A Wireless VGA Streaming Solution

A 6.1151 Final Project by Travis Ziegler

Table of Contents

Introduction

Displaying image and video feeds lies at the heart of many modern electronics like security cameras, video games consoles, and general purpose computers. As such, the core of my proposed project revolves around designing an interactive image and video editing interface on the PSoC. Providing a user interface that allows for receiving compressed images or video streams, modifying them in a user-selectable way, and outputting the result in real time provides a very useful base. With this core project, expansion to a more fun and creative exploration comes naturally.

Specifically, one exciting application of this system is that of streaming. Streaming has become so popular nowadays that most of the video data we consume comes from streams -- think YouTube, Netflix, etc. Transforming the PSoC into a video streaming client would be great and do-able, given that the "core" project of displaying compressed media works at a reasonable speed and resolution. After I achieve video streaming, what's the next logical progression? Interactive streams. NVidia, Playstation, Google, and so on are currently pioneering this field by launching video game streaming services. These work by streaming the video feed of a game onto a client device, and streaming the client device's input back to the host. With this technology it suddenly becomes possible for very low-end hardware to "play" highly demanding video games. Which is precisely the final goal of my project: Creating an embedded system that enables interactive streaming of compressed video-game feeds.

Thousands of lines of code, various struggles with PSoC6 documentation, clock inaccuracies, signal glitches, DMA troubles, and bad optimizations later, the project was completed successfully. The results of my efforts are shown in the images below. Figure 1 presents the best possible picture the PSoC6 can possibly output to a VGA monitor -- a 352x480 pixel image with 15-bit color resolution. Figure 2 presents a momentary capture of an ongoing live video stream originating from a nearby Linux machine. I was able to engineer 30FPS low latency (<1.5ms) streaming over WiFi at a real resolution of 192x160 pixels (which corresponds to a virtually upscaled resolution of 192x320 pixels. More on this later.) To make this possible, the videostream had to be heavily compressed. Color information had to be downsampled by a factor of four but otherwise the compression used was lossless. This will be discussed in detail in later sections.

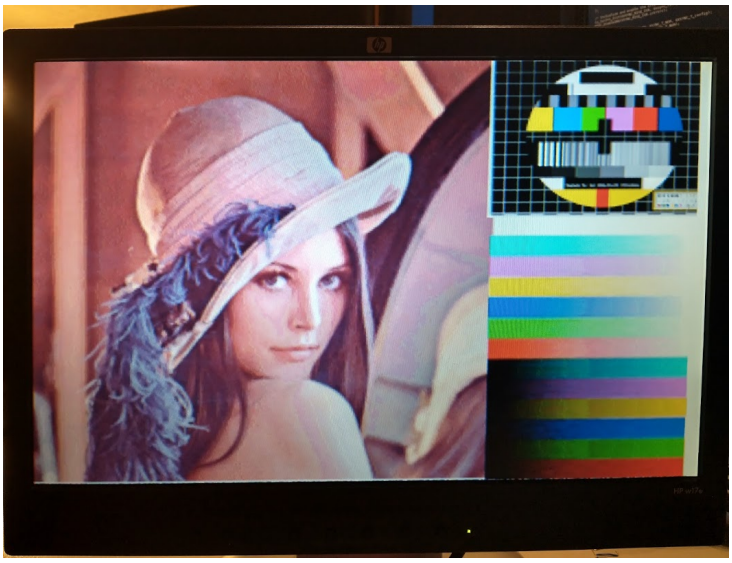


Figure 1. Full Resolution (RGB555, 344x240) VGA Monitor Test.

Figure 2. Realtime game streaming at 30FPS at a lowered resolution with heavy compression.

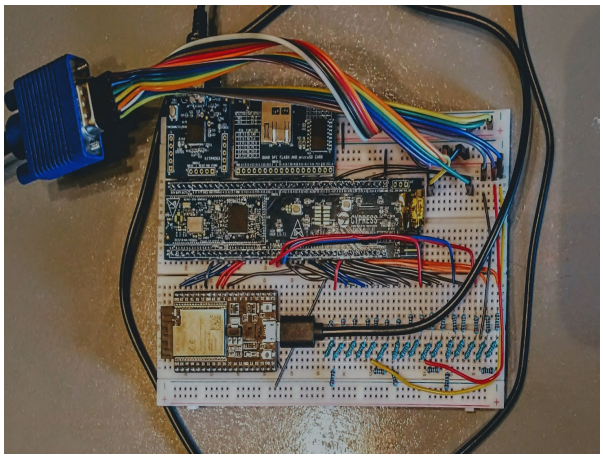


Figure 3. Hardware Setup. PSoC6 and ESP32 working together.

Link to video of PSoC/ESP32 Streaming Doom 2016 in real time:
https://drive.google.com/file/d/1-jbulYnPb3N8D912lSg62HLdD7m2vJCN/view

Producing VGA output with the PSoC

The first step to create a high performance streaming device was to output a picture to a VGA monitor. At the beginning of the project, I started working with the PSoC 5LP MCU which was quite pleasant to use. So much so that I will first explain my strategy of getting a valid VGA signal using the PSoC 5LP, and then explain how I realized a similar design on the PSoC6.

VGA is all about timing. Get the timing right, get a great image; get the timing slightly wrong, and everything breaks. A VGA cable has 6 main input pins: Horizontal Sync (HSync), Vertical Sync (VSync), Analog Red, Analog Green, Analog Blue, and Ground. To draw an image on the screen, the VGA monitor starts drawing the current analog RGB value at the top left of the screen, moves one pixel to the right, draws whichever RGB value is given at that point in time, moves another pixel to the right, and so on. Once it reaches the far right of the screen, it jumps back to the left but moves one pixel down, and repeats the process. This is how the image gets drawn horizontal line by horizontal line, until it hits the very last line. After it reaches the last line, the monitor jumps back to the top left pixel and draws the next frame. So the job of the VGA signal generator is to constantly change the analog RGB values as the monitor moves along the lines to create an image. In addition, the signal generator needs to tell the monitor how quickly it will update its RGB values, i.e. how many pixels there will be on each horizontal line, and how many horizontal lines there are in a frame. To do this, it needs to generate HSync and VSync signals which specify the video format -- the resolution and refresh rate. There are a range of standard HSync and VSync signal definitions for the VGA VESA standard -- a standard that monitors conform to. For this project, I decided to use the industry standard 640x480@60Hz VGA format, as this is the lowest resolution supported by almost all VGA monitors nowadays.

Horizontal Timing				Vertical Timing		
Scanline part	**Pixels**	**Time [µs]**		**Frame part**	**Lines**	**Time [ms]**
Visible area	640	25.422045680238		Visible area	480	15.253227408143
Front porch	16	0.63555114200596		Front porch	10	0.31777557100298
Sync pulse	96	3.8133068520357		Sync pulse	2	0.063555114200596
Back porch	48	1.9066534260179		Back porch	33	1.0486593843098
Whole line	800	31.777557100298		Whole frame	525	16.683217477656

Table 1. VGA Timings for 640 x 480 @ Hz. From http://tinyvga.com/vga-timing/640x480@60Hz.

Table 1 defines the required signals to create a 640 x 480 @ 60 Hz image. From it, we can derive the HSync signal to be a square wave with period 31.77us (whole line), with a duty cycle such that the wave is low for the 3.81us (sync pulse) and is high for the rest of the period. Using a few onboard PWM UDB Block's on the PSOC5 and carefully selected counter values, I was able to create the waves shown in Figure 4. The hardest part was finding a common period such that the waves always remained in phase with each other (notice how the waves in Figure 4 are completely stationary). Generating two independent HSync and VSync signals is easy, but generating them so that they don't drift apart over time was hard.

When programming these PWM blocks, one needs to specify a period, a compare value, and a clock source. On every clock tick, a counter inside the block gets incremented. When the counter reaches the period, it resets, and the output signal goes high. When the counter reaches the compare value, the output signal goes low. This is how I created variable duty-cycle square waves on the PSoC5. The PSoC6 works similarly, so I used the same general method there. You may find these values in Table 2.

Notice how the period of VSYNC_T is nearly eight times the period of HSYNC_T. For every multiplication by two, I had to increment the period by one in order to keep the signals from drifting. I'm not quite sure why but that's the only way it worked. Thus,

scaled_period = scale_factor * original_period + (scale_factor - 1)

I made my life easier by using non-standard fractional clock dividers that the PSoC6 luckily provides. This allowed for minimal counter scaling.

TCPWM Block	Clock	Period	Compare
HSYNC_T	10Mhz	318	300
VSYNC_T	152.4kHz	2551	2542
HBLANK_T	10Mhz	318	300

Table 2. PSoC6 HSync, VSync, and HBlank PWM definitions.

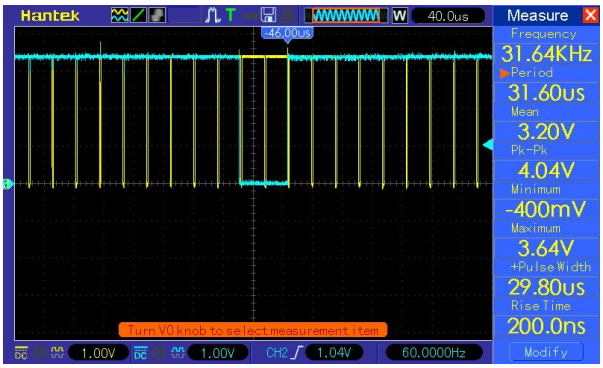


Figure 4. HSync (Yellow) and VSync (Blue) generated by the PSoC6.

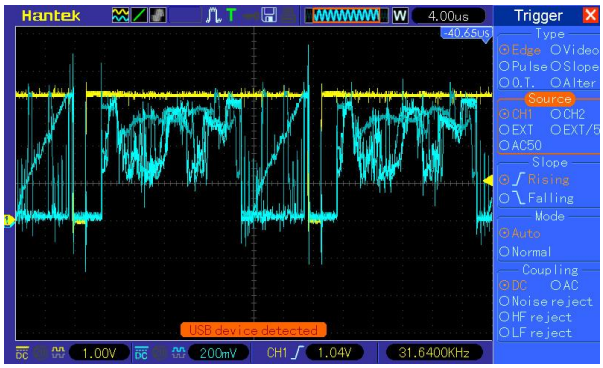


Figure 5. Hsync (Yellow) and Analog R Channel (Blue). Showing image.

