about SR flip-flops from the Van Ess labs, I incorporated a SR flip-flop to latch (set) high as soon as the button is pressed and to reset using a digital input pin, `Button_Reset`, written by software after the requested action is complete. The combination effectively debounced the button while still being responsive.

2.2. TFT Screen & Its SPI Interface

To interface the PSoC and the TFT screen, I used a serial-peripheral interface (SPI), as directed in the TFT tutorial created by the 6.115 staff (see the schematic in the box labeled TFT in **Figure 5**). I configured it for both reading and writing capabilities (with the `MISO` and `MOSI` pins, respectively), however I only ended up writing to the screen because of the defect with the `MISO` pin as described in **Section 1**. A fast clock rate of 12 Mbps was used to minimize delay writing large amounts of data to the screen.

2.3. The Camera, I²C, and a Complex Latching System

The hardware and firmware for the camera was the most complex. The camera has eighteen pins. Two are for power (`3.3V` and `GND`). There is also a `RESET` (always set to 1) and a `PWDN` (always set to 0), which I only change at initialization. The `SDA` (serial data) and `SCL` (serial clock) pins connect to an I²C master interface, which allows writing values to the registers in the camera controller (which sets things like the color mode, image size, clock divider, etc.). Eight data lines (`D0` through `D7`) output the data for each image frame clocked in byte-by-byte, in whichever format the camera is initialized to output (e.g. RGB444, RGB555, RGB565, etc.).

The combination of output control pins `VS`, `HS`, and `PCLK` indicate when a data byte is available and where it lies within the frame. Based on the default settings, `VS` (vertical synchronization) goes high momentarily when a new frame is being captured by the camera. `HS` (horizontal synchronization) is high while byte data for each row of pixels is being output. `HS` momentarily goes low between each row. `PCLK` goes high every time a data byte is read at the data lines (`D0-D7`). Finally, `MCLK` (master clock) is an input to the camera and sets the clocking frequency used by the camera controller.

To connect the camera, I first set up an I²C interface between the camera and the PSoC using the `SDA` and `SCL` pins on the camera. My I²C was configured for a relatively slow 10 kbps data rate. Speed was not important as this interface is only used during initialization. In the software section, I use this interface to write registers in the camera that set it to output RGB565 data and divide `MCLK` to reduce the frequency of `PCLK`.

I used a programmable clock to drive the camera's `MCLK` pin. This master frequency determines the rate and which the I²C system can work and also the frequency at which data is output from the data lines `D0-D7`. This is where I encountered a difficult problem which was solved through trial and error. Any frequency of `MCLK` below ~1.25 MHz seemed to be too slow for proper I²C function; the camera would not initialize correctly and begin outputting gibberish to the TFT screen. But, any frequency above 875 kHz was too fast for the software to read each sequential output byte (data would be missed); I could clearly see regions where pixels had been skipped drawing the image to the TFT screen. Even when I applied a clock divider to `MCLK` to slow down `PCLK` (which could be done by writing a specific value to a register in the OV7670 controller through the I²C), the frequency was either still too fast or was beginning to adversely affect the scaling of the image (this will be covered more in the software section).

To solve this issue, I decided that I needed a system in hardware that would buffer several bytes of camera data and allow the software to read multiple bytes at once. I designed a dual-stage latching system that allowed me to buffer two pixels, or four bytes of RGB565 data, from the camera. The complete schematic of this latching system is shown in the box labeled CAMERA in **Figure 5**.

To create the latch, I used a two-bit counter (00, 01, 10, 11) clocked by `PCLK` and reset by $\overline{\text{HREF}}$. As a result, the counter would restart at the beginning of each new row and it would roll over after every set of 4 data bytes. When the counter is 00, it enables a group of D flip-flops (8 bits wide) to latch the current contents of the data lines `D0-D7`, and outputs that data – the high byte of pixel n – to be stored latched again later in a status register (`High_Byte_0`). One cycle

of PCLK later, the counter is 01, and a second D flip-flop group latches D0-D7, storing the low byte of pixel *n* and passing it to a second status register (Low_Byte_0). When the counter is 10, a third flip-flop group and status register handle the high byte of pixel *n+1* (High_Byte_1). Finally, when the counter is 11, a fourth flip-flop group and status register handle the low byte of pixel *n+1* (Low_Byte_1). A timing diagram of the process described above is shown visually in **Figure 9.**

However, I still needed a way to indicate to my software the moment when these four status registers were ready with valid data to read. So, I created a pixel latch flag using D flip-flops to combine VS, HS, and the two-bit counter to provide a flag to tell my software when valid pixel data was available in the four status registers using the pin Pixel_Latch. In **Figure 7**, the timing diagram from the OV7670 datasheet shows us that valid data can be read from the camera when VS in low, HS/HREF is high, and PCLK is rising. By ANDing the outputs of a D flip-flop with input $\overline{VS}$ (VS is referred to as VSYNC in the schematic) and with input HS (referred to as HREF in the schematic), and clocking both when the counter rises to 01, I guarantee that my two pixels (four data bytes) are read when VS is low (i.e. after its initial pulse signifying the start of a frame), when HS is high (i.e. I was in a valid row of pixels), and after the four registers are full. I can clear the latch as soon as I read the data in software by writing to the pin Latch_Clear. This must be done within four cycles of PCLK. **Figure 9** also shows the timing of Pixel_Latch.

Using this system, I was able to connect a 2 MHz clock to MCLK, which was amply high enough for I^2C operation. And by configuring the camera controller to divide MCLK by 8, my software was now able to read data from a PCLK rate of 250 kHz. Without the hardware buffer (reading one data byte at a time), this frequency would have been too high. With the latching system, I have ample time to read my pixels.

3. SOFTWARE

The complete software code that I wrote for my final project can be found in **Appendix 2.** My main function menu loop and image processing functions are in main.c (see **Appendix 2A**). Auxiliary functions to initialize the camera can be found in camera.c (see **Appendix 2B**). In addition, I used example code provided by the 6.115 staff in tft.c (see **Appendix 2C**) to write to the TFT screen.

3.1. Brief Intro: What is RGB565?

Before I discuss the details of my software implementation, I will quickly discuss a conversion that appears many times in my code. In order to cheaply store the RGB data, the TFT screen I use stores the pixel values in RGB565 format. In the RGB565 color format, red is represented by a 5-bit value (from 0 to 31), green is represented by a 6-bit value (from 0 to 63), and blue is represented by a 5-bit value (from 0 to 31). Classically, colors are represented by the RGB888 format, which represents red, green, and blue with three 8-bit values ranging from 0 to 255. RGB888 provides 16,777,216 colors, while RGB565 only provides 65,536 colors. As a

result, RGB565 significantly increases the efficiency of the screen, but at a small cost to the color quality that is not easily noticeable to the observer.

When I perform blur and gradient operations on my images (which require convolution of the image with a 3x3 kernel), I must convert the RGB565 values to RGB888 values so I can isolate and independently process the red, green, and blue values. I follow simple conversion rules detailed in **[3]**. To convert from RGB565 16-bit pixel to RGB888, I use the following formulas:

- uint8 red = ((pixel & 0x7C00) >> 10) << 3
- uint8 blue = ((pixel & 0x03E0) >> 5) << 3
- uint8 green = (pixel & 0x001F) << 3

To convert from RGB888 back to RGB565, I use the following formula:

- uint16 pixel = (((red >> 3) & 0x1F) << 11) | (((green >> 2) & 0x3F) << 5) | ((blue >> 3) & 0x1F)

In the following sections, I will not explicitly specify when these conversions occur.

3.1. Initializing the Camera

In `camera.c` (see **Appendix 2B**), I wrote two functions to initialize the camera controller: `cameraStart()` and `writeBytes()`. The function `cameraStart()` performs a complete hardware reset of the camera (by setting the `RESET` pin to 0, delaying, and then raising it to 1). Then, the I^2C component is initialized, and a software reset is performed by writing `0x80` to register `0x12`, which according to the OV7670 datasheet resets the camera. To write registers in the camera, I use the PSoC's built-in I^2C command `MasterSendStart()`. The OV7670 datasheet explains that `0x42` is the device address for write and `0x43` is the device address for read. But I cannot simply call `MasterSendStart(0x42)` to write to the registers of the OV7670 chip because, according to the I^2C component documentation, `MasterSendStart()` has two arguments. The first is 7-bit left-shifted device address (which for `0x42` and `0x43` would be `0x21`), and the second is a 0 or nonzero number; a 0 signifies a write, and a nonzero number signifies a read. So, to start a write, I must call `MasterSendStart(0x21, 0)` for the I^2C component.

After the hardware and software reset of the camera, I call `writeBytes()`, which writes the configuration settings for the camera based on the OV7670 datasheet. Surprisingly, this was one of the most difficult challenges of the project. The default register values of the OV7670 do not put it into a functional state; the camera will output no valid data after reset. After thoroughly reading and re-reading the datasheet, I compiled a list of registers and values which I thought would initialize the camera with the exact settings I wanted, such as setting the `MCLK/PCLK` clock divider and configuring the color mode to RGB565. But, when I tested these settings, I could not get the camera to output even the semblance of an image. So, I turned to the internet. I found many references to people having the same problem because the only publicly available datasheet available for the camera controller online was marked "preliminary." I tried several combinations of register initialization values I found on the internet to see if I could get the camera to output anything resembling an image using one of these configurations.

Finally, using some code from **[4]**, I was able to configure the camera in RGB565 mode. (This code referenced several "magic" values from other internet sources as the only way to get the camera to turn on. None of these magic values were documented in the camera's preliminary datasheet.) I slightly modified this set of registers to set register `0x3E` to `0x13` (the original code sets `0x3E` set to `0x00`). Register `0x3E`, among other things, controls the clock divider between the input clock `MCLK` and the output clock `PCLK`. A value of `0x00` sets the clock divider to 1. Even with my new latching system, having `PCLK` = `MCLK` = 2 MHz was still too fast (and software would miss data), so I changed the clock divider to 8 so `PCLK` = `MCLK`/8 = 250 kHZ (signified by `0x13`).

With these initialization settings, I was finally able to read a color image from the camera and draw it to the screen. However, I faced one more problem. One or more of these register/value pairs I from my internet source (which I have yet to identify) was causing the image to output a distorted 80x360 image that appeared to repeat horizontally. Instead of draining more energy trying to isolate which mystical, undocumented register/value pair would fix this, I decided to use this glitch to my advantage. As I mentioned in the introduction, storing a complete 240x240 RGB565 image is impossible with the limited PSoC RAM and could only be accomplished using at least four buffered sections of 28.1 kB. If I instead isolated one of these distorted 80x360 tiles, and I scaled it down to 80x240 (i.e. 2/3 of one tile of the image), I could store the entire thing in the PSoC RAM because it would only occupy 37.5 kB (a little more than half of 64 kB RAM available). I re-scaled the image to 240x240 as it was displayed on the TFT.

3.2. Finding an Image to Process: Take Pic, Live, and Sample

I extract this small tile in the function `captureImage()` within `main.c` — which is called when "Take Pic" is pressed. This function reads an image from the camera and crops it to a single 80x240 section. In `captureImage()`, I poll for the first pulse from `VSYNC` and then set a flag `startFrame` to signal when a new frame capture should begin. Using `startFrame` ensures that I do not begin reading an image mid-frame and always start from the first (upper left) pixel.

Once `startFrame` has been set, I poll if `Pixel_Latch` is high with `Pixel_Latch_Read()`. When `Pixel_Latch` is high, it means that four data bytes (i.e. two pixels) are available to be read from the buffered status registers. Because of the extra time I gained from using the latching system, I am able to check the (x,y) coordinates of the current pixel being read in to see if it is within the 80x360 window of the tile. In addition, I filter out 1/3 of the pixel rows to scale my final image stored to be 80x240 pixels. After reading the two pixels (four bytes) from the status registers, I reset `Pixel_Latch` by writing a 1 and then a 0 to `Latch_Clear`. The process is repeated until the full 80x240 frame is stored in a 37.5 kB array called `image_buffer`. When I display the contents of `image_buffer` on the TFT screen, I want to scale it back up to remove distortion and make it fill the 240x240 section above the menu on the screen. To do this, I call `drawType0Image()`, which writes each pixel three times to expand the final image to be 240x240 pixels.