At this point, after connecting the output of these HSync/VSync PWM blocks to the monitor, the monitor shows a black screen and recognizes that it's supposed to function in 640x480@60Hz mode (which can be verified in the settings panel of the monitor). Now the obvious question arises: how do we output pixel values? To create a picture with the defined 640x480 resolution the analog R, G, and B pins on the VGA connector need to be driven to a voltage between 0-0.7V (minimum - maximum color) at a frequency of 25.175MHz. Only if we can change the RGB pins at this frequency, will we achieve the 640x480 resolution specified by HSync and VSync. If the analog R, G, and B pins switch any slower than 25.175MHz, the actual resolution of the display will be lower than 640. For instance, if we generated new R, G, and B values every 12Mhz, our "achieved" horizontal resolution would be 12/25.175 * 640 = 305.

To create the 0V-0.7V RGB signals, the obvious method would be to use a builtin DAC. Sadly, 25MHz is incredibly fast and simply writing pixel values to the PSoCs DACs for each pixel would take way too long. The DAC on the PSoC5 has a setting time of around 1us by itself (freq=1Mhz) which is far too slow. The achieved resolution would be terrible using this method. Not to mention that the PSoC6 only has a single DAC to begin with.

Instead, I used a smarter method. I decided to use GPIO pins to output a digital signal onto the breadboard, and then build my own R2R ladder DAC using an array of resistors. The settling time of resistors is in the propagation delay range (the time it takes for electrons to propagate; in the picosecond range), so this clearly eliminates the settling time issues created by other IC DACs. Additionally, the GPIO ports can supply the non-negligible current needed by the VGA monitor. The DACs on the PSoC can only supply current in the microamp range, which would not create enough voltage across the input pins to the VGA monitor, as the VGA monitor has an internal 70Ohm resistor on each pin (See the schematic diagram in Figure 7). The icing on the cake is that GPIO on the PSoC6 is very fast (>=50MHz switching frequency on all pins), so it certainly wouldn't be the bottleneck in the system. Hence, using the GPIO/R2R DAC combination was the best design choice to make.

Now that settling time and GPIO speed has been taken out of the equation, the limiting factor in achieving high resolutions is *how fast we can write digital values to the GPIO pins.* The PSoC5's fastest CPU clock is 74MHz, so if we could update a GPIO Port's value every three clock cycles (74MHz/3 = 25MHz), then we could achieve extremely high resolutions. However, this is not feasible. Reading data from SRAM, and writing it to the GPIO pin takes more than 3 cycles. Furthermore, we cannot let the CPU fetch data from SRAM and write it to the GPIO port every cycle, because the CPU has to do many other things (such as decode an incoming compressed video stream). The solution to the problem is Direct Memory Access, or DMA.

The PSoC5 has at least one DMA controller, while the PSoC6 has three. The function of a DMAC is to move data from addressable memory from one location to another, independent of the CPU. Thus, we can create a framebuffer in SRAM that encodes the values needed to be written to the GPIO, then program the DMAC to constantly move data from the framebuffer to the memory mapped GPIO port. On the PSoC5, this can be easily achieved by using an 8-bit UDB Control Register and then simply mapping each bit to a GPIO port, as seen in Figure 7. The DMAC controller can then transfer data directly from SRAM to the UDB Register, and automatically increment the SRAM address after each transfer. As an example, if bit 0 of the UDB register is mapped to GPIO Pin 9.0, and our framebuffer looked like this: {0x00, 0xff, 0x00, 0x01}, then the sequence of the DMAC would be:

- Move 0x00 into GPIO register → Pin 9.0 goes low.
- Move 0xff into GPIO register → Pin 9.0 goes high.
- Move 0x00 into GPIO register → Pin 9.0 goes low.
- Move 0x01 into GPIO register → Pin 9.0 goes high.

When the DMAC is enabled and gets triggered to start a transfer, it transfers the data very quickly and consistently (around 14-20 cycles or so). However, it can only do this for a maximum of 65536 bytes, after which it either disables itself or can automatically start another pre-programmed transfer (after some delay). Going into the details about how exactly to set-up DMA Transmit Descriptors on the PSOCs would take pages, so I will try to simplify the explanation as much as possible and explain how and when to output data to GPIO pins in order to create a valid VGA signal. Another caveat about VGA is that there

exist blanking intervals during which black (e.g. 0V) has to be output to the monitor. These intervals happen slightly before and after vertical and horizontal sync pulses. To account for these, I used digital logic blocks on the PSOC5 and extra horizontal blanking timers on the PSOC6. The timing logic is shown in Table 3.

Time	Event	Description
t	HSYNC_T goes low, enter ISR.	If we're not inside the vertical blanking interval, the ISR: - reloads the DMAC with a pointer to the next horizontal line inside the framebuffer in SRAM. DMA Channel is enabled and will be triggered by HBlank_T. If we are inside the vertical blanking interval, the ISR: - disables the DMA channel if enabled.
t + BLANKING INTERVAL	HBLANK_T goes low	HBLANK_T going low triggers the enabled DMA channel and triggers a transfer for n number of bytes, where n is the achievable horizontal resolution. n has to be such that the transfer finishes before the next HSYNC_T event.
t - BLANKING INTERVAL	DMAC finishes the transfer	The last byte on the horizontal line has just been written to the GPIO port, and the DMA channel disables itself, waiting for the next HSYNC_T event.

Table 3. DMA Timing process for PSoC6. See ISR `dma_line_done(void)` in code appendix.

The above logic was enough to display a simple 8-bit color image on the PSoC5 by letting the DMAC write to the 8-bit UDB register. This method achieved a horizontal resolution of around 250 pixels, which means that the DMA updated the GPIO pins at around 10MHz.

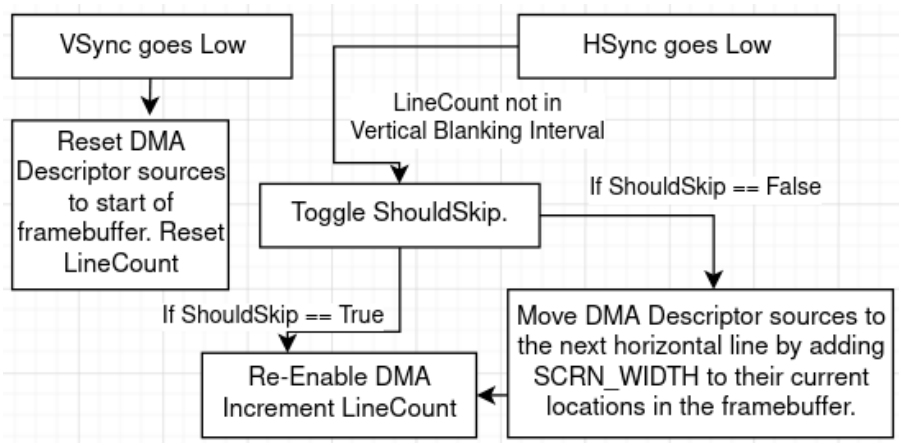


Figure 6. Operation of reloading DMA Transaction Descriptors to iterate over the framebuffer.

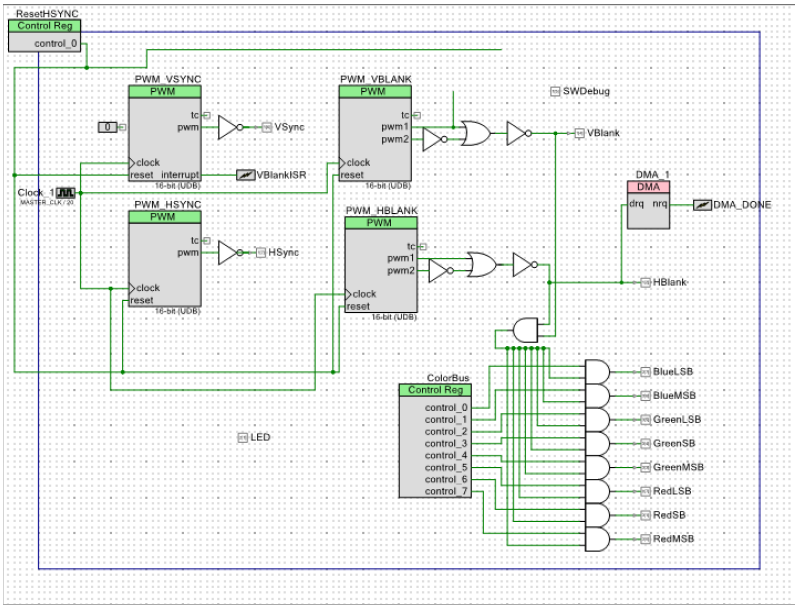


Figure 7. PSoC5 Schematic used to drive VGA. Extra control logic for starting HSYNC and VSYNC at the same time is in place.

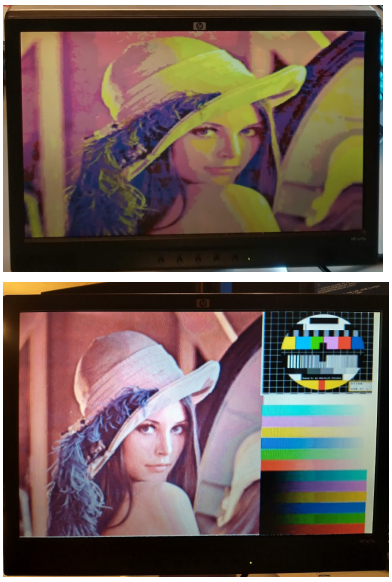


Figure 8. PSOC5's RGB332 247x240 (top) versus PSOC6's RGB555, 344x240 (bottom)

This is the general methodology I used to display an image on the VGA monitor. I was very happy when I got it to work on the PSoC5 and looked forward to just copy-pasting my code to run it on the PSoC6 when it finally arrived. Yet, to my surprise, the PSoC6 uses a completely different development environment and has no UDB components and schematic editor. So I had to start over. I used PSoC6's device configurator to laboriously find the right clock, period, and compare values to configure the PWM blocks all over again. I had to re-learn how to program the DMA controllers to do what they were supposed to do. Many things went wrong or simply didn't work, and there was little documentation on the topic that it got very frustrating. But in any case, I made some progress by reading every relevant datasheet, code example, and community forum post until I finally got things working.

The first major problem I encountered was that UDB components no longer exist on the PSoC6, so I couldn't simply just write to a control register using Cypress' API. After days of digging, I found Cypress' PSoC 62 Register Technical Reference Manual (TRM) which lists out all the registers of the PSoC62 MCU. On page 705, it actually lists where in memory each GPIO Port gets mapped to. Thus, it turned out to be possible to directly write to GPIO ports by just finding the right address of the port, and then writing values to it. For instance,

```
*((uint32_t) 0x40320000) = 0x0f;
```

sets the GPIO pin output values for Port 0 to:

P0.0	P0.1	P0.2	P0.3	P0.4	P0.5	P0.6	P0.7
3.3V	3.3V	3.3V	3.3V	0V	0V	0V	0V

Simple as that! This was a fantastic realization, as I now did not have to rely on the obsolete UDB components anymore. I could just directly write to memory and update the state of the GPIO Ports using DMA, like I did on the PSoC5.

Configuring the DMA channels to actually fire when they were supposed to, configuring the interrupts to work, and stitching them together was a big hassle no doubt (implementation details I will not bore you with), but realizing that GPIO pins were memory mapped was key. It was at this point that I knew the project would be a success.

It's also worth mentioning that I'm using 15-bit color on the PSoC6. How? The concept is generally the same; yet, instead of dealing with one DMA controller, I had to deal with two separate DMA controllers (called DW0 and DW1 on the PSoC6). Each controller writes to a separate set of GPIO pins, and their transfers are interleaved. This gets into the complicated topic of how the PSoC6 handles bus arbitration between various bus masters. I will come back to this later when talking about dual core processing, but for now just realize that it is possible to let two DMA controllers write to two separate ports from two separate frame buffers at the same time. Not at all easy -- but possible. (The timings are so strict that removing a single "NOP" from one of the ISR's causes the DMA controllers to fall out of sync and break the image).

As a minor note, the DMA ISR only moves to the next horizontal line every two horizontal lines, so as to virtually scale the image in the vertical axis by two. This is to account for the fact that the horizontal resolution is the limiting factor, and otherwise we would get strange portrait resolutions like 344x480. For some applications this may be desired, but for high speed video streaming, the higher portrait resolution would just slow things down unnecessarily.

Below in Figure 9 you can see the final hardware setup of the PSoC6. The R2R ladders are standard and use 100Ohm and 200Ohm resistors for R and 2R. These were specifically chosen to not draw too much current from the PSoC6 but enough to create an acceptable voltage for the VGA monitor (max current draw is 3.3V/300Ohm = 11mA which is fine). Also, before outputting to the VGA, another 200Ohm resistor is put in series in order to scale the voltage to the required 0V-0.7V.

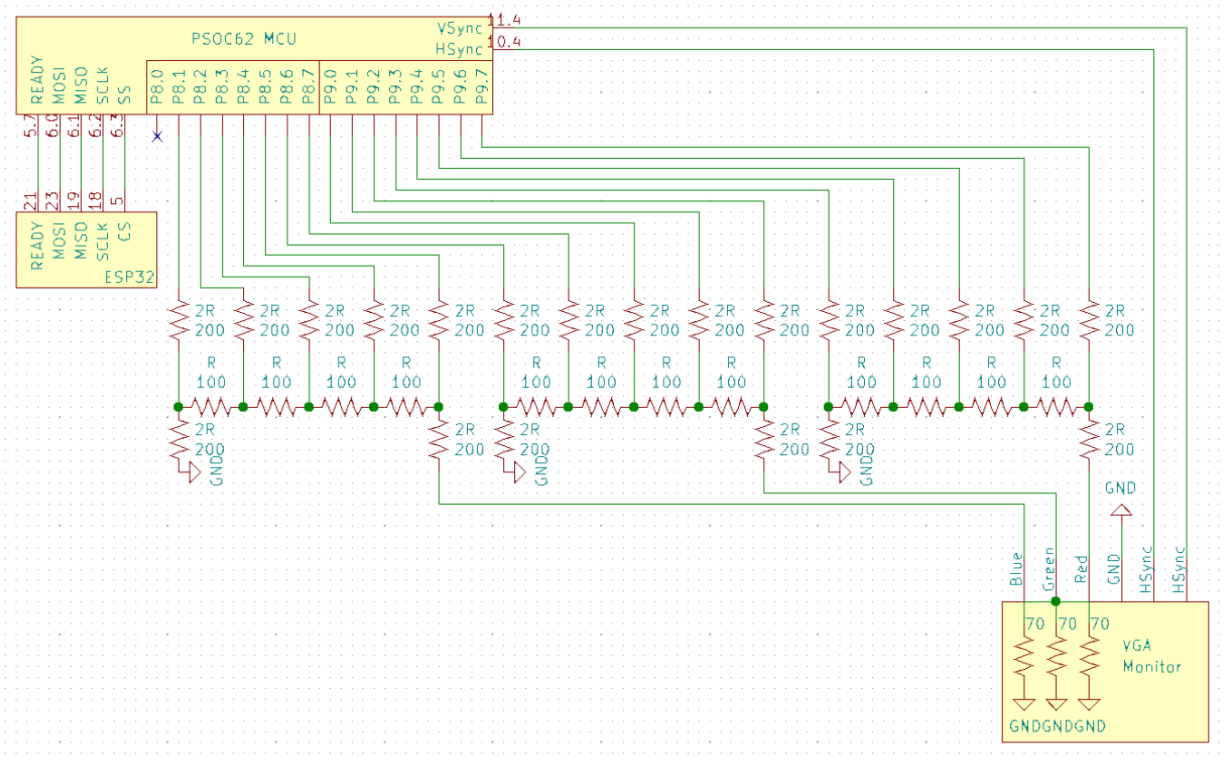


Figure 9. Master hardware schematic. PSOC6 and ESP32 communicate over SPI. Three 5-bit R2R ladder DACs convert digital GPIO signals to 0-0.7V Analog signals.

Transmitting Compressed Video Data

After I proved it possible to output a clean VGA image using the PSoC5, I had to move toward data compression. At the time, I did not know how fast the WiFi on the PSoC6 would be, or how fast the PSoC6 could decompress data, so speed was the golden priority. I chose a compression scheme roughly following the JPEG standard. I will broadly outline the steps involved in compression and decompression and then comment on their implementations (some details omitted).

Compression:

1. Given an 8-bit grayscale image input image, divide it into 8x8 chunks, i.e. matrices.
2. For each 8x8 matrix, compute the two dimensional Discrete Cosine Transform (2D DCT) of the matrix.
3. Optionally quantize the matrix by element-wise division or simply zeroing out values.
4. Loop through the matrix in a zigzag fashion. As you go, use predefined Huffman Tables to encode the size/length of the element and its magnitude in a single Variable Length (1-16bit) Huffman code. Sequentially append each code, followed by an encoded amplitude of the DCT coefficient to a bitstring.

Decompression:

1. Given the encoded bitstring produced above, loop through all the Variable Length Huffman Codes, recover the size/length and amplitude of the element and place it into an empty 8x8 matrix (again, in a zigzag fashion).
2. Optionally inverse quantize it by multiplying.
3. Take the two dimensional Inverse Discrete Cosine Transform (2D IDCT) of the matrix.
4. Place the 8x8 matrix (pixel values) in its proper 8x8 block of the 8-bit grayscale image.

To say a few words about Variable Length Huffman (VLE) coding: If we have this mapping

Huffman Codeword (Binary)	Number
0	10
1	23
01	44
11	76

Then we essentially encode 10 with 1 bit. 23 encodes with 1 bit. 44 encodes with two bits. 76 encodes with 2 bits. Now consider encoding the number sequence 10, 23, 44, 76. Realize that in binary 10 takes 4 bits to represent, 23 takes 5 bits, 44 takes 6 bits, and 76 takes 8 bits. Adding all these, we get that the raw, unencoded sequence takes 23 bits to store. However, if we use the Huffman Codewords instead, we see that 10, 23, 44, 76 encodes to 010111 -- which is only 6 bits! Thus the compression ratio is 23:6.

Of course, Huffman coding in JPEG is not this simple. In JPEG each codeword actually encodes two pieces of information, and we have different types of codewords. Each codeword is followed by a variable length amplitude which is related to the one's compliments of a signed number. These codewords and amplitude build a run-length encoding scheme for an 8x8 DCT matrix. Run-length encoding (RLE) makes it very easy to compress large amounts of zeros. Since DCT matrices are fairly sparse by nature (since high frequency components in an image are rare), they make perfect candidates for RLE. The idea behind RLE is that you can encode a string of zeros followed by a number like so:

0008 RLE-encodes to (3,8)
00002 RLE-encodes to (4,2)
(3, 6) RLE-decodes to 0006

As seen above, we save space by simply storing two numbers as a tuple (NUMBER OF ZEROS, AMPLITUDE).

Now if we combine RLE with Huffman codes, we get the JPEG encoding scheme. A JPEG Huffman codeword stores the number of zeros, called RUNLEN and the length in bits of the following amplitude, called AMP_SIZE. In other words, codeword = (RUNLEN, AMPSIZE). Then the next AMPSIZE bits following the codeword specify the actual value of the amplitude. So for example, if we have a codeword that decodes to (5, 2) and the following two bits are 1 and 0, then the decoded string would be: 0000010.

The last piece of the puzzle is how codewords are generated. The answer to that is that the JPEG people provide default codeword tables (generated from thousands of natural images) that make frequently appearing RUNLEN / AMPSIZE pairs take up less space than infrequent RUNLEN / AMPSIZE. By making frequent codewords take up less space, a lot of space is saved on average. For

instance, the smallest codeword is 00 which maps to a run length of zero, with an amp size of one… which is simply a one! Which makes sense because 1 is very common to find inside a sparse DCT matrix. On the other hand, the 16bit codeword 1111111110010101 maps to three zeros followed by an ampsize of 10. An ampsize of 10 is a number in the range of -512 to -1023 or 512 to 1023 which is very uncommon. Thus, it becomes apparent how we can save a lot of space using this method. The full huffman table exists in ivunit/jpeg/table.c.

With this compression, I've achieved compression rates similar to JPEG. Compression ratios of 256x256 24 bit grayscale images are around 10:1 to 10:2. As a specific example, a compressed test image of Lenna takes 7276 bytes to store (instead of the raw 256*256 = 65536 bytes). Further results are discussed later.

The above compression only deals with grayscale images split into 8x8 chunks. However, I wanted to transmit color. This needs extra logic, as the issue with color is that it takes up a lot of space; instead of having one m x n grayscale channel, you have 3 m x n color channels. So a 256 by 256 pixel image would take up 256 * 256 * 3 bytes to store instead of just 256 * 256. This is problematic because of throughput limitations in sending data over WiFi. The solution is to leverage the fact that the human eye is far less sensitive to color information than it is to black and white / light and dark information. If we could somehow separate light/dark information from color information, we could more heavily compress color without too much loss of quality. Luckily, we can do exactly that using colorspace conversion from RGB into the YCbCr colorspace. YCbCr represents a color using one luminance channel (Y) and two chrominance channels (Cb, Cr). The Y channel holds light and dark information while Cb, Cr hold color information. Figure 10 shows the result of the color conversion and downsampling of color information.