It is important to note, as can be seen in **Figure 9**, the first four bytes that I read of each image are actually invalid data. I do not actually remove this invalid data from `image_buffer` or before I draw the image to the screen because its effects are relatively inconsequential on the final product. It does create two columns of incorrect pixel data on the left of the image, which is only obvious if you are really scrutinizing the images.

When a user presses "Live," the function `live()` within `main.c` is called. The `live()` function calls `captureImage()` repeatedly – filling and re-filling `image_buffer` – until a second button press stops the cycle. However, at least two frames must be rendered before the user is allowed to stop the livestream. This effectively debounces the button giving the use time to release it before pressing a second time. The last photo taken is the data that remains in `image_buffer` and on the TFT.

When a user presses "Sample," the function `drawPepper()` is called. The function `drawPepper()` copies a popular image processing sample image of peppers to the screen. I hardcded the RGB565 pixel values of this image in the PSoC's flash memory (note the PSoC has a large 128 kB flash memory) by writing it as a `const` at the beginning of `main.c`. The 14,400 `uint16` pixel values defined there were precomputed from a 120x120 version of the peppers image in MATLAB (see **Appendix 2D** for the MATLAB code). The function `drawPepper()` copies this 120x120 image into the `image_buffer` and then displays it by calling `drawType1Image()`. The function `drawType1Image()` writes each pixel value twice and doubles each row, essentially scaling the image 200% to 240x240 when it is displayed on the TFT screen.

3.3. Image Processing Functions: Gradient, Blur, and Invert

Once the user has selected one of the options to display an image (and fill `image_buffer` with their selected, scaled-down data), they can select one or more image processing functions to apply to that data. Below I will describe the three image processing functions I created: gradient, blur, and inversion.

The inversion function is the simplest operation. When a user presses "Invert," `invertImage()` is called. The function simply performs a bitwise inversion of every byte in `image_buffer` (i.e. NOTs every bit of every byte in `image_buffer`). This simultaneous inverts the RGB components of each pixel, creating a photo-negative effect.

When a user presses "Blur" or "Gradient," `main.c` will call `blurImage()` or `gradientImage()`, respectively. Both of these functions call `convolve()`, with a different argument to specify which 3x3 kernel `image_buffer` will be convolved with (Gaussian blur for argument `0`, Sobel gradient kernels for argument `1`). In the `convolve()` function, for each pixel in the image, I extract the 3x3 region centered around that pixel. (In edge cases, I fill the underflowing or overflowing regions with the current pixel value). After I calculate the new value for each pixel using the kernel, I cannot simply replace the current pixel value in the buffer with the result. If I did this, the neighboring pixels will use affected pixel values in the next calculation instead of the original pixel values and produce an incorrect result. To account for this, I made a small 720-byte `row_buffer` to hold the convolved result of three rows in the

image at a time. Using this small buffer, the newly computed pixel values could be written back to the `image_buffer` with a two-row delay to ensure the convolution was done correctly while minimizing memory usage to stay within the PSoC limits. **Figure 10** shows how `row_buffer` and `image_buffer` are updated with respect to the current pixel being processed.

To blur an image, I use the 3x3 Gaussian blur kernel $G = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$ (from **[5]**). The blur kernel must be applied to the RGB components of each pixel separately to maintain a proper color image. To take the gradient of the image, I first converted all the pixels to greyscale by averaging their RGB components. Then, I convolved the greyscale image with two different 3x3 kernels $G_x = \begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix}$ and $G_y = \begin{bmatrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix}$ (from **[1]**). I take the magnitude of the result of the convolution in *x* and the result of the convolution in *y* $\sqrt{{G_x}^2 + {G_y}^2}$ and assign it to the current pixel.

Because all these functions just update the contents of `image_buffer`, I could apply them sequentially, as each will just apply the new function to the previous contents of `image_buffer`. The current state of the `image_buffer` can be displayed at any time using one of two functions: `drawType0Image()` or `drawType1Image()`. A type 0 image describes an image captured by the camera, so it would be stored as 80x240 in the image buffer and each pixel needed to be written three times to display it to the TFT screen. A type 1 image describes the sample peppers image, which is stored as 120x120 in the `image_buffer`. So, I needed to write each pixel two times and each row two times to scale it up to 240x240. I called `drawType0Image()` or `drawType1Image()` after each image processing function as called to display the resulting change to `image_buffer` on the TFT screen.

3.4. Menu Selection

I created the GUI menu by polling the potentiometer and button actions using an infinite loop in the main function. Similar to Exercise 4 of Kovid Lab 2, I divided the ADC output into six ranges for the six menu entries. If the user rotated the potentiometer into a different range than the previous state, my code clears the previous menu entry and highlights the new entry, using the TFT emWin graphics library. I read the status of `Button_Latch` to determine if the button had been pressed and, if so, which function I should call. After each call was completed, I cleared `Button_Latch` by setting `Button_Reset` high then low. This effectively avoied any problem with detecting double presses.

4. RESULTS & DEMO

I put my final setup through a series of tests to display its full functionality. These demos can be seen in **Figures 11-16** in **Appendix 1.**

In **Figure 11,** using "Live" mode and my sister's help pressing the button at the right moment, I captured a photo holding a sign that says "6.115 ROCKS!" I then clicked "Gradient" to calculate the gradient of the image, in an attempt to outline the edges of the text on the sign. Then, I inverted the gradient output simply by clicking "Invert" right after my processed image appeared on the display. This demo shows how I am able to sequentially apply my image processing filters. It also shows how I can use the "Live" mode to capture the perfect photo, rather than getting an immediate result from "Take Pic."

In **Figure 12**, I made my dad pick up my dog so I could take a picture and invert him so he'd be a "Zombie Dog" (we must turn to desperate forms of entertainment in quarantine). "Live" mode also helped here so I could see a preview and make sure my dog was centered in the frame. After a click of the "Invert" button, my dog was immediately transformed.

In **Figure 13**, similar to **Figure 11**, I consecutively applied the gradient and inversion to a picture of me in my Burton Conner floor shirt. You can see the clear outline of the bomber plane and "BOMBERS" letters on my shirt. In addition, in the original image, one can notice how nicely the orange color came out.

In **Figure 14**, I capture a selfie in "Take Pic" mode. I reached out and pressed the button for an immediate result. Then, I clicked "Gradient" so I could get a cool outline of my facial features.

In **Figure 15**, I am again holding the "6.115 ROCKS!" sign, but this time I click "Blur" three times in sequence after I take the picture. You can see how each time the image gets incrementally blurrier.

In **Figure 16**, I show each of my image processing operations performed independently on my sample image of peppers. The top left shows the original high-quality sample image. The top right shows the gradient. After I calculated the gradient, I clicked "Sample" again to return to the original. Then, I applied a blur, shown in the bottom left. I "reset" the image again and then inverted it, as can be seen in the bottom right.

I also performed some rudimentary analyses of the speed of each operation with my phone timer. I measured the amount of time it took to display 10 frames in "Live" mode and found that it averaged about 0.87 seconds per frame. I also found that "Take Pic" and "Sample" both take less than 1 second from button press to display. (I assume "Take Pic" must take ≤ 0.87 seconds since `live()` just called `captureImage()` over and over again). "Gradient" takes the longest (~2 seconds from button click to full render), which is likely because it uses a square root function (a notoriously expensive operation) to take the magnitude of the convolved results with G_x and G_y. "Invert" processes almost immediately, and "Blur" takes only slightly longer; but, both complete in less than one second.

Videos of the full operation of my GUI can be found here – https://youtu.be/mPehnXVb3rM and https://youtu.be/pnxzISbvqC0 – with complete demonstrations of the "Live" mode feature of the setup.

4.1. Future Improvements

While my final project was a success, I still envision several ways in which it could be improved given more time. Likely the easiest future step could be removing the two columns of

invalid data that are read at the beginning of this frame. These two columns are probably most easily spotted in the inverted image in **Figure 12**, but otherwise are very easy to overlook. These two columns of data actually happen to be the last two columns of data captured in the previous frame. I could recover these two columns and shift the image so it would display perfectly in the future.

Additionally, I could work on fixing the aspect ratio of the camera. You may notice that the images on my TFT screen suffer from a slight vertical stretch. It is possible that I could improve this in software by further scaling the image. More effectively, however, I could spend many more hours studying the ins and outs of the camera controller registers and finally be able to initialize the camera with the optimal size, scale, and aspect ratio.

REFERENCE LINKS

1. https://en.wikipedia.org/wiki/Sobel_operator
2. https://forum.pjrc.com/archive/index.php/t-30559.html
3. https://docs.microsoft.com/en-us/windows/win32/directshow/working-with-16-bit-rgb
4. https://github.com/westonb/OV7670-Verilog/blob/master/src/OV7670_config_rom.v
5. http://alumni.media.mit.edu/~maov/classes/vision09/lect/09_Image_Filtering_Edge_Detection_09.pdf

APPENDIX 1: FIGURES

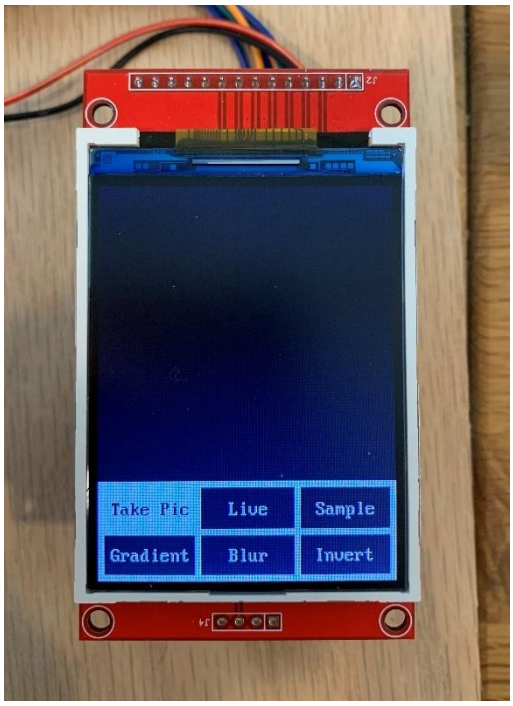


Figure 1: Initial Menu Dsiplay

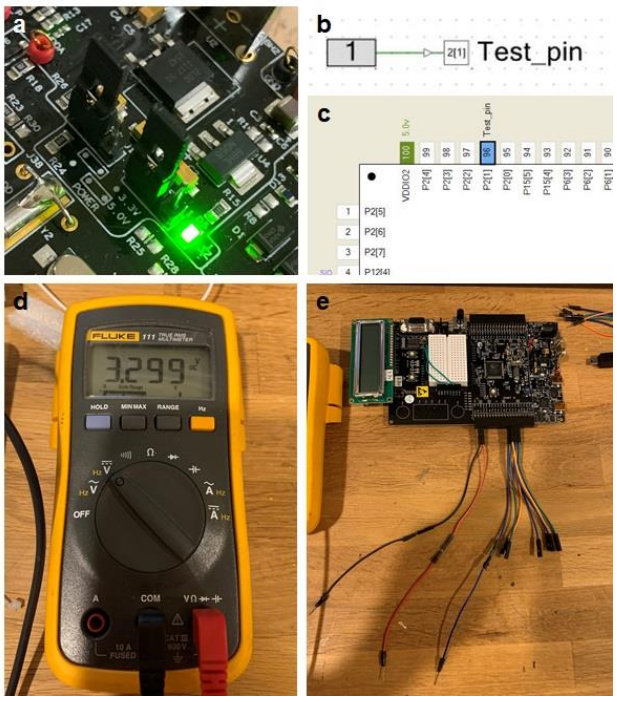


Figure 2: Verifying 3.3 V GPIO on the PSoC 5LP "Big Board"