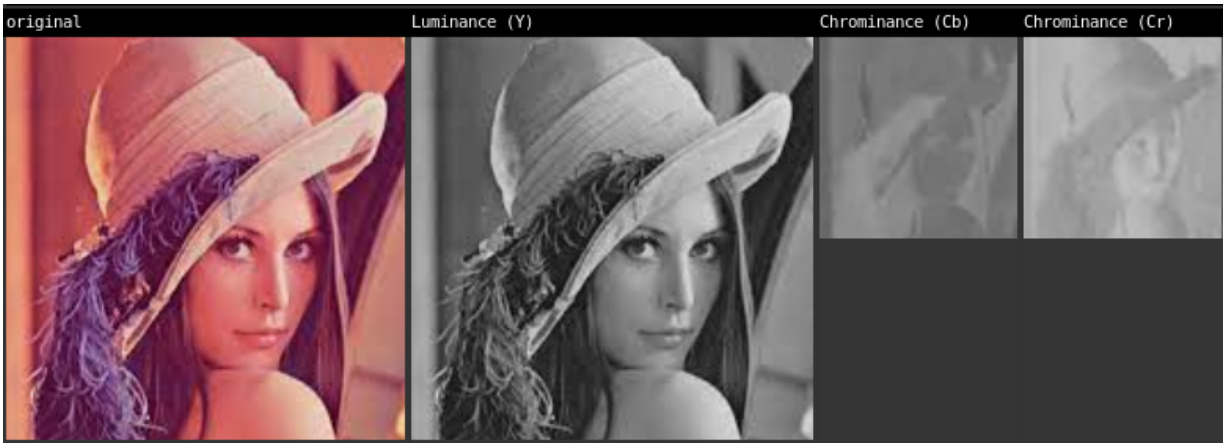


Figure 10. Lenna split into luminance and chrominance components, with chrominance downsampled by a factor of two. Downsampled chrominance takes less bits to transmit.

To summarize, we downscale the chrominance channels because our eyes are less sensitive to it, while maintaining the high resolution luminance channel. Then we compress each channel separately as described above (DCT, then huffman encodes it), and we concatenate the resulting three bit strings into one large bitstring. This large bitstring then represents a single compressed frame. That's the compression routine.

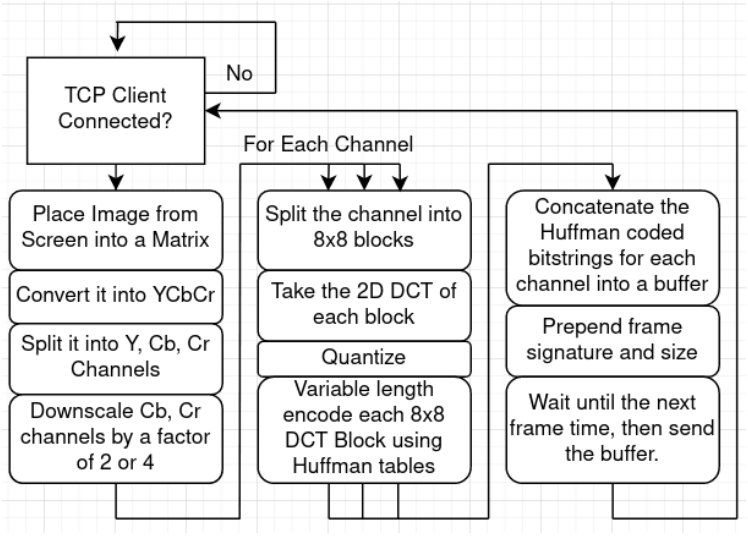


Figure 12. Transmission / Compression pipeline running on a Linux machine.

Comments on Implementation

A note on color space conversions: Color space conversions from space M to N are implemented by matrix multiplying a pixel vector in M by a color conversion matrix to receive a vector in space N. There exist multiple YCbCr color spaces but I chose the ITU-R BT.709 color space because the decompression matrix has two zeros and three ones in it, which makes computing it easy. In my implementation, I also integer approximate the floating point numbers for speed. See Figure 8.

$$\begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.2126 & 0.7152 & 0.0722 \\ -0.1146 & -0.3854 & 0.5 \\ 0.5 & -0.4542 & -0.0458 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.5748 \\ 1 & -0.1873 & -0.4681 \\ 1 & 1.8556 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix}$$

```
int32_t red = ((1<<21) * y + 0       * cb + 3302595 * cr) >> 21;
int32_t grn = ((1<<21) * y - 392796  * cb - 981677  * cr) >> 21;
int32_t blu = ((1<<21) * y + 3891475 * cb + 0       * cr) >> 21;
```

Figure 8. Colorspace conversion matrices (Left) and code to decode YCbCr into RGB (Right).

A note on the DCT / Variable Length Huffman coding: I will spare implementation details. Reading the JPEG whitepaper would be far more concise than anything I can write in a few pages. If you wish, the implementation of this whole process is around 1300 lines of C code, and can be found in ivunit/jpeg/jpg.c. I used the ITU's (International Telecommunication Union), original 1992, DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES - REQUIREMENTS AND GUIDELINES paper for Huffman tables and the 1991 JPEG whitepaper to help guide my implementation. I also drew ideas from Popovie's paper THE FAST COMPUTATION OF DCT IN JPEG ALGORITHM to compute a fast one-dimensional integer DCT / IDCT which I could then use to compute the two-dimensional cases. My implementation of these is very fast. My 1D, 8 element IDCT takes:

27 Bitshifts. 30 Addition/Subtraction. 3 Division. 3 Multiply.

which is fairly competitive with the literature. See ivunit/jpeg/jpg.c, dct8 and idct8 for the implementation. Also note that I really just need to optimize decompression because I have practically infinite compute power for compression on my main PC compared to my PSoC6's 150Mhz CM4. In my decompression routines, I make no dynamic memory allocations, I use "restrict" pointers, static inline functions to help GCC optimize my code. I also managed to enable -Ofast optimization to let GCC know to heavily optimize my PSoC code.

Now that I've discussed transmission, I will discuss receiving frames. A discussion of overall performance of the system in terms of speed and compression follows.

Receiving Compressed Video Data

At first I attempted to use the PSoC6's built-in WiFi capabilities. However, I found this quite troublesome, as there was practically zero documentation, and I ran into many roadblocks. For instance, the PWM timers for the VGA's would just stop working after initializing the WiFi module, FreeRTOS interfered with time critical DMA interrupts, and the WiFi transmission speed was disappointingly slow (raw data transmission was around 170Kb/s with maximum optimizations turned on). So I scrapped the idea of using the PSoC6's WiFi, and instead moved the WiFI interface to a trusted, well-known, and reliable microcontroller: The ESP32. Espressif's ESP32 MCU is a fast, general purpose IOT device. It benchmarked at around 1Mb/s transfer speeds over LAN, beating the PSoC6 by a large margin. The ESP32's job is to connect to my transmitting server over TCP/IP and relay data to the PSoC6 over SPI. SPI is very fast; it benchmarked around 750Kb/s continuous transfers using a clock speed of around 11MHz and a transfer size of 512 bytes.

Because I was building a realtime system, I couldn't afford to buffer an entire frame's worth of data on the ESP32, and then send it in one go. This would have increased latency too much. So instead, I use rolling buffers that output small bursts of data over SPI as soon as enough data comes in. Again, I will spare writing down the implementation details, but you may find the ESP32 project under esp32/tcp_client. Also, refer to the flowchart in Figure 13 to get an idea of how the implementation works.

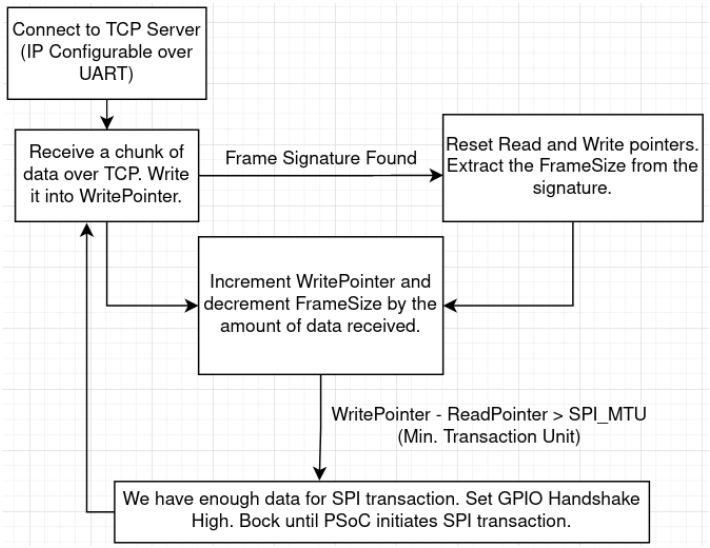


Figure 13. ESP32 Relaying Pipeline. Continuously relays streaming frame data from TCP/IP to SPI

using rolling buffers.

I will explain how the PSoC6 receives and processes data over SPI. Assume for now that the entire CPU is running in an infinite loop. Since the PSoC6 is the SPI master, it has to initiate SPI transactions. However, it cannot arbitrarily choose to initiate transfers because the ESP32 might not have enough data available or might otherwise be preoccupied with something. Instead, the ESP32 notifies the PSoC6 when it is ready to start a transaction by setting a GPIO pin high. This causes an interrupt on the PSoC6, the PSoC6 sets a flag, and on the next iteration of the infinite loop, the PSoC6 initiates a blocking SPI transfer of SPI_MTU bytes (Min. Transaction Unit). The received bytes of this transfer go into a large, accumulating buffer that holds data for one frame (of course there exists logic to properly increment the buffer pointer and reset it after a full frame has been processed). In any case, frame data builds up over successive SPI transfers, as the ESP32 receives more data over TCP. At some point, the entire frame has been received (the PSoC realizes this if a transaction contains many trailing zeros). Once this is the case, the PSoC decodes the frame. I've decided to include the decode routine below, as it might help explain the process.

```
/* Check if we've received an SPI_MTU Block ending in zeros. If this is the
 * case, then we know the frame is done. Note that it would probably be better
 * to parse the size of the frame when encountering a start symbol instead of
 * relying on this.
 */
if (huffman_data[huffman_data_i + SPI_MTU - 3] == 0 &&
        huffman_data[huffman_data_i + SPI_MTU - 2] == 0 &&
        huffman_data[huffman_data_i + SPI_MTU - 1] == 0) {

        /* Huffman Decode YCbCr channels */

        // +10 because we have a 10 byte signature at the beginning.
        huffman_data_i = decompress_channel(ychannel,
                huffman_data+10, IMG_WIDTH, IMG_HEIGHT);
        huffman_data_i += decompress_channel(cbchannel,
                10+huffman_data+huffman_data_i, CHROMA_WIDTH, CHROMA_HEIGHT);
        decompress_channel(crchannel,
                10+huffman_data+huffman_data_i, CHROMA_WIDTH, CHROMA_HEIGHT);

        /* YCbCr convert to RGB and draw fill the shared VGA buffer */
        dec_ycrcb(ychannel, cbchannel, crchannel,
                SUBSAMPLE_CHROMA, 0, 10, IMG_WIDTH, IMG_HEIGHT);
}
/* If frame didn't end, next frame will be written to this idx */
else huffman_data_i += SPI_MTU;
```

The entire routine is shown in Figure 14 as a flowchart.