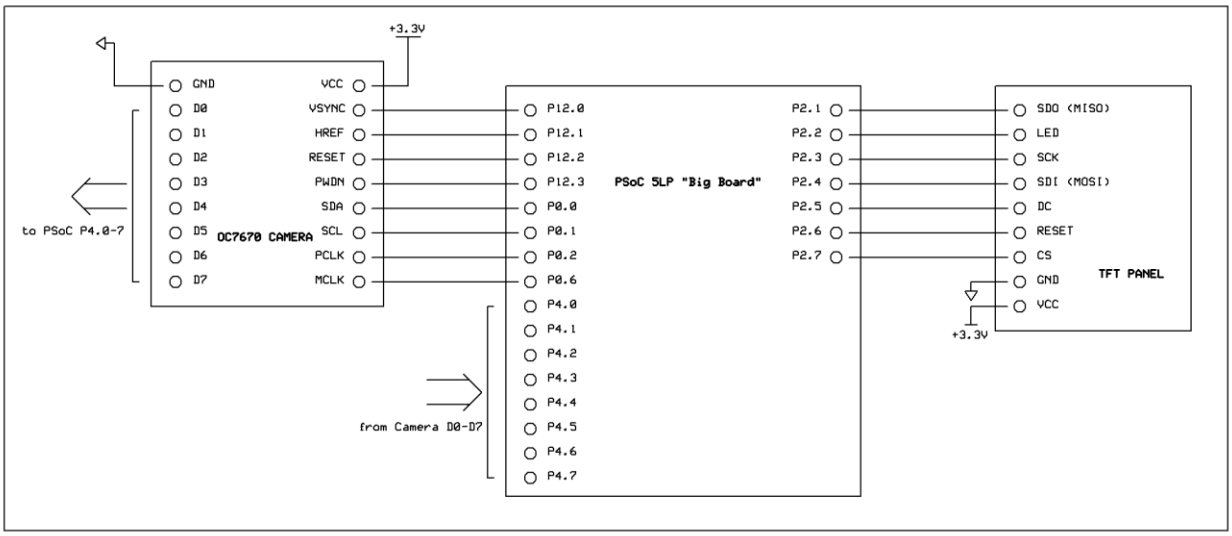


Figure 3: Final Schematic

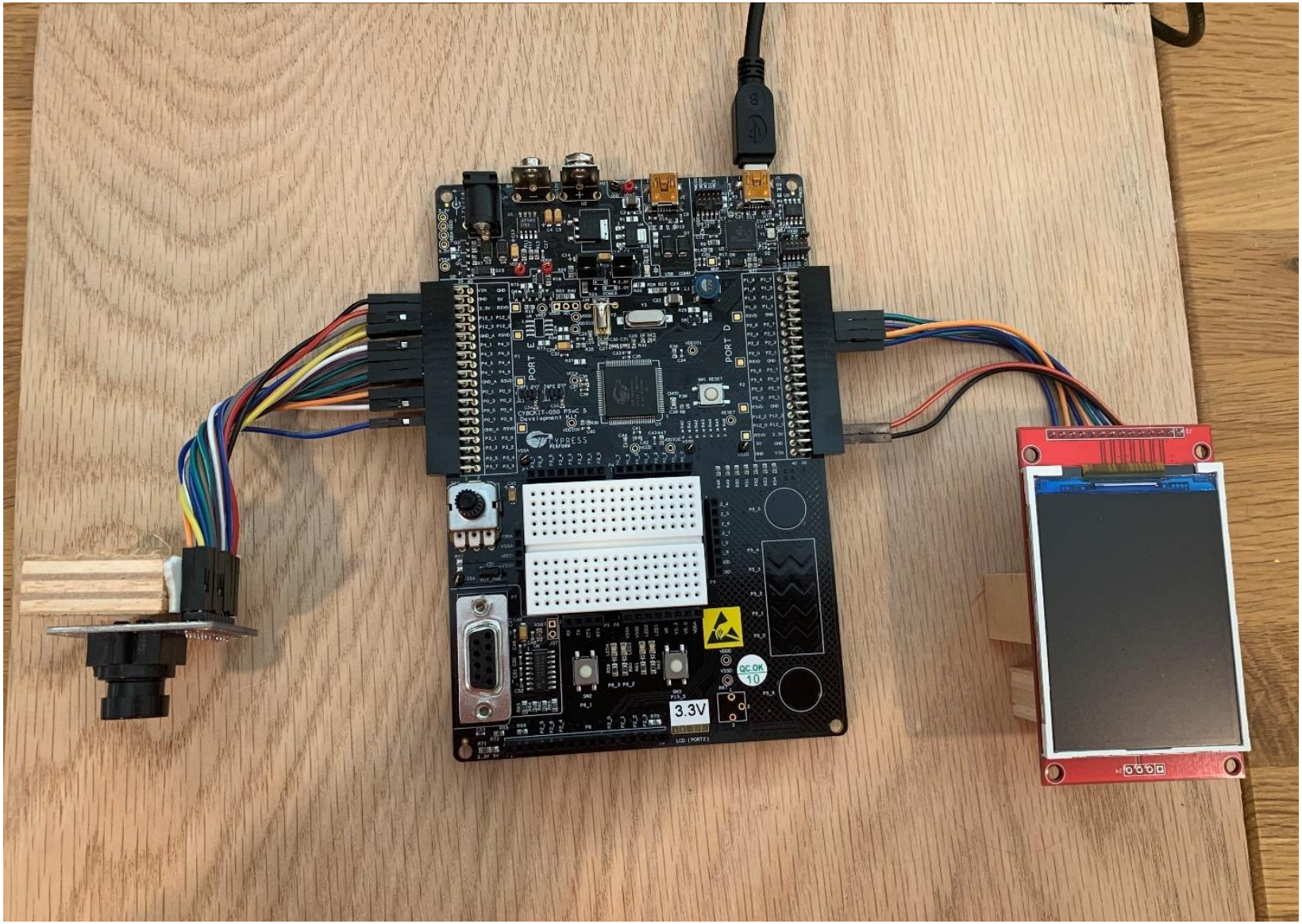


Figure 4: Final Setup

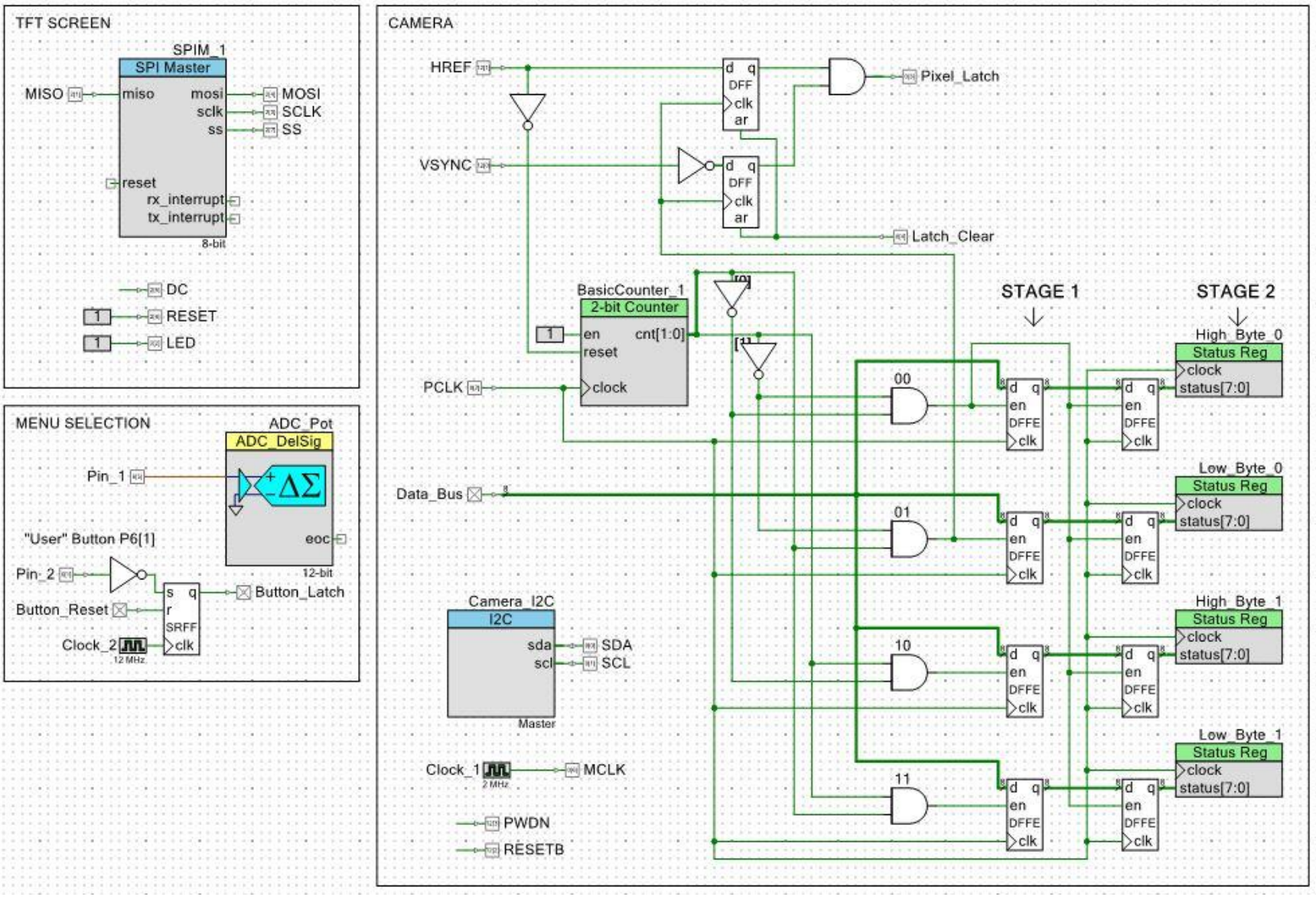
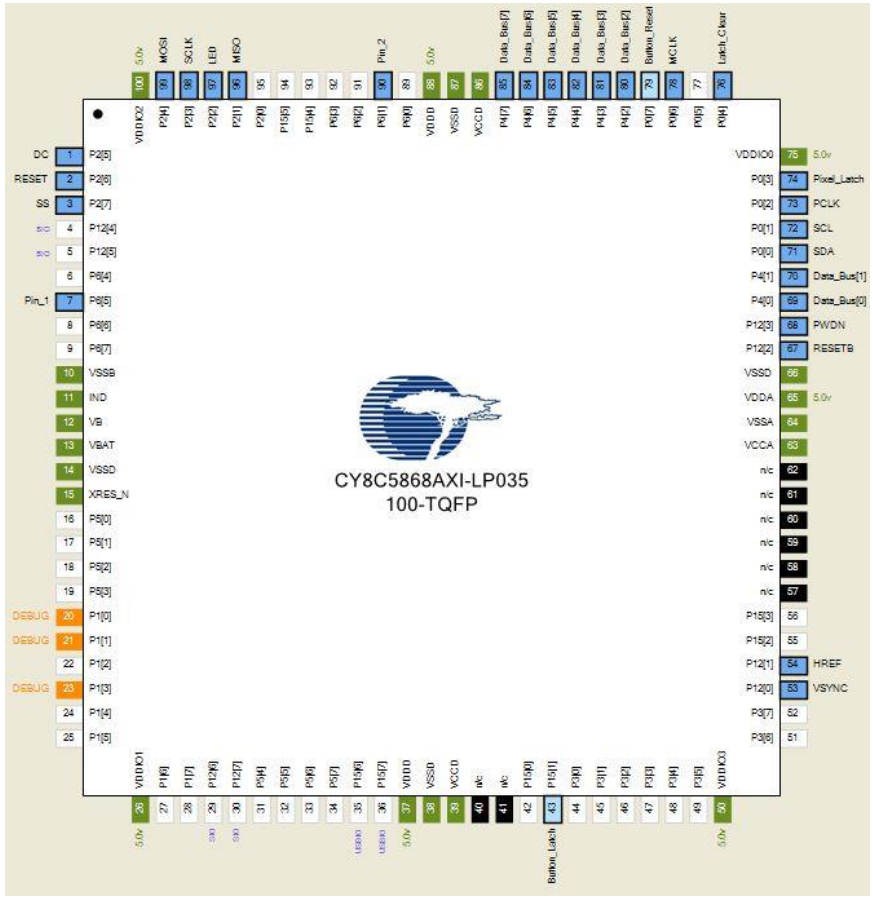


Figure 5: "Top Design" Schematic in PSoC Creator



	Name	Port	Pin	Lock
	\Data_Bus[7:0]\	P4[7:0]	69,70,80…85	☑
	Button_Latch	P15[1]	43	☐
	Button_Reset	P0[7]	79	☐
	DC	P2[5]	1	☑
	HREF	P12[1]	54	☑
	Latch_Clear	P0[4]	76	☑
	LED	P2[2]	97	☑
	MCLK	P0[6]	78	☑
	MISO	P2[1]	96	☑
	MOSI	P2[4]	99	☑
	PCLK	P0[2]	73	☑
	Pin_1	P6[5]	7	☑
	Pin_2	P6[1]	90	☑
	Pixel_Latch	P0[3]	74	☑
	PWDN	P12[3]	68	☑
	RESET	P2[6]	2	☑
	RESETB	P12[2]	67	☑
	SCL	P0[1]	72	☑
	SCLK	P2[3]	98	☑
	SDA	P0[0]	71	☑
	SS	P2[7]	3	☑
	VSYNC	P12[0]	53	☑

Figure 6: Final Pin Assignemnts in PSoC Creator

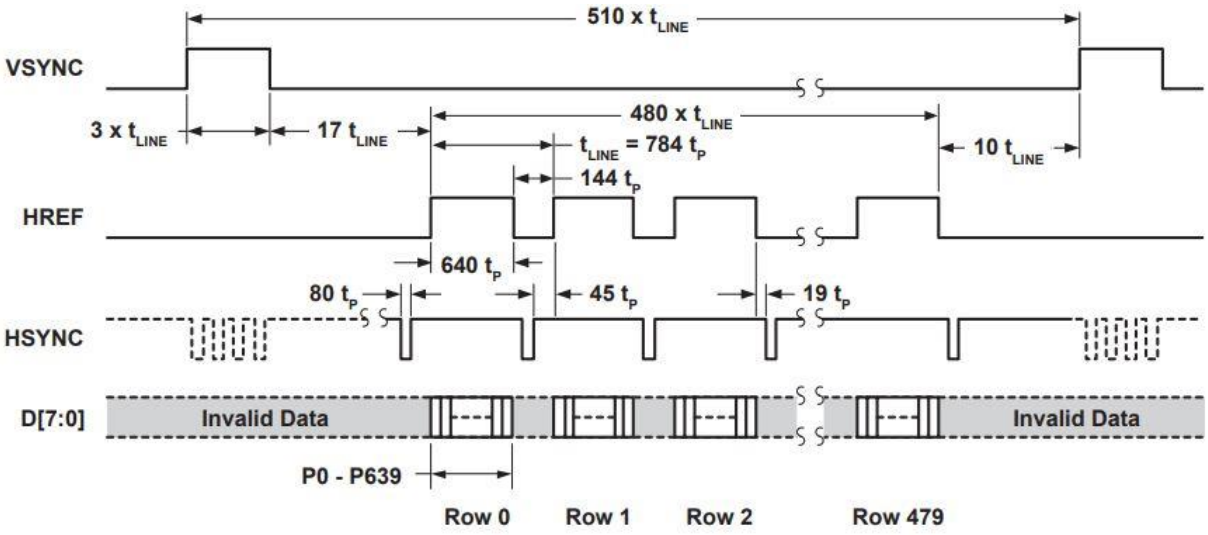


Figure 7: VGA Timing Diagram from OV7670 Datasheet

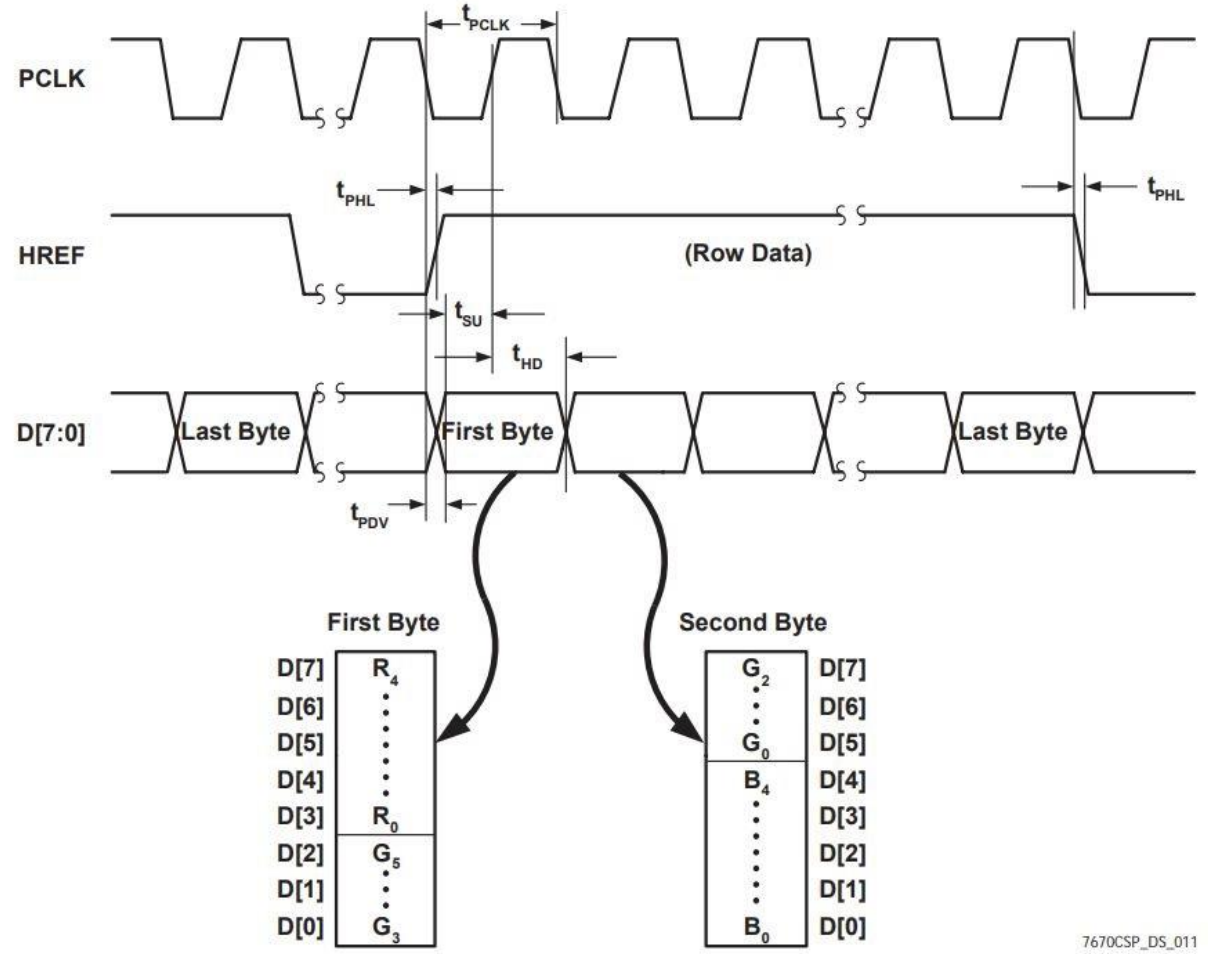


Figure 8: RGB565 Timing Diagram from OV7670 Datasheet

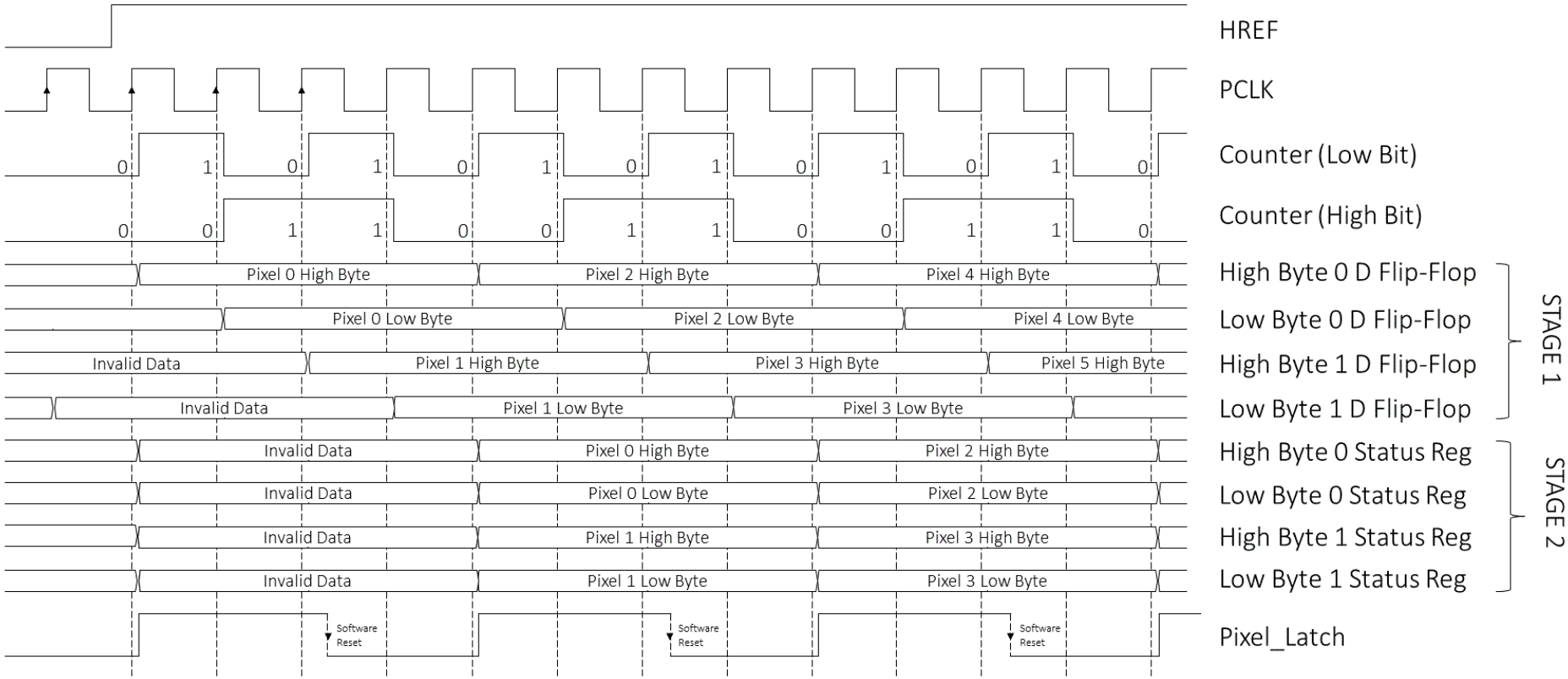


Figure 9: Timing Diagram

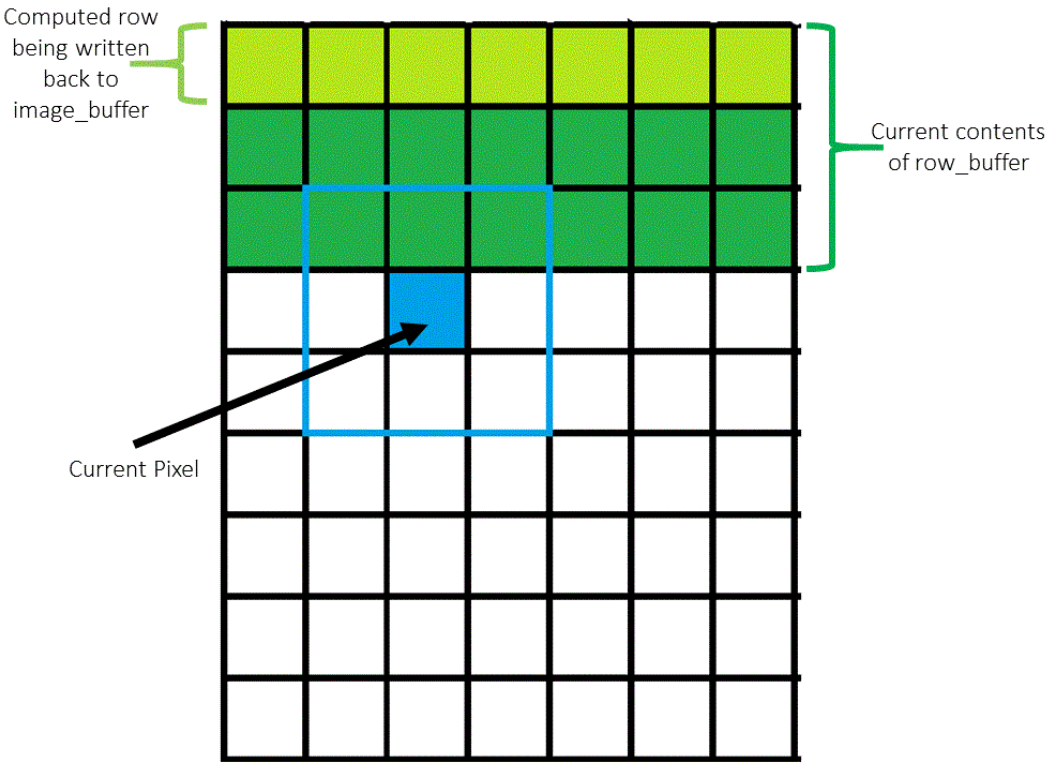


Figure 10: Shows how `row_buffer` and `image_buffer` are updated with respect to the current pixel being processed in `main.c`