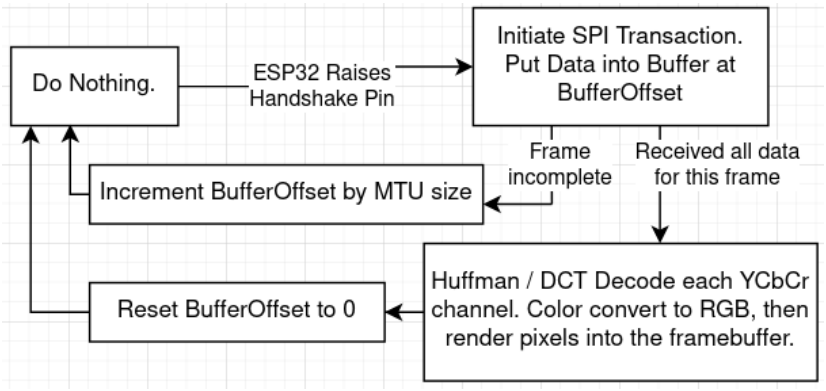


Figure 14. PSoC6 Core 0 (CM4) operation. Handles incoming frame data over SPI and decompresses full frames into the VGA framebuffer.

At this point, we can transmit frames from the Linux server to the PSoC6 in an efficient manner. Now the question arises: how do we output these frames to the VGA monitor? How can we handle all this decompression without interfering with the DMA interrupts? Well, it might be possible to put everything in one core, using interrupts. However, the PSoC6 has two cores, and since VGA is extremely time sensitive, I thought it reasonable to dedicate a core to handling VGA.

The PSoC6 has two cores: Core 0, called CM0+, and core 1, called CM4. I configured CM4 to clock at 150 MHz and CM0+ to clock at 50MHz. CM0+ is in charge of servicing VGA interrupts and updating DMA transmit descriptors to move along the framebuffer. CM4 is in charge of heavy lifting which includes initiating SPI transfers, decompressing incoming frames, and writing the RGB values to a framebuffer. In a few words:

- CM4 writes to the framebuffer.
- DW0 and DW1 controllers read from the framebuffer.
- CM0+ updates DW0 and DW1 descriptors periodically to move along the framebuffer.

It's finally time to elaborate on what this "framebuffer" that I've been referring to actually is. In the VGA section, I've explained that a framebuffer is simply an area in memory that holds the bits needed to be written to the GPIO DAC. In C, this would consist of two char arrays: one Port8 char array and one Port9 char array for a total of 15 bit color (recall a char is 8 bit). For simplicity, I've been using the word "framebuffer" to refer to both Port8 and Port9 arrays. To place an RGB pixel at row _r and column _c, the following code gets invoked:

```
/*
 * Places an RGB pixel at a given row and column
 */
#define PUT_PIXEL(_r, _c, red, grn, blu)                                    \
        port8_fb[(_r) * SCRN_WIDTH + _c] = (grn<<6) | (blu<<1);             \
        port9_fb[(_r) * SCRN_WIDTH + _c] = (red<<3) | (grn>>2);             \
```

Where the parameters red, grn, and blu are 5 bit unsigned integers. We split these bits into the Port8 and Port9 framebuffers to achieve the following layout (pulled from main.c of CM4 project).

```
/* Once we know the RGB555 (15bit number), we write them to the shared
 * framebuffer so that DMA can output it to the three DACs over GPIO
 * Format:
 *
 *      Port8 (LSB->MSB)                  Port9 (LSB->MSB)
 * [ X B0 B1 B2 B3 B4 G0 G1 ]    [ G2 G3 G4 R0 R1 R2 R3 R4 ]
 *      \...DAC..../   \.....DAC........./   \...DAC..../
 *           |                |                    |
 *        Analog B         Analog G            Analog R
 */
```

B0 denotes the LSB of Blue and it's mapped to GPIO Pin 8.1. B1 is the second bit of Blue and is mapped to GPIO Pin 8.2, and so on. Thus by writing shifting RGB values to Port8 and Port9, we directly control the analog RGB signals that go into the VGA monitor. I hope this demystifies the framebuffer's functionality. This leaves two questions: where does the framebuffer live, and how is it accessed by both CPUs and by the DMA seemingly at the same time?

Addressing question 1, the Port8 and Port9 arrays live in a user-defined area in SRAM. Programming both core CM0+ and CM4 requires two separate projects that are tied together with identical linker scripts. A linker script is a script that tells the compile at which addresses to put memory. Linker scripts define where general purpose RAM starts, where the compiler should put executable code for CM4, where to put code for CM0+, where flash memory starts, etc. Both projects have a copy of CM0+ linker script and a CM4 linker script. The goal was to define a shared region of memory that both cores can access. To do this, I created a shared linker script directory and moved the default CM0+ and CM4 linkers to the shared directory. I then carefully expanded the flash size of CM0+ (because by default it only has 8192 bytes which is not enough for holding all the VGA code), making sure to modify the CM4 linker script accordingly so that no overlapping sections occur. Then, in both linkers, I defined created a new memory area called public_ram and defined a section called .cy_sharedmem to be placed in public_ram:

CM0+ Linker (Partial Extract):

```
ram            (rwx)   : ORIGIN = 0x08000000, LENGTH = 0x2000
public_ram     (rwx)   : ORIGIN = 0x08002000, LENGTH = 0x60000
flash          (rx)    : ORIGIN = 0x10000000, LENGTH = 0x5000
```

CM4 Linker (Partial Extract):

```
public_ram     (rwx)   : ORIGIN = 0x08002000, LENGTH = 0x60000
ram            (rwx)   : ORIGIN = 0x08062000, LENGTH = 0x9D800
flash          (rx)    : ORIGIN = 0x10000000, LENGTH = 0x200000
```

As you can see, the newly created public_ram section starts at 0x8002000 and has a size of 0x60000. This comes from the fact that I'm outputting VGA at a resolution of 352x480 pixels, and we need two bytes per pixel, so we need at least 352*480*2 = 0x52800 bytes of space. The other sections have been modified to accommodate for the extra public_ram section (ram shrunk). Understanding these numbers is not too important. Anyway, here is the cy_sharedmem definition:

```
 /* To use unprotected public RAM, uncomment the following .cy_sharedmem.*/
.cy_sharedmem (NOLOAD):
{
```

```
. = ALIGN(4);
__public_ram_start__ = .;
KEEP(*(.cy_sharedmem))
. = ALIGN(4);
__public_ram_end__ = .;
} > public_ram
```

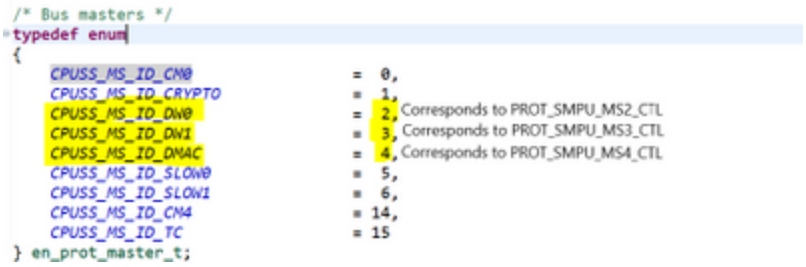
And this is the declaration of the framebuffers (from main.c in CM4 project):

```
/*
 * The main VGA Framebuffers which live in shared memory.
 *
 * DMA Controllers DW0, DW1 will read form these buffers and
 * sequentially output the bytes stored in them into their respective
 * GPIO Ports (Port8 and Port9).
 *
 * CM4 Core fills these buffers with RGB Data.
 *
 * CM0+ Core configures DW0, DW1 to iterate through these buffers for
 * each horizontal line in a frame, then resets the DataWires to start
 * at the beginning for netx frame.
 */
volatile uint8_t port9_fb[SCRN_SIZE] CY_SECTION(.cy_sharedmem);
volatile uint8_t port8_fb[SCRN_SIZE] CY_SECTION(.cy_sharedmem);
```

As you can see, the CY_SECTION macro places these arrays into the .cy_sharedmen which I've defined to start at address 0x8002000. Hence, to access these arrays in the CM0+ project, we can get pointers to their absolute address by doing this (from main.c CM0+ project):

```
/* Compute the addresses of the shared memory buffers. These
 * are defined in main.c in the CM4 project. They live in the
 * shared section .cy_sharedmem defined in linker script.
 */
port9_fb = (uint8_t*) 0x8002000 + SCRN_SIZE;
port8_fb = (uint8_t*) 0x8002000;
```

This answers the question of how we can access a globally shared framebuffer. To answer the question about how it is accessed at the same time by DW0, DW1, CM0+, and CM4, we need to discuss bus arbitration. DW0, DW1, CM0+, CM4, are all bus masters. A bus master can request access to the main databus, then read and write data from devices on the bus (like SRAM or memory mapped IO). Each bus master has a priority level associated with it which can be set in its corresponding Protection Unit (CPUSS) register. The bus master mapping is as follows:



We see that DW0 is bus master 2, so to set the priority of DW0 we need to modify the PROT_SMPU_MS2_CTL register. The same logic applies to other bus masters. By giving DW0 and DW1 the highest priority, we ensure that they consistently get access to the databus (they need it the most, as they are writing data from SRAM to GPIO which is very timecritical). All other bushmasters get lower priorities assigned. The relevant code (from cm4/main.c) is:

```
/*
* To avoid bus arbitration when the DMA DataWires are active, we can
* set busmaster priorities in the MSx_CTL Protection Registers.
*
* See PSOC62 Register TRM for details. Note that the addresses the TRM
* PDF provides are wrong. Looking through the source, the below addresses
* are correct for the following busmasters.
*/
volatile uint32_t* MS0_CTL = (uint32_t*) 0x40230000;
volatile uint32_t* MS1_CTL = (uint32_t*) 0x40230004;
volatile uint32_t* MS2_CTL = (uint32_t*) 0x40230008;
volatile uint32_t* MS3_CTL = (uint32_t*) 0x4023000C;
volatile uint32_t* MS14_CTL= (uint32_t*) 0x40230038;
volatile uint32_t* MS15_CTL= (uint32_t*) 0x4023003C;

/*
* Give CM0+ Core, DW0 DMA Controller, priority zero (highest);
* Give CM4 Core and other BusMasters priority three (lowest)
*/
*MS14_CTL = 0b1100000011;      // CM4
*MS15_CTL = 0b1100000011;      // TC
*MS1_CTL  = 0b1100000011;      // TC
*MS0_CTL  = 0b0000000011;      // CM0+
*MS2_CTL  = 0b0000000011;      // DW0
*MS3_CTL  = 0b0000000011;      // DW1
```

Conflicts between bus masters are handled in a round robin fashion except for when DW0 and DW1 request access. In that case, they always get priority. Arbitrating between DW0 and DW1, however, is not as easy because they need simultaneous access to SRAM and MMIO. I found no eleganent solution to solve this problem. The solution I came up with was to trigger the DW0 and DW1 transfers at the exact same time using two separate timers. These timers are like one cycle offset from each other, so that one DW starts ever-so-slightly earlier than the other. If these timing conditions are met exactly and the timers are started exactly at the right times, then DW0 and DW1 interleave with each other like so:

DW0 gets access to the bus. Reads from SRAM writes to MMIO.

DW1 gets access to the bus. Reads from SRAM writes to MMIO.
DW0 gets access to the bus. Reads from SRAM writes to MMIO.
…

This results in a ~0.25pixel shift between DW0 and DW1 on the VGA monitor which is hardly noticeable. The downside of this method is that changing any line of code in the CM0+ project, will break the VGA output and will cause either a larger than 0.25 pixel shift or one DW completely taking priority over the other. See Figure 15. for a comparison of VGA glitchiness because of incorrect bus arbitration settings.

Another bus arbitration issue -- one which I could not fully resolve -- is that of SPI transfers hogging the databus. It seems that whenever an SPI transfer occurs, the databus gets locked for the duration of the transfer, and this causes the VGA logic to start a horizontal line too late, and a horizontal line artifact appears. See Figure 15 for how these lines manifest themselves. To somewhat circumvent this from happening, I tried to synchronize SPI transactions with the vertical sync pulse to localize all the SPI transaction lines at the top of the screen, and draw the video in the lower portion of the screen. This worked somewhat and reduced the missed lines to a single missed line. I figured out that this single line happens when the framebuffer gets filled in CM4 core, so for some reason bus arbitration is not working properly for MMIO. It works well for SRAM, but not for writing to GPIO ports. That stumped me, but at that point I was so happy that I managed to fix the other major, dealbreaking bus arbitration issues, and I found myself content enough.

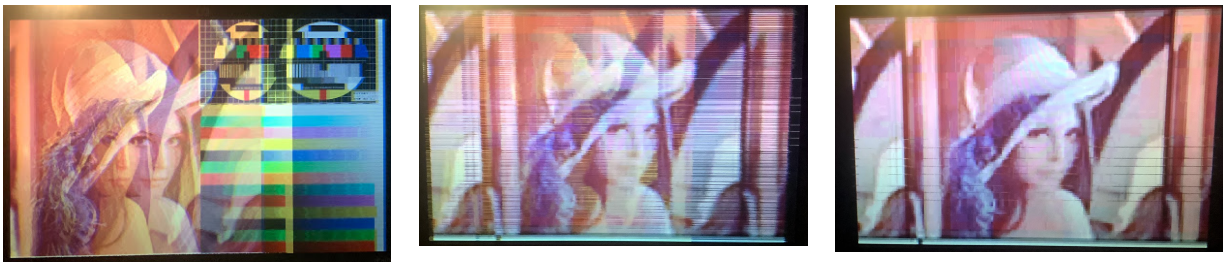


Figure 15. Bus arbitration induced issues.

Left: One DMA controller started too early and consistently takes precedence over the other (DW0 and DW1 are no longer interleaving in sync).

Center: DW0, DW1, CM0+, and CM4 all have the same bus priority set in their respective CPUSS registers. They are all fighting over SRAM and create many delays for the DMA controllers.

Right: DW0, DW1 have the highest bus master priority, but SPI transactions somehow still create DMA a few DMA line misses.

As seen, in Figure 15. bus arbitration is no joke. The reason I spend so many words discussing it, is that it can break a project if not handled correctly.

Streaming Results

After I implemented and optimized the video streaming infrastructure, it was time to tune global parameters to take complete advantage of the PSoC6. After much trial and error, performance measurements using timers and debug logs, I found impressive results in terms of quality and speed. The whole system has to produce a new frame in a less-than 32ms time period in order to sustain video streaming at 30FPS. That was the major design constraint. By benchmarking various routines in my PSoC code, I found that the bottleneck (not surprisingly) is the decompression of the JPEG-like data in the PSoC6's CM4 main routine. Huffman decoding a bitstream and then taking many IDCTs, followed by YCbCr color conversion does take its toll on the poor 150MHz CM4 core. There are three main hyperparameters to tune which trade speed for quality:

Parameter name	Function	Chosen Value
WIDTH and HEIGHT	Sets the resolution of the transmitted and output image.	160x128
NUM_COLORS	The number of colors to downsample to BEFORE DCT compression.	128
SUBSAMPLE_CHROMA	Resolution scale factor applied to the chroma channels to compress color.	4

Table 4. Parameters to trade-off quality for performance

Reducing the resolution at which we transmit and draw is an obvious tradeoff: high resolution images take longer to transmit/draw while low resolution images go faster.

By reducing the number of colors (dividing R, G, B 8-bit channel colors by a multiples of 2), the DCT matrices become more sparse and have elements with smaller magnitudes which helps in two ways: Increases compression efficiency and decreases the time it takes to decompress. Ultimately, each channel can only output a maximum of 32 colors because each channel gets a 5 bit DAC; however, if we divide channel elements to get 32 colors before DCT, this has an extremely lossy effect. Using few colors before DCT greatly reduces image quality, as the IDCT cannot reconstruct the original matrices well -- because of all the lost color information. Therefore, it is best to keep the color resolution as high as possible for as long as possible and only downsample right before writing to the framebuffer.

Chroma subsampling reduces the spatial resolution in color by reducing the resolution of the chroma (Cb, Cr) channels. See Figure 16.

No Subsampling.
2 * 256 * 256 bytes of color information

Subsample by 4.
2 * 64 * 64 bytes of color information

Subsample by 16.
2 * 16 * 16 bytes of color information

Subsample by 32.
2 * 8 * 8 bytes of color information

Figure 16. The effects of chroma subsampling.

The aforementioned parameters allowed for decompression to occur within the 32ms period of a frame. Specifically, decompression and rendering to the framebuffer took on average, 21-23ms per frame (with the above parameters). The remaining 7-9ms were spent in sending frames over WiFi, buffering frame portions in the ESP32, sending them over SPI, and buffering frame portions in the PSoC. Notice that the resolution 160x128 is very wide. The reason for this arises from the horizontal line skipping in the VGA driver -- remember that the CM0+ moves to the next horizontal line every second horizontal sync. By scaling the vertical resolution by two, we prevent pixels from looking like rectangles. Hence, the actual resolution on the PSoC6 is 160x256. To make this full screen, I upscale each pixel value before writing it, so that each pixel actually takes up four pixels, to create an upscaled resolution of 320x512. The logic to place a pixel at row _r, and column _c is:

```
/* Upsample each pixel into 4 pixels */
PUT_PIXEL(2*_r,   2*_c,   red, grn, blu)
PUT_PIXEL(2*_r+1, 2*_c,   red, grn, blu)
PUT_PIXEL(2*_r,   2*_c+1, red, grn, blu)
PUT_PIXEL(2*_r+1, 2*_c+1, red, grn, blu)
```

The end result is that the original 160x128 image gets scaled by four in the vertical, and by two in the horizontal in order to almost fill the VGA monitor's 452x480 pixel screen space.

Cryptography

My Independent inquiry (II) component is cryptography; however, the focus of my II shifted toward data compression and creating highly efficient programs. In that sense, one could consider the large amounts of work I did to perfect and optimize the DCT and Variable Length Huffman coding as a large component of my II (as that required the most research and work). Aside from that though, I read Part I and Part II of *Cryptography Engineering* by Kohno et al (chapters 1-8) which covered the basics of cryptography and discussed block ciphers, block cipher modes, hash functions, MAC codes, and how to put all these components together to create a secure channel. My initial goal was to encrypt the video stream with a secure channel. A secure channel can be built from a key, a block cipher and a MAC generator (message authentication code). For this project, I implemented the 3DES (Triple Data Encryption Standard) block cipher, and a generalized bitmap-based method for bit permutations (inspired by ideas from Donald Knuth's *The Art of Programming*), but I did not get to fully implementing the secure channel. I wanted to use the PSoC6's native hardware crypto offloading capabilities, but after I realized how much the PSoC had to be strained for VGA / video decompressing to work simultaneously, I decided to not to pursue this route of work. Any additional load on the PSoC would have had a drastic performance impact. The system as it is right now is so heavily optimized and time dependent, that (like I said), adding any code could easily break things in various subsystems.

I will briefly discuss my method for general, fast bit permutations on n-bit integers. The method relies on a Benez Network; a network that consists of multiple stages, each stage having n muxes. A single stage of this network takes an n-bit (power of two) number and moves bits from its high word to its low word if a mux is set for that bit. If a mux for a bit is not set, the bit just passest through. Subsequent stages do the same for each subword. This recurses until a subword size of one. Then the whole process repeats in the inverse, growing from a subword size of one back to the original subword size of n/2. The exact way this works is not terribly important for the purposes of this writerup, just realize that by finding

the right states of every mux (on or off), the network has the ability to permute a group of bits in a pre-defined way. An example 8-bit network is shown in Figure 17 and was taken from https://programming.sirrida.de/bit_perm.html.