Figure 11: The essential test! Is it even a 6.115 project if this isn't my sample image? – original, gradient, inverted (applied sequentially)

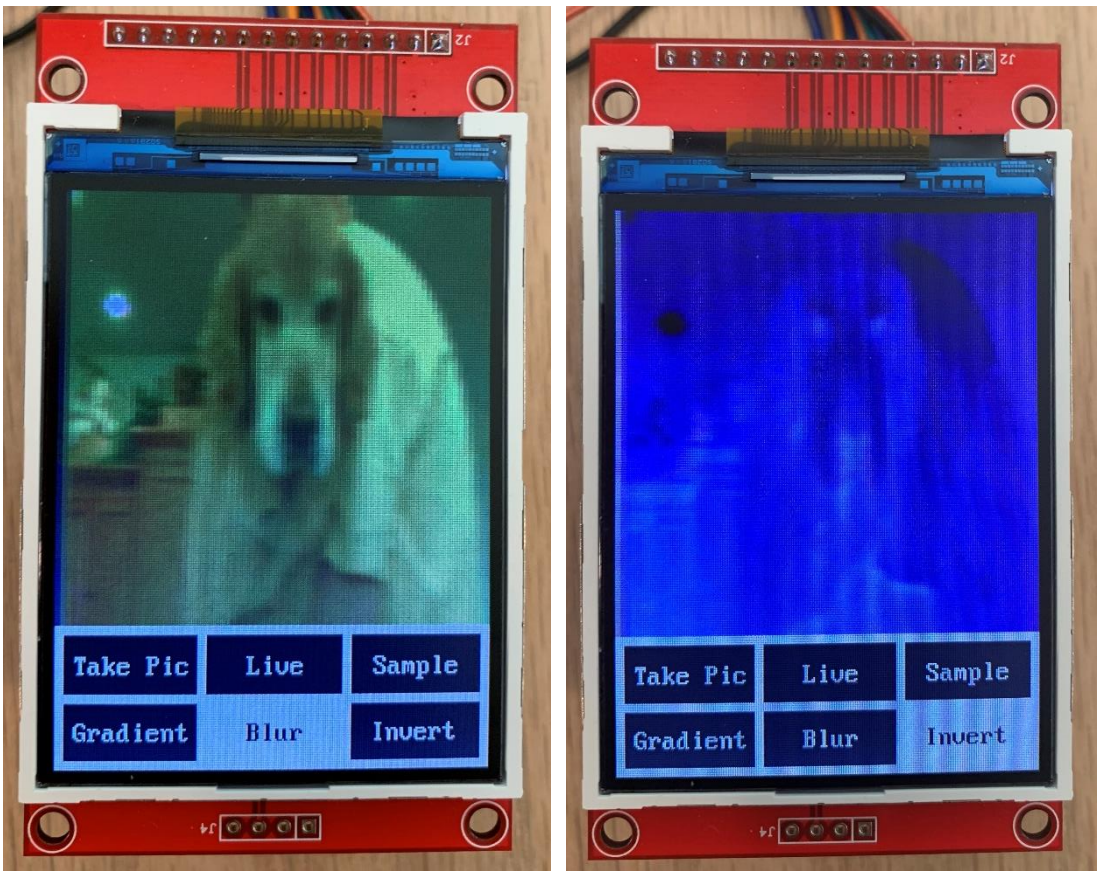


Figure 12: My dad holding my dog – original and inverted

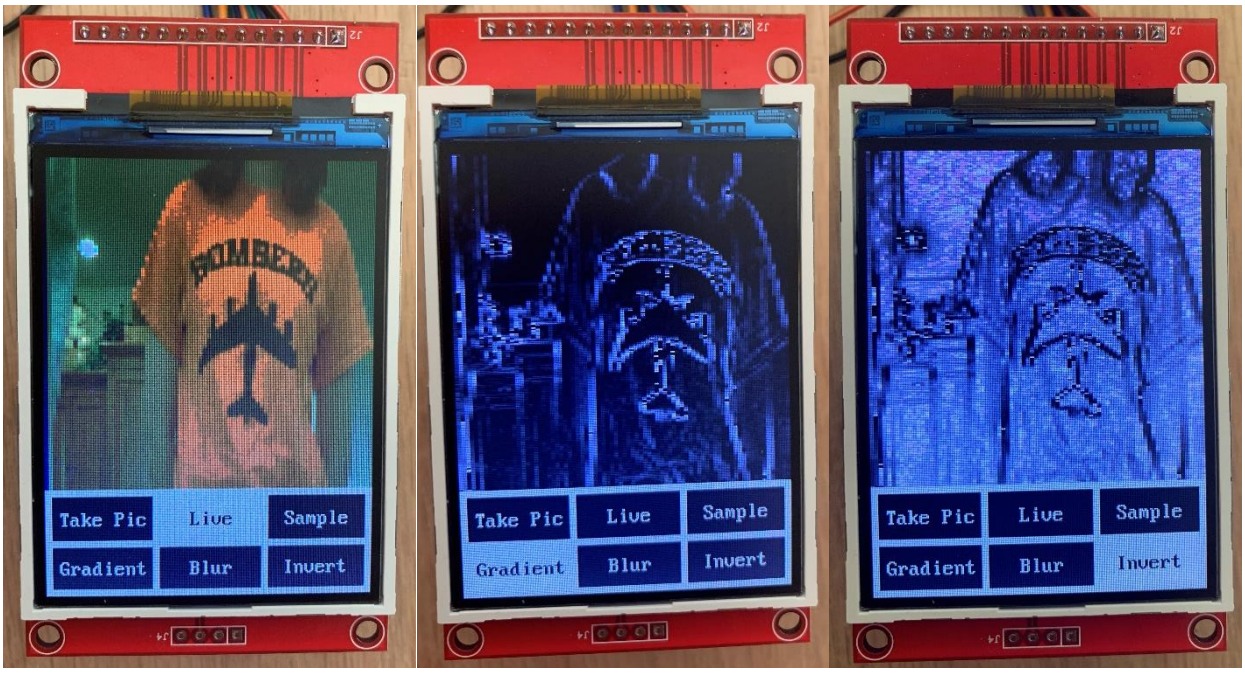


Figure 13: Me in my bomber shirt – original, gradient, inverted (applied sequentially)

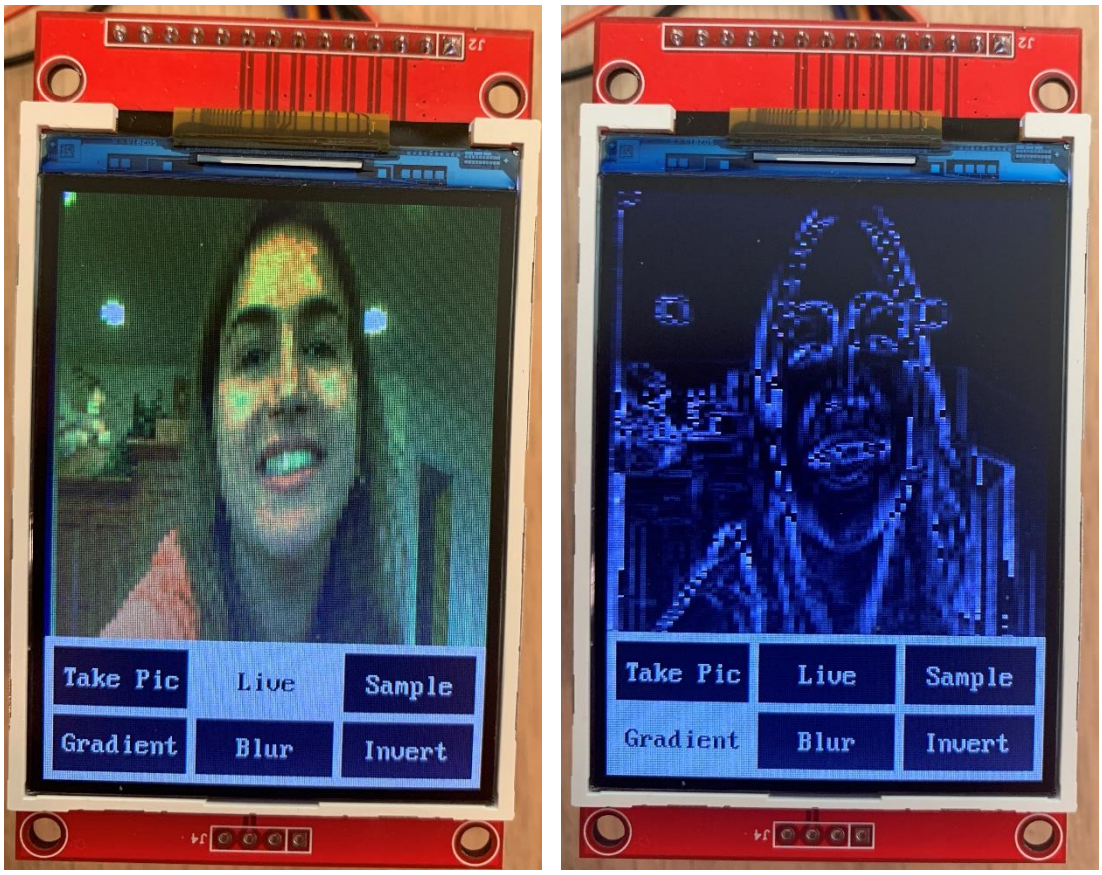


Figure 14: A selfie! And proof my project was not completed by a robot :) – original and gradient

Figure 15: 6.115 Rocks! – original, blur applied 3 times sequentially

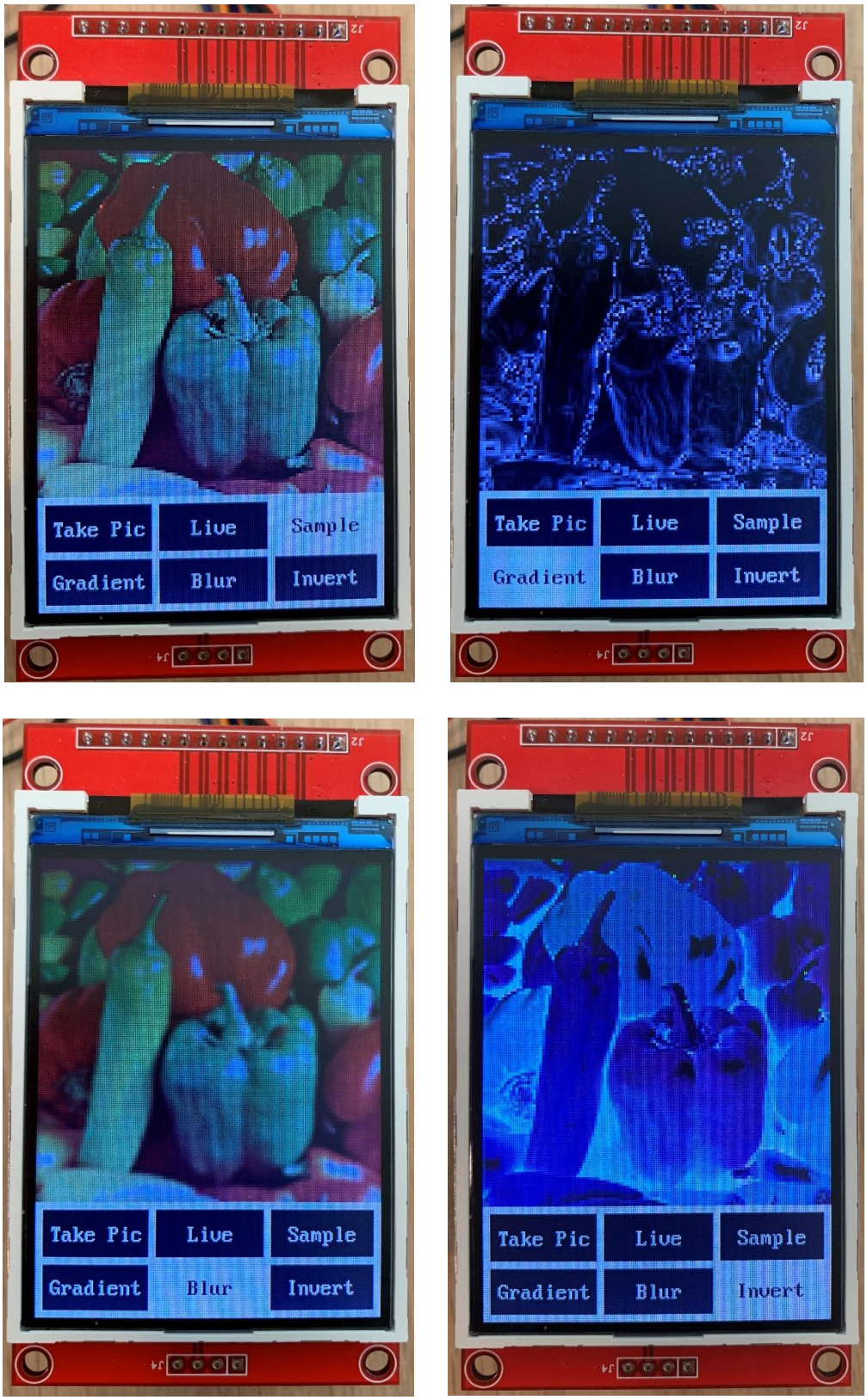


Figure 16: Sample peppers image – original, gradient, blur, invert (each applied individually, image reset before each operation)

APPENDIX 2: CODE

APPENDIX 2A, MAIN.C

NOTE: I used methods from https://docs.microsoft.com/en-us/windows/win32/directshow/working-with-16-bit-rgb to convert between RGB565 and RGB88.

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include <project.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "GUI.h"
#include "tft.h"
#include "camera.h"

// ---------
// CONSTANTS
// ---------
// RGB565 sample image constant array
const uint16 pepper_img[] = {41383,45577,45578,47595,49610,49643, ... (14400 uint16 values) ...,
52553,55230,48527,48855,48854,46708};

// ----------------
// GLOBAL VARIABLES
// ----------------
// Camera variables
uint16 num_pixels_in_buffer = 80*240;  // number of uint16 pixel values in a buffer
uint16 num_bytes_in_buffer = 80*2*240;
uint8 image_buffer[80*2*240];          // create global image buffer
uint16 count = 0;
uint16 total_count = 0;
// Menu text positions
int take_pic_x = 10;
int live_x = 105;
int sample_x = 175;
int gradient_x = 10;
int blur_x = 105;
int invert_x = 174;
int first_row_y = 255;
int second_row_y = 292;
// Flags and counters
int lastButton;                       // tracks last button selection
int live_frame = 0;
int img_flag = -1;                    // determines whether we are processing image
                                      // captured from camera (0) or sample image (1)
// Kernels and buffer for image processing functions
uint8 row_buffer[120*3*2];
int gaussian_blur_kernel_3x3[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};
int gaussian_blur_divider = 16;
int G_x[9] = {1, 0, -1, 2, 0, -2, 1, 0, -1};
int G_y[9] = {1, 2, 1, 0, 0, 0, -1, -2, -1};
```

```
// --------------------
// FUNCTION DEFINITIONS
// --------------------
void captureImage(void);
void invertImage(void);
void convolve(int);
void blurImage(void);
void gradientImage(void);
void live(void);
// Menu functions
void initMenu(void);
void resetButton(int);
void writeWhiteButton(uint16, uint16);
// Sample images
void drawPepper(void);
// Displaying images
void drawType0Image(int);
void drawType1Image(void);

int main() {
    CyGlobalIntEnable;                  // Enable global interrupts
    SPIM_1_Start();                     // Initialize SPIM component

    uint16 n;                           // Initialize buffer with 0s
    for (n = 0; n < num_pixels_in_buffer; n++) {
        image_buffer[n] = (uint8)0;
    }
    cameraStart();                      // Initialize ArduCAM
    CyDelay(1000);

    tftStart();                         // Initialize TFT Screen
    GUI_Init();                         // Initilize graphics library
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16);

    ADC_Pot_Start();                    // Start the ADC_DelSig_1
    ADC_Pot_StartConvert();             // Start the ADC_DelSig_1 conversion

    initMenu();                         // Draw menu


    uint16 adcResult = 0;
    lastButton = -1;
    for(;;) {

        if (ADC_Pot_IsEndConversion(ADC_Pot_WAIT_FOR_RESULT)) {
            int flag = 0;
            if (Button_Latch_Read()) { flag = 1; }
            adcResult = ADC_Pot_GetResult16();    // Read the ADC and assign the value adcResult

            if (adcResult & 0x8000) { adcResult = 0; }  // Ignore negative ADC results
            else if (adcResult >= 0xfff) { adcResult = 0xfff; }     // Ignore high ADC results

            // Select one of 6 menu options and display
            if (adcResult < 0x2AA) {                                // TAKE PIC
                // Reset the last button
                if ((lastButton != -1) & (lastButton != 0)) {
                    resetButton(lastButton);
                }
                // Highlight button
                if (lastButton != 0) {
                    writeWhiteButton(0,240);
                    GUI_SetTextMode(GUI_TM_REV);
                    GUI_DispStringAt("Take Pic", take_pic_x, first_row_y);
                }
                // New last button
                lastButton = 0;
                // Check if button pressed
                if (flag) {
                    resetButton(0);
                    // TAKE PICTURE
```

```
        // reset live_frame so tft screen is reinitialized
        live_frame = 0;
        captureImage();
        // Reset button
        Button_Reset_Write(1);
        Button_Reset_Write(0);
        lastButton = -1;
    }
} else if (adcResult >= 0x2AA && adcResult < 0x554) {   // LIVE
    // Reset the last button
    if ((lastButton != -1) & (lastButton != 1)) {
        resetButton(lastButton);
    }
    // Highlight button
    if (lastButton != 1) {
        writeWhiteButton(80,240);
        GUI_SetTextMode(GUI_TM_REV);
        GUI_DispStringAt("Live", live_x, first_row_y);
    }
    // New last button
    lastButton = 1;
    // Check if button pressed
    if (flag) {
        resetButton(1);
        // LIVE
        live();
        // Reset button
        Button_Reset_Write(1);
        Button_Reset_Write(0);
        lastButton = -1;
    }
} else if (adcResult >= 0x554 && adcResult < 0x7FE) {   // SAMPLE
    // Reset the last button
    if ((lastButton != -1) & (lastButton != 2)) {
        resetButton(lastButton);
    }
    // Highlight button
    if (lastButton != 2) {
        writeWhiteButton(160,240);
        GUI_SetTextMode(GUI_TM_REV);
        GUI_DispStringAt("Sample", sample_x, first_row_y);
    }
    // New last button
    lastButton = 2;
    // Check if button pressed
    if (flag) {
        resetButton(2);
        // Draw sample image
        drawPepper();
        // Reset button
        Button_Reset_Write(1);
        Button_Reset_Write(0);
        lastButton = -1;
    }
} else if (adcResult >= 0x7FE && adcResult < 0xAA8) {   // GRADIENT
    // Reset the last button
    if ((lastButton != -1) & (lastButton != 3)) {
        resetButton(lastButton);
    }
    // Highlight button
    if (lastButton != 3) {
        writeWhiteButton(0,280);
        GUI_SetTextMode(GUI_TM_REV);
        GUI_DispStringAt("Gradient", gradient_x, second_row_y);
    }
    // New last button
    lastButton = 3;
    // Check if button pressed
    if (flag) {
        resetButton(3);
        // GRADIENT
```

```
                    gradientImage();
                    // Reset button
                    Button_Reset_Write(1);
                    Button_Reset_Write(0);
                    lastButton = -1;
                }
            } else if (adcResult >= 0xAA8 && adcResult < 0xD52) {   // BLUR
                // Reset the last button
                if ((lastButton != -1) & (lastButton != 4)) {
                    resetButton(lastButton);
                }
                // Highlight button
                if (lastButton != 4) {
                    writeWhiteButton(80,280);
                    GUI_SetTextMode(GUI_TM_REV);
                    GUI_DispStringAt("Blur", blur_x, second_row_y);
                }
                // New last button
                lastButton = 4;
                // Check if button pressed
                if (flag) {
                    resetButton(4);
                    // BLUR
                    blurImage();
                    // Reset button
                    Button_Reset_Write(1);
                    Button_Reset_Write(0);
                    lastButton = -1;
                }
            } else {                                                // INVERT
                // Reset the last button
                if ((lastButton != -1) & (lastButton != 5)) {
                    resetButton(lastButton);
                }
                // Highlight button
                if (lastButton != 5) {
                    writeWhiteButton(160,280);
                    GUI_SetTextMode(GUI_TM_REV);
                    GUI_DispStringAt("Invert", invert_x, second_row_y);
                }
                // New last button
                lastButton = 5;
                // Check if button pressed
                if (flag) {
                    resetButton(5);
                    // INVERT
                    invertImage();
                    // Reset button
                    Button_Reset_Write(1);
                    Button_Reset_Write(0);
                    lastButton = -1;
                }
            }
        }
    }
}

void live() {
    live_frame = 0;
    lastButton = -1;

    uint i = 0;
    for (;;) {
        // Reset button
        Button_Reset_Write(1);
        Button_Reset_Write(0);

        captureImage();

        // If we've captured at least 2 frames,
        // check if we've pressed the button to stop
```

```
        // livestream
        if ((i > 1) && Button_Latch_Read()) {
            CyDelay(500);
            break;
        }

        // Stop incrementing i so it doesn't overflow
        if (i < 10) { i++; }
    }
}

void captureImage() {
    int startFrame = 0;
    uint16 row_count = 0;
    for(;;) {

        // Check if we have started capturing a new image
        if (startFrame == 0) {
            if (VSYNC_Read()) {
                startFrame = 1;
                Latch_Clear_Write(1);           // reset pixel latching system
                Latch_Clear_Write(0);
            }
        }

        // Check if we have exceeded buffer size
        if (count >= num_bytes_in_buffer) {

            // DRAW CURRENT IMAGE BUFFER

            startFrame = 0;                     // reset startFrame
            img_flag = 0;                       // set image type

            drawType0Image(live_frame);

            count = 0;                          // clear counter and break from loop
            total_count = 0;
            live_frame++;
            break;
        } else if (startFrame && Pixel_Latch_Read()) { // Pixel_Latch is 1, write to image buffer