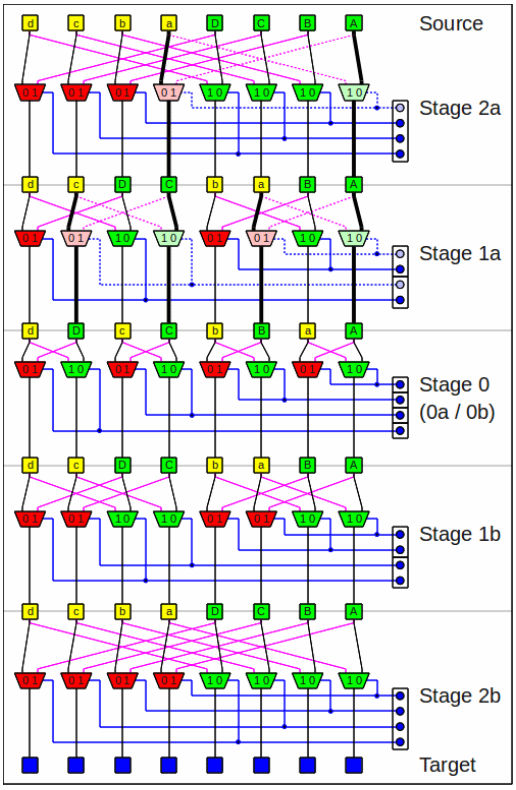


Figure 17. 8-Bit Benez Network for general bit permutations.

Looking at the diagram in Figure 17. is likely more helpful than reading about it. The pink lines represent the "path" a bit takes if it's mux is ON. The black lines represent the "path" a bit takes if it's mux is off. You may see how in the first stage (Stage 2a), every bit has the opportunity to move to a different subword (left 4 bit subword or right 4 bit subword). In stage 1a, you may see how the pattern repeats for each 4 bit subword. Each bit in the 4 bit subword has the opportunity to move into a different 2-bit sub-subword within their 4-bit subword, but they cannot cross to the other 4 bit subword. Stage zero is the recursive base case: A bit can either exchange spots with its neighbor, or stay in its place. After the base case, the recursion starts rewinding to its initial words-size. Stage 1b deals with 4-bit subwords, and Stage 0 deals with single bit subwords.

By writing an algorithm in Python (permgen.py), I managed to solve for the state of all muxes given any bit permutation. The state of the muxes is encoded by bitmasks. These bitmasks can be used to efficiently emulate the Benez network in software using fast elementary operations like Xor, Or, And, and Shifting. As an example, to solve for the following the permutation that maps ABCD -> BCDA, I can just run the following Python script:

```
src = ['A', 'B', 'C', 'D']
dst = ['B', 'C', 'D', 'A']
print(get_masks(src, dst))
```

which outputs the bitmasks:

```
[[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 0, 1]]
```

For which the stages would be:

```
A B C D        Input
| | | |
A B C D        Stage 2a       mask: 0,0,0,0
 X   X
B A D C        Stage 1        mask: 0,1,0,1
| / | \
B C D A        Stage 2b       mask: 0,0,0,1
```

The benefit of using this method is that most of the computation in executing these permutations is the generation of these masks. The actual execution of these masks is very fast. As a concrete example, the initial permutation of the DES algorithm calls for the following (numbers are 1 indexed bit positions):

***PC*-1**

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Which are encoded as the following mask (which I computed using the big python script):

```
/************************/
/* IP Permutation Mask  */
/************************/
uint64_t perm_mask_ip[] ={
  0x000000006aaaaaaa, 0x00003cf000003cf0,
  0x0033006600330066, 0x0606040106060401,
  0x1111000011110000, 0x4411441144114411,
  0x1203032112030321, 0x0d06090c02090603,
  0x00aa00a500aa005a, 0x0000663300006633,
  0x00000000e1f0f0f0
};
```

To execute this 64-bit mask the following code is sufficient:

```
#define ___do_delta_swap(n_div2)            \
       y = (x ^ (x >> n_div2)) & *mask;      \
       x = x ^ y ^ (y << n_div2);            \
       mask++;

uint64_t
perm_bitstring_64(uint64_t x, uint8_t n, uint64_t* mask)
{
       uint64_t y;
       ___do_delta_swap(32);
       ___do_delta_swap(16);
       ___do_delta_swap(8);
       ___do_delta_swap(4);
       ___do_delta_swap(2);
       ___do_delta_swap(1);
       ___do_delta_swap(2);
       ___do_delta_swap(4);
       ___do_delta_swap(8);
       ___do_delta_swap(16);
       ___do_delta_swap(32);
       return x;
}
```

which is extremely fast. How fast? I implemented three versions of general bit permutation algorithms. One 2 line version in python, one simple version in C, and finally, the fast mask-permutation C algorithm. I benchmarked all these codes. The benchmark computes the same 1000 random permutations on the same 10000 randomly generated inputs across all three functions. The results are as follows with -O3 optimization:

```
Python Results:
---------------
Total seconds:  55.606396436691284
Seconds / Call: 5.560639643669129e-06
Sanity Check:   0xa24ea601b8dd66e9

Dumb C Results:
-----------------
Ticks elapsed: 104646835474
Total seconds: 29.068298950037402
Seconds / Call: 2.9068298950037401e-06
Sanity Check:  0xa24ea601b8dd66e9

Fast C Results:
-----------------
Ticks elapsed: 495474170
Total seconds: 0.13763045227640969
Seconds / Call: 1.3763045227640968e-08
Sanity Check:  0xa24ea601b8dd66e9

Speed-up Factor Results
-----------------------
Dumb C vs Dumb Python   1.91
Fast C vs Dumb C        211.21
```

As seen, my implementation using pre-computed bitmasks is twice as fast as a dumb C implementation and 400 times as fast as an implementatino in Python.

Using the original DES Whitepaper, I implemented the DES encryption algorithm using bitmasks I generated from the paper. I then benchmarked my DES implementation with that of the popular open source OpenSSL Library, and found that my implementation is three times slower, despite all my efforts. Upon inspection, I realized that OpenSSL uses simplified permutation masks. This allows them to skip many stages in the Benez network and save computation time. This was rather discouraging, and it dawned on me that there would be no way for the PSoC to handle real time decryption of a DES data stream, so I abandoned the attempt. Instead, I focused on making the VGA work faster, better, and at a higher quality, as that was more exciting and rewarding.

Closing Comments

Working on this project was a very rewarding but difficult experience. It is safe to say that I've learned a lot about microcontrollers, and project design in general. I feel confident that if required and given enough time, I could produce production level projects on microcontrollers such as the PSoC6 or the ESP32. Since I was able to overcome the lack of documentation, code examples, and quirks of the PSoC6, I will likely be able to deal with whatever challenges other microcontrollers throw at me. I have learned a lot about optimization, and efficient programming on low-end devices, my C skills have improved greatly, and my understanding of embedded devices in terms of hardware, peripherals, toolchains, and IDEs has greatly improved. (As a fun side note, I used three development environments in this project: PSoC Creator, Modus Toolbox, and the Espressif IDF toolchain. I have to admit, using Modus Toolbox was at first overwhelming and definitely had a learning curve, but I actually came to like it over time. Perhaps even more than I like PSoC Creator.)

In any case, I'm glad my project worked out as I imagined it.

Code Appendix

There are four sub-projects in my codebase with over 7000 lines of code that I generated, so listing all the code here would be infeasible. The GitHub repository is https://github.com/travisjayday/vga-streamer. The subprojects and their locations are:

SubProjects	**Location in GitHub**
ESP32	esp32/tcp_client
JPEG Encoder / Decoder	ivunit/jpeg
DES & Bit Permutation Implementations	crypto
PSoC6 CM4 Project	psoc6/mtw/VgaStreamer/cm4_app
PSoC6 CM0+ Project	psoc6/mtw/VgaStreamer/cm0_app

Below you will find code snippets I'm particularly proud of. Keep in mind that these are taken out of context and most likely rely on nearby code / lookup-tables to work, so please look at the GitHub repository for context. At the very end, you will find a flowchart of most of the software components of this project, tying everything together.

Size 8, 1 Dimensional Integer Discrete Cosine Transform

```
/*
 * w, c: width of block. For columnwise DFT,
use w=8, c=colum
 *       For row-wise DCT, use w=0, c=0.
 */
void dct8(double b[8], double out[8], uint8_t
w, uint8_t c) {
    double g[8] = {};
    double f[8] = {};
    double f_8;

    // Stage 1
    g[0] =  b[0*w+c] + b[7*w+c];
    g[1] =  b[1*w+c] + b[6*w+c];
    g[2] =  b[2*w+c] + b[5*w+c];
    g[3] =  b[3*w+c] + b[4*w+c];
    g[4] = -b[4*w+c] + b[3*w+c];
    g[5] = -b[5*w+c] + b[2*w+c];
    g[6] = -b[6*w+c] + b[1*w+c];
    g[7] = -b[7*w+c] + b[0*w+c];

    // Stage 2
    f[0] =  g[0] + g[3];
    f[1] =  g[1] + g[2];
    f[2] = -g[2] + g[1];
    f[3] = -g[3] + g[0];
```