            // READ IN PIXELS
            if (((total_count%160) == 0) && (total_count!=0)) {
                if (row_count < 2) { row_count++; }         // Only count 2/3 rows
                else { row_count = 0; }
            }
            if (row_count < 2) {
                image_buffer[count] = High_Byte_0_Read();  // Read in 2 pixels (4 bytes) at once
                image_buffer[count+1] = Low_Byte_0_Read(); // from latching system
                image_buffer[count+2] = High_Byte_1_Read();
                image_buffer[count+3] = Low_Byte_1_Read();
                count += 4;                                 // Increment count
            }
            Latch_Clear_Write(1);                           // Clear latch
            Latch_Clear_Write(0);
            total_count += 4;
        }
    }
}

void invertImage() {
    // Perform bitwise inversion on every byte in image buffer
    int i;
    for (i = 0; i < num_bytes_in_buffer; i++) {
        image_buffer[i] = ~image_buffer[i];
    }
    // Draw image buffer based on whether this is a sample
    // or a captured image
    if (img_flag == 0) {
        drawType0Image(0);
    } else if (img_flag == 1) {
```

```
        drawType1Image();
    }
}

void blurImage() {
    convolve(0);
}

void gradientImage() {
    convolve(1);
}

void convolve(int kernel_flag) {
    // KERNEL 0 -- GAUSSIAN BLUR
    // KERNEL 1 -- SOBEL EDGE DETECTION (GRADIENT)

    // Find dimensions of image in image buffer
    uint8 num_cols;
    uint8 num_rows;
    if (img_flag == 0) {        // from camera
        num_cols = 80;
        num_rows = 240;
    } else if (img_flag == 1) { // sample img
        num_cols = 120;
        num_rows = 120;
    } else {
        return;
    }

    uint8 i;
    uint8 j;
    int x;
    int y;

    uint16 index;
    uint16 pixel;
    uint8 red;
    uint8 green;
    uint8 blue;
    uint8 neighbor_red[9];
    uint8 neighbor_blue[9];
    uint8 neighbor_green[9];
    int row_ind;
    int col_ind;
    uint count;
    uint16 neighbor_index;
    uint8 sum_red;
    uint8 sum_green;
    uint8 sum_blue;
    uint8 sobel_mag;
    int gx_mag;
    int gy_mag;
    uint16 buffer_count = 0;
    uint16 shift_count;
    uint8 pixel_mag;

    uint16 red_mask = 0xF800;
    uint16 green_mask = 0x07E0;
    uint16 blue_mask = 0x001F;

    // GRAB NEIGHBORING REGION & PERFORM CONVOLUTION
    for (i = 0; i < num_rows; i++) {

        // FILL UP & RESET ROW_BUFFER AS WE GO
        if (buffer_count >= 3*num_cols) {
            // copy first row into image buffer
            for (j = 0; j < num_cols; j++) {
                index = (i-3)*num_cols+j;
                image_buffer[2*index] = row_buffer[2*j];
                image_buffer[2*index+1] = row_buffer[2*j+1];
            }
```

```
            // shift rows in row_buffer up
            shift_count = 0;
            for (x = num_cols; x < 3*num_cols; x++) {
                row_buffer[2*shift_count] = row_buffer[2*x];
                row_buffer[2*shift_count+1] = row_buffer[2*x+1];
                shift_count++;
            }
            // reset counter to start rewriting 3rd row
            buffer_count = 2*num_cols;
        }

        // EXTRACT NEIGHBORING REGION
        for (j = 0; j < num_cols; j++) {
            index = i*num_cols+j;
            // extract uint16 pixel value from high byte and low byte
            pixel = ((image_buffer[2*index] << 8) & (0xFF00)) | (image_buffer[2*index+1] &
0X00FF);
            // convert from RGB565 to uint8 red, green, and blue values
            red = ((pixel & red_mask) >> 11) << 3;   // 5 bits << 3
            green = ((pixel & green_mask) >> 5) << 2;  // 6 bits << 2
            blue = (pixel & blue_mask) << 3;          // 5 bits << 3

            count = 0;
            for (x = -1; x <= 1; x++) {
                for (y = -1; y <= 1; y++) {
                    if ((x == 0) && (y == 0)) {
                        neighbor_red[count] = red;
                        neighbor_green[count] = green;
                        neighbor_blue[count] = blue;
                    } else {
                        row_ind = i+x;
                        col_ind = j+y;
                        // handle edge cases
                        if ((row_ind < 0) || (col_ind < 0) || (row_ind >= num_rows) || (col_ind
>= num_cols)) {
                            // if underflow or overflow, copy center pixel
                            neighbor_red[count] = red;
                            neighbor_green[count] = green;
                            neighbor_blue[count] = blue;
                        } else {
                            // not an edge case, calculate as normal
                            neighbor_index = row_ind*num_cols+col_ind;
                            pixel = ((image_buffer[2*neighbor_index] << 8) & (0xFF00)) |
(image_buffer[2*neighbor_index+1] & 0X00FF);
                            neighbor_red[count] = ((pixel & red_mask) >> 11) << 3; // 5 bits<<3
                            neighbor_green[count] = ((pixel & green_mask) >> 5) << 2;// 6 bits<<2
                            neighbor_blue[count] = (pixel & blue_mask) << 3; // 5 bits<<3
                        }
                    }
                    count++;
                }
            }

            // PERFORM CONVOLUTION
            sum_red = 0;
            sum_green = 0;
            sum_blue = 0;
            gx_mag = 0;
            gy_mag = 0;
            for (x = 0; x < 9; x++) {
                if (kernel_flag == 0) { // gaussian blur
                    sum_red +=
(uint8)(gaussian_blur_kernel_3x3[x]*neighbor_red[x]/gaussian_blur_divider);
                    sum_green +=
(uint8)(gaussian_blur_kernel_3x3[x]*neighbor_green[x]/gaussian_blur_divider);
                    sum_blue +=
(uint8)(gaussian_blur_kernel_3x3[x]*neighbor_blue[x]/gaussian_blur_divider);
                } else { // edge detection
                    pixel_mag = (uint8)((neighbor_red[x]+neighbor_green[x]+neighbor_blue[x])/3);
                    gx_mag += G_x[x]*(int)(pixel_mag);
                    gy_mag += G_y[x]*(int)(pixel_mag);
```

```
                }
            }
            if (kernel_flag == 1) {
                sobel_mag = (uint8)(sqrt((double)(gx_mag*gx_mag + gy_mag*gy_mag)));
                sum_red = sobel_mag;
                sum_green = sobel_mag;
                sum_blue = sobel_mag;
            }
            pixel = (((sum_red >> 3) & 0x1F) << 11) | (((sum_green >> 2) & 0x3F) << 5) |
((sum_blue >> 3) & 0x1F);
            row_buffer[2*buffer_count] = (pixel & 0xFF00) >> 8;
            row_buffer[2*buffer_count+1] = pixel & 0x00FF;
            buffer_count++;
        }
    }

    // write last two rows of row_buffer to image_buffer
    int n;
    int col_count;
    for (n = 0; n < 3; n++) {
        col_count = 0;
        for (x = n*num_cols; x < (n+1)*num_cols; x++) {
            index = (num_rows-(3-n))*num_cols+col_count;
            image_buffer[2*index] = row_buffer[2*x];
            image_buffer[2*index+1] = row_buffer[2*x+1];
            col_count++;
        }
    }

    if (img_flag == 0) {        // from camera
        drawType0Image(0);
    } else if (img_flag == 1) { // sample img
        drawType1Image();
    }
}

void resetButton(int button) {
    // Find button boundaries
    uint16 SC = 5;
    uint16 EC = 77;
    uint16 SP = 245;
    uint16 EP = 277;
    if (button == 0) {              // TAKE PIC
    } else if (button == 1) {       // LIVE
        SC = 83; EC = 157;
    } else if (button == 2) {       // SAMPLE
        SC = 163; EC = 234;
    } else if (button == 3) {       // GRADIENT
        SP = 283; EP = 314;
    } else if (button == 4) {       // BLUR
        SC = 83; EC = 157;
        SP = 283; EP = 314;
    } else {                        // INVERT
        SC = 163; EC = 234;
        SP = 283; EP = 314;
    }

    // Write black pixels to button region
    write8_a0(0x2A);                // send Column Address Set command
    write8_a1(SC >> 8);                // set SC[15:0]
    write8_a1(SC & 0x00FF);
    write8_a1(EC >> 8);                // set EC[15:0]
    write8_a1(EC & 0x00FF);
    write8_a0(0x2B);                // send Page Address Set command
    write8_a1(SP >> 8);                // set SP[15:0]
    write8_a1(SP & 0x00FF);
    write8_a1(EP >> 8);                // set EP[15:0]
    write8_a1(EP & 0x00FF);
    write8_a0(0x2C);                // send Memory Write command
    uint16 i;
    for (i=0; i<((EC-SC+1)*(EP-SP+1)); i++) {   // Fill button with black
```

```
        write8_a1(0x00);
            write8_a1(0x00);
    }
    write8_a0(0x00);                            // Send NOP command to end writing process

    // Write button text
    GUI_SetTextMode(GUI_TM_NORMAL);
    if (button == 0) {                          // TAKE PIC
        GUI_DispStringAt("Take Pic", take_pic_x, first_row_y);
    } else if (button == 1) {                   // LIVE
        GUI_DispStringAt("Live", live_x, first_row_y);
    } else if (button == 2) {                   // SAMPLE
        GUI_DispStringAt("Sample", sample_x, first_row_y);
    } else if (button == 3) {                   // GRADIENT
        GUI_DispStringAt("Gradient", gradient_x, second_row_y);
    } else if (button == 4) {                   // BLUR
        GUI_DispStringAt("Blur", blur_x, second_row_y);
    } else {                                    // INVERT
        GUI_DispStringAt("Invert", invert_x, second_row_y);
    }
}

void writeWhiteButton(uint16 SC, uint16 SP) {
    uint16 EC = SC+79;
    uint16 EP = SP+39;
    write8_a0(0x2A);                  // send Column Address Set command
    write8_a1(SC >> 8);                 // set SC[15:0]
    write8_a1(SC & 0x00FF);
    write8_a1(EC >> 8);                 // set EC[15:0]
    write8_a1(EC & 0x00FF);
    write8_a0(0x2B);                  // send Page Address Set command
    write8_a1(SP >> 8);                 // set SP[15:0]
    write8_a1(SP & 0x00FF);
    write8_a1(EP >> 8);                 // set EP[15:0]
    write8_a1(EP & 0x00FF);
    write8_a0(0x2C);                    // send Memory Write command
    uint16 i;
    for (i=0; i<3200; i++)
       {
        write8_a1(0xFF);
               write8_a1(0xFF);
    }
    write8_a0(0x00);                    // send NOP command to end writing process
}

void initMenu() {
    // Set menu borders
    uint16 SC = 0;                      // Fill 240x80 space with 6 box table
    uint16 EC = 239;
    uint16 SP = 240;
    uint16 EP = 319;
    write8_a0(0x2A);                  // send Column Address Set command
    write8_a1(SC >> 8);                 // set SC[15:0]
    write8_a1(SC & 0x00FF);
    write8_a1(EC >> 8);                 // set EC[15:0]
    write8_a1(EC & 0x00FF);
    write8_a0(0x2B);                  // send Page Address Set command
    write8_a1(SP >> 8);                 // set SP[15:0]
    write8_a1(SP & 0x00FF);
    write8_a1(EP >> 8);                 // set EP[15:0]
    write8_a1(EP & 0x00FF);
    write8_a0(0x2C);                    // send Memory Write command

    // Draw "table"
    uint16 i;
    for (i=0; i<19200; i++) {
        uint16 y = (uint16)(floor((double)i/240));
        uint16 x = i-240*y;
        if ( (x<5) | ((x>=78)&(x<=82)) | ((x>=158)&(x<=162)) | (x>234) ) {
            // Write white
            write8_a1(0xFF);
```

```
            write8_a1(0xFF);
        } else if ( (y<5) | ((y>=38)&(y<=42)) | (y>74) ) {
            // Write white
            write8_a1(0xFF);
           write8_a1(0xFF);
        } else {
            // Write black
            write8_a1(0x00);
            write8_a1(0x00);
        }
    }
    write8_a0(0x00);                    // send NOP command to end writing process

    // Print menu options using graphics library
    GUI_DispStringAt("Take Pic", take_pic_x, first_row_y);
    GUI_DispStringAt("Live", live_x, first_row_y);
    GUI_DispStringAt("Sample", sample_x, first_row_y);
    GUI_DispStringAt("Gradient", gradient_x, second_row_y);
    GUI_DispStringAt("Blur", blur_x, second_row_y);
    GUI_DispStringAt("Invert", invert_x, second_row_y);
}

void drawType0Image(int frame) {
    // TYPE 0 IMAGE IS AN IMAGE CAPTURED BY THE CAMERA
    // stored in 80x240 in the image buffer

    // frame tells us whether this is the first "frame" captured
    // and if we should re-initialize the TFT display
    if (frame == 0) {                   // initialize the TFT display
        tftStart();
    }
    uint16 SC = 0;                  // define a 240x240 square with the top left corner at (0,0)
    uint16 EC = 239;
    uint16 SP = 0;
    uint16 EP = 239;
    write8_a0(0x2A);                // send Column Address Set command
    write8_a1(SC >> 8);             // set SC[15:0]
    write8_a1(SC & 0x00FF);
    write8_a1(EC >> 8);             // set EC[15:0]
    write8_a1(EC & 0x00FF);
    write8_a0(0x2B);                // send Page Address Set command
    write8_a1(SP >> 8);             // set SP[15:0]
    write8_a1(SP & 0x00FF);
    write8_a1(EP >> 8);             // set EP[15:0]
    write8_a1(EP & 0x00FF);
    write8_a0(0x2C);                // send Memory Write command

    uint16 i;
    for (i=0; i < num_pixels_in_buffer; i++)    // fill the square with the stored image
       {
        // write the pixel 3x to fix scalings
        uint8 hb = image_buffer[2*i];
        uint8 lb = image_buffer[2*i+1];
        write8_a1(hb);
        write8_a1(lb);
        write8_a1(hb);
        write8_a1(lb);
        write8_a1(hb);
        write8_a1(lb);
       }
    write8_a0(0x00);                    // send NOP command to end writing process
}

void drawType1Image() {
    // TYPE 1 IMAGE HAS THE FORMAT OF THE SAMPLE PEPPERS IMAGE
    // stored as 120x120 in immage buffer

    tftStart();             // initialize the TFT display
    uint16 SC = 0;          // define a 240x240 square at pixel (0, 0)
    uint16 EC = 239;        // EC < 240, EP < 320
    uint16 SP = 0;
```

```
    uint16 EP = 239;
    write8_a0(0x2A);         // send Column Address Set command
    write8_a1(SC >> 8);      // set SC[15:0]
    write8_a1(SC & 0x00FF);
    write8_a1(EC >> 8);      // set EC[15:0]
    write8_a1(EC & 0x00FF);
    write8_a0(0x2B);         // send Page Address Set command
    write8_a1(SP >> 8);      // set SP[15:0]
    write8_a1(SP & 0x00FF);
    write8_a1(EP >> 8);      // set EP[15:0]
    write8_a1(EP & 0x00FF);
    write8_a0(0x2C);         // send Memory Write command
    uint16 i;
    uint16 j;
    int n;

    // draw in pepper array
    for (i=0; i<120; i++) {   // fill the square with image
        for (n = 0; n<2; n++) {
            for (j=0; j<120; j++) {
                uint16 index = i*120+j;
                uint8 hb = image_buffer[2*index];
                uint8 lb = image_buffer[2*index+1];
                write8_a1(hb);
                write8_a1(lb);
                write8_a1(hb);
                write8_a1(lb);
            }
        }
    }
    write8_a0(0x00);                    // send NOP command to end writing process
}

void drawPepper() {
    img_flag = 1;
    // copy pepper into image buffer and mark flag
    int i; int j;
    for (i=0; i<120; i++) {
        for (j=0; j<120; j++) {
            uint16 index = i*120+j;
            uint16 pixel = pepper_img[index];
            image_buffer[2*index] = pixel >> 8;
            image_buffer[2*index+1] = pixel & 0x00FF;
        }
    }
    // display on tft screen
    drawType1Image();
}

/* [] END OF FILE */
```

APPENDIX 2B, CAMERA.C

NOTE: I obtained the configuration settings for the camera from
https://github.com/westonb/OV7670-Verilog/blob/master/src/OV7670_config_rom.v

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include "camera.h"
#include "GUI.h"
#include <stdlib.h>

//===============================================================
// cameraStart()
// this function must be called to initializes the camera
//===============================================================
void cameraStart(void)
{
    // PWDN - 0, RESET = 0 -- Hardware reset
    PWDN_Write(0);
    RESETB_Write(0);
    CyDelay(200); // Delay for changes to take effect

    // Life RESET pin to 1
    RESETB_Write(1);
    CyDelay(200);  // Delay for changes to take effect

    // Initialize I2C component
    Camera_I2C_Start();

    // Software reset for the camera
    Camera_I2C_MasterSendStart(0x21, 0);
    Camera_I2C_MasterWriteByte(0x12);
    Camera_I2C_MasterWriteByte(0x80);
    Camera_I2C_MasterSendStop();
    CyDelay(250); // Delay for changes to take effect

    // Write configuration settings from the internet
    writeBytes();

}

void writeBytes(void) {
    // Use camera configuration settinsgs for RGB565 and QVGA from
    // https://github.com/westonb/OV7670-Verilog/blob/master/src/OV7670_config_rom.v
    // with slight modification to send 0x13 to 0x3E to divide PCLK by 4
    uint8 RGB565_640_480[] = {0x12, 0x04, 0x11, 0x80, 0x0C, 0x00, 0x3E, 0x13, 0x04, 0x00, 0x40,
0xd0, 0x3a, 0x04, 0x14, 0x18, 0x4F, 0xB3, 0x50, 0xB3, 0x51, 0x00, 0x52, 0x3d, 0x53, 0xA7, 0x54,
0xE4, 0x58, 0x9E, 0x3D, 0xC0, 0x17, 0x14, 0x18, 0x02, 0x32, 0x80, 0x19, 0x03, 0x1A, 0x7B, 0x03,
0x0A, 0x0F, 0x41, 0x1E, 0x00, 0x33, 0x0B, 0x3C, 0x78, 0x69, 0x00, 0x74, 0x00, 0xB0, 0x84, 0xB1,
0x0c, 0xB2, 0x0e, 0xB3, 0x80, 0x70, 0x3a, 0x71, 0x35, 0x72, 0x11, 0x73, 0xf0, 0xa2, 0x02, 0x7a,
0x20, 0x7b, 0x10, 0x7c, 0x1e, 0x7d, 0x35, 0x7e, 0x5a, 0x7f, 0x69, 0x80, 0x76, 0x81, 0x80, 0x82,
0x88, 0x83, 0x8f, 0x84, 0x96, 0x85, 0xa3, 0x86, 0xaf, 0x87, 0xc4, 0x88, 0xd7, 0x89, 0xe8, 0x13,
0xe0, 0x00, 0x00, 0x10, 0x00, 0x0d, 0x40, 0x14, 0x18, 0xa5, 0x05, 0xab, 0x07, 0x24, 0x95, 0x25,
0x33, 0x26, 0xe3, 0x9f, 0x78, 0xa0, 0x68, 0xa1, 0x03, 0xa6, 0xd8, 0xa7, 0xd8, 0xa8, 0xf0, 0xa9,
0x90, 0xaa, 0x94, 0x13, 0xe5};
    int i;
    for (i = 0; i < (int)((sizeof RGB565_640_480)/2); i++) {
        Camera_I2C_MasterSendStart(0x21, 0);
        Camera_I2C_MasterWriteByte(RGB565_640_480[2*i]);   // reigster to write to (clock)
```

```
        Camera_I2C_MasterWriteByte(RGB565_640_480[2*i+1]); // data to write to register
        Camera_I2C_MasterSendStop();
        CyDelay(1);
    }
}

/* [] END OF FILE */
```

APPENDIX 2C, TFT.C

NOTE: This code was provided by the 6.115 staff in the TFT setup manual

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include "tft.h"

//=============================================================
// write8_a0()
// writes an 8-bit value to the TFT with the D/C line low
//
// Arguments:
//      data - 8-bit value
//=============================================================
void write8_a0(uint8 data)
{
    DC_Write(0x00);                                     // set DC line low
    SPIM_1_WriteTxData(data);                           // send data to transmit buffer
    while (!(SPIM_1_ReadTxStatus() & 0x01)){};          // wait for data to be sent
}

//=============================================================
// write8_a1()
// writes an 8-bit value to the TFT with the D/C line high
//
// Arguments:
//      data - 8-bit value
//=============================================================
void write8_a1(uint8 data)
{
    DC_Write(0x01);                                     // set DC line high
    SPIM_1_WriteTxData(data);                           // send data to transmit buffer
    while (!(SPIM_1_ReadTxStatus() & 0x01)){};      // wait for data to be sent
}

//=============================================================
// writeM8_a1()
// writes multiple bytes to the TFT with the D/C line high
//
// Arguments:
//      pData - pointer to an array of 8-bit data values
//      N - the size of the array *pData
//=============================================================
void writeM8_a1(uint8 *pData, int N)
{
    DC_Write(0x01);                                     // set DC line high
    int i;
    for (i=0; i<N; i++)
    {
        SPIM_1_WriteTxData(pData[i]);                   // send data to transmit buffer
        while (!(SPIM_1_ReadTxStatus() & 0x01)){};  // wait for data to be sent
    }
}

//=============================================================
// read8_a1()
// reads an 8-bit value to the TFT with the D/C line high
//=============================================================
```

```
uint8 read8_a1(void)
{
       for (;;) {}                                      // read function not implemented
};

//=============================================================
// readM8_a1()
// reads multple 8-bit values from the TFT with the D/C line high
//
// Arguments:
//      pData - an array where the read values we be stored
//      N - the number of values that will be read (also size of
//          the array pData)
//=============================================================
void readM8_a1(uint8 *pData, int N)
{
       for (;;) {}                                      // read function not implemented
}

//=============================================================
// tftStart()
// this function must be called to initializes the TFT
//=============================================================
void tftStart(void)
{
    write8_a0(0x01);        // send Software Reset Command (must wait at least 5ms after command)
    CyDelay(10);
    write8_a0(0x36);        // send Memory Access Control command
       write8_a1(0x88);
    write8_a0(0x3A);        // send COLMOD: Pixel Format Set command
    write8_a1(0x55);
    write8_a0(0x11);        // send Sleep Out command
    write8_a0(0x29);        // send Display ON command
    CyDelay(250);           // delay to allow all changes to take effect
}


/* [] END OF FILE */
```

APPENDIX 2D, PROCESS_IMAGE.M

NOTE: I used methods from https://afterthoughtsoftware.com/posts/convert-rgb888-to-rgb565 to convert between RGB888 and RGB565.

```
% read in sample image
p = imread('peppers.bmp');
p_size = size(p);
p_565 = zeros(p_size(1)*p_size(2),1);
% convert pixels in image to RGB565
count = 1;
for x = 1:p_size(1)
    for y = 1:p_size(2)
        % grab RGB888 pixel values
        red = cast(p(x,y,1),'uint16');
        green = cast(p(x,y,2),'uint16');
        blue = cast(p(x,y,3),'uint16');
        % b = (blue >> 3) & 0x1F
        b = bitand(floor(blue/8), 31);
        % g = ((green >> 2) & 0x3F) << 5
        g = floor(bitand(floor(green/4), 63)*32);
        % r = ((red >> 3) & 0x1F) << 11
        r = floor(bitand(floor(red/8), 31)*2^(11));
        % combine into 1 16-bit RGB565 value
        rgb = cast(r + g + b, 'uint16');
        p_565(count,1) = rgb;
        count = count + 1;
    end
end
% save results to a text file
txtfile = fopen('peppers_rgb565.txt','w');
for i = 1:(p_size(1)*p_size(2))
    fprintf(txtfile, '%u,', p_565(i,1));
end
fclose(txtfile);
```