Size 8, 1 Dimensional Integer Inverse Discrete Cosine Transform

```
static inline void
idct8(const pix_t in[8], pix_t out[8], uint8_t
w, uint8_t c) {
    pix_t g[8] = {};
    pix_t f[8] = {};

    for (int i = 0; i < 8; i++)
       g[i] = in[i*w+c];

#if NORMALIZE_DFT
    pix_t s[8] = {};
    s[0] = sqrt(2);
    s[1] = 2.0 / (cos(1.0 * PI / 16.0));
    s[2] = 2.0 / (cos(2.0 * PI / 16.0));
    s[3] = 2.0 / (cos(3.0 * PI / 16.0));
    s[4] = 2.0 / (cos(4.0 * PI / 16.0));
    s[5] = 2.0 / (cos(5.0 * PI / 16.0));
    s[6] = 2.0 / (cos(6.0 * PI / 16.0));
    s[7] = 2.0 / (cos(7.0 * PI / 16.0));

    for (int i = 0; i < 8; i++)
       g[i] = g[i] / s[i] * 10;

    // Stage 7
    f[0] = g[0] * 4;
    f[5] = g[1] * 4;
```

```
    f[4] =  g[4] + g[5];
    f[5] =  g[5] + g[6];
    f[6] =  g[6] + g[7];
    f[7] =  g[7];

    // Stage 3
    g[0] =  f[0] + f[1];
    g[1] = -f[1] + f[0];
    g[2] =  f[2] + f[3];
    g[3] =  f[3];
    g[4] =  f[4];
    g[5] =  f[5];
    g[6] =  f[6];
    g[7] =  f[7];

    // Stage 4
    f_8  = -g[6] + g[4];
#ifndef USE_DCT_INT_ENCODING
    f_8  =   0.382 * f_8;

    f[0] =  1  *g[0];          // *8
    f[1] =  1  *g[1];          // *8
    f[2] =  0.707  *g[2];           // *6
    f[3] =  1  *g[3];          // *8
    f[4] =  0.541  *g[4] + f_8;    // *4
    f[5] =  0.707  *g[5];               // *6
    f[6] =  1.306 *g[6] + f_8;    // *10
    f[7] =  1  *g[7] ;          // *8*/
#else
    f_8  =  3 * f_8;

    f[0] =  8  *g[0];          // *8
    f[1] =  8  *g[1];          // *8
    f[2] =  6  *g[2];           // *6
    f[3] =  8  *g[3];          // *8
    f[4] =  4  *g[4] + f_8;    // *4
    f[5] =  6  *g[5];               // *6
    f[6] =  10 *g[6] + f_8;    // *10
    f[7] =  8  *g[7] ;          // *8*/
#endif

    // Stage 5
    g[0] =  f[0];
    g[1] =  f[1];
    g[2] =  f[2] + f[3];
    g[3] =  f[3] - f[2];
    g[4] =  f[4];
    g[5] =  f[5] + f[7];
    g[6] =  f[6];
    g[7] =  f[7] - f[5];

    // Stage 6
    f[0] =  g[0];
    f[1] =  g[1];
    f[2] =  g[2];
    f[3] =  g[3];
    f[4] =  g[4] + g[7];
    f[5] =  g[5] + g[6];
    f[6] = -g[6] + g[5];
    f[7] =  g[7] - g[4];
```

```
    f[2] = g[2] * 8;
    f[7] = g[3] * 4;
    f[1] = g[4] * 8;
    f[4] = g[5] * 4;    // f4 = g4 + g7
    f[3] = g[6] * 8;
    f[6] = g[7] * 4;

#else
    // Stage 7
    f[0] = g[0] * IDF_SF * 4;
    f[5] = g[1] * IDF_SF * 4;
    f[2] = g[2] * IDF_SF * 8;
    f[7] = g[3] * IDF_SF * 4;
    f[1] = g[4] * IDF_SF * 8;
    f[4] = g[5] * IDF_SF * 4;    // f4 = g4 +
g7
    f[3] = g[6] * IDF_SF * 8;
    f[6] = g[7] * IDF_SF * 4;
#endif

    // Stage 6
    g[0] =  f[0];
    g[1] =  f[1];
    g[2] =  f[2];
    g[3] =  f[3];
    g[4] =  (f[4] - f[7]);
    g[5] =  (f[5] + f[6]);
    g[6] =  (f[5] - f[6]);
    g[7] =  (f[4] + f[7]);

#ifndef USE_DCT_INT_DECODING
    // Stage 5
    f[0] =  g[0];
    f[1] =  g[1];
    f[2] =  (g[2] - g[3]) / 2;
    f[3] =  (g[2] + g[3]) / 2;
    f[4] =  g[4];
    f[5] =  (g[5] - g[7]) / 2;
    f[6] =  g[6];
    f[7] =  (g[5] + g[7]) / 2;

    // Stage 4
    g[0] = f[0] / 1;
    g[1] = f[1] / 1;
    g[2] = f[2] / 0.707;
    g[3] = f[3] / 1;
    g[5] = f[5] / 0.707;
    g[6] = ((0.541 + 0.382) * f[6] - 0.382 *
f[4]);
    //g[4] = (f[4] + 0.382 * g[6]) / (0.541 +
0.382);
    g[4] = (1 * f[4] +  0.353 * f[6] - 0.146 *
f[4]) / (0.923);
    g[7] = f[7] / 1;
#else
#if IDF_SF == 8
    // Stage 5
    f[0] =  g[0];
    f[1] =  g[1];
    f[2] =  (g[2] - g[3]);
```

```
    // Stage 7
    g[0] = f[0] / 4;
    g[1] = f[5] / 8;
    g[2] = f[2] / 8;
    g[3] = f[7] / 8;
    g[4] = f[1] / 8;
    g[5] = f[4] / 8;
    g[6] = f[3] / 8;
    g[7] = f[6] / 8;

#if NORMALIZE_DFT
    s[0] = sqrt(2);
    s[1] = 2.0 / (cos(1.0 * PI / 16.0));
    s[2] = 2.0 / (cos(2.0 * PI / 16.0));
    s[3] = 2.0 / (cos(3.0 * PI / 16.0));
    s[4] = 2.0 / (cos(4.0 * PI / 16.0));
    s[5] = 2.0 / (cos(5.0 * PI / 16.0));
    s[6] = 2.0 / (cos(6.0 * PI / 16.0));
    s[7] = 2.0 / (cos(7.0 * PI / 16.0));

    // Stage 7
    for (int i = 0; i < 8; i++)
       out[i] = g[i] * s[i];
#else
    for (int i = 0; i < 8; i++)
       out[i] = g[i];
#endif

}
```

```
    f[3] =  (g[2] + g[3]);
    f[4] =  g[4];
    f[5] =  (g[5] - g[7]);
    f[6] =  g[6];
    f[7] =  (g[5] + g[7]);

    g[0] = f[0] / 8;
    g[1] = f[1] / 8;
    g[3] = f[3] / 16;
    g[7] = f[7] / 16;
    g[2] = f[2] / 12;
    g[5] = f[5] / 12;
    g[6] = (7 * f[6] - 3 * f[4]) / 64;
    g[4] = (8 * f[4] + 3 * f[6] - 1 * f[4]) /
55;

#elif IDF_SF == 10
    // Stage 5
    f[0] =  g[0];
    f[1] =  g[1];
    f[2] =  (g[2] - g[3]) / 2;
    f[3] =  (g[2] + g[3]) / 2;
    f[4] =  g[4];
    f[5] =  (g[5] - g[7]) / 2;
    f[6] =  g[6];
    f[7] =  (g[5] + g[7]) / 2;

    g[0] = f[0] / 10;
    g[1] = f[1] / 10;
    g[2] = f[2] / 7;
    g[3] = f[3] / 10;
    g[5] = f[5] / 7;
    g[6] = (9 * f[6] - 4 * f[4]) / 100;
    g[4] = (10 * f[4] + 4 * f[6] - 1 * f[4]) /
90;
    g[7] = f[7] / 10;
#endif

#endif

    // Stage 3
    f[0] =  (g[0] + g[1]) / 2;
    f[1] =  (g[0] - g[1]) / 2;
    f[2] =  g[2] - g[3];
    f[3] =  g[3];
    f[4] =  g[4];
    f[5] =  g[5];
    f[6] =  g[6];
    f[7] =  g[7];

    // Stage 2
    g[0] =  (f[0] + f[3]) / 2;
    g[1] =  (f[1] + f[2]) / 2;
    g[2] =  (f[1] - f[2]) / 2;
    g[3] =  (f[0] - f[3]) / 2;
    g[6] =  f[6] - f[7];
    g[5] =  f[5] - g[6];
    g[4] =  f[4] - g[5];
    g[7] =  f[7];
```

	`` // Stage 1 out[0*w+c] = (g[0] + g[7]) / 2; out[1*w+c] = (g[1] + g[6]) / 2; out[2*w+c] = (g[2] + g[5]) / 2; out[3*w+c] = (g[3] + g[4]) / 2; out[4*w+c] = (g[3] - g[4]) / 2; out[5*w+c] = (g[2] - g[5]) / 2; out[6*w+c] = (g[1] - g[6]) / 2; out[7*w+c] = (g[0] - g[7]) / 2; }```

Size 8x8, 2 Dimensional Integer Discrete Cosine Transform	*Size 8x8, 2 Dimensional Integer Inverse Discrete Cosine Transform*
``void dct_encode(uint8_t* imdata, uint32_t width, double data[8][8], const int xpos, const int ypos) { // block = input block. Do not modify. double block[8][8] = {}; for (int i = 0; i < 8; i++) for (int j = 0; j < 8; j++) block[i][j] = imdata[(ypos + i) * width + j + xpos]; double out[8] = {}; // This calculates the rowise (left/right) transforms uint8_t r = 0; uint8_t i = 0; for (r = 0; r < 8; r++) { dct8((double*) block + r * 8, out, 1, 0); memcpy(&data[r][0], out, 8 * sizeof(double)); } // This calculates the columwise (up/down) transform uint8_t c = 0; for (c = 0; c < 8; c++) { dct8((double*) data, out, 8, c); for (i = 0; i < 8; i++) data[i][c] = out[i]; } #ifdef USE_DCT_INT_ENCODING for (int i = 0; i < 8; i++) for (int j = 0; j < 8; j++) data[i][j] /= 32.0; #endif }```	```static void dct_decode(uint8_t* restrict buf, uint16_t width, pix_t data[8][8], int xpos, int ypos) { // This calculates the columwise (up/down) transform uint8_t c = 0; for (c = 0; c < 8; c++) idct8((int32_t*) data, (int32_t*) data, 8, c); // This calculates the rowise (left/right) transforms uint8_t r = 0; for (r = 0; r < 8; r++) idct8((pix_t*) data + r * 8, &data[r][0], 1, 0); // Copy result to output buffer for (int i = 0; i < 8; i++) for (int j = 0; j < 8; j++) { int32_t d = data[i][j]; buf[(ypos + i) * width + j + xpos] = d < 0? 0 : d > 255? 255 : d; } }```

JPEG Huffman Variable Length Encoder

```
/*
 * Huffman encode an 8x8 DCT block of data with JPEG tables.
 * @prev_dcval: DC value of previous 8x8 block.
 * @b: 8x8 quantized block.
 * @buf: output buffer where huffman encoded data will be written to.
 *
 * Returns size of output buffer in bytes.
 */
uint8_t
huff(int32_t prev_dcval, int32_t b[8][8], uint8_t* buf8)
{
    uint16_t* buf = (uint16_t*) buf8;
    uint16_t* buf_start = buf;
    uint8_t* codes = (uint8_t*) calloc(128, 1);
    uint16_t* amps = (uint16_t*) calloc(128, 1);
    uint8_t* code_i = codes;
    uint16_t* amp_i = amps;

    uint16_t codeword;
    uint8_t codelen;
    int r, c, i;
    int32_t dc_amp;
    uint8_t size;
    uint16_t amp;
    uint8_t code;
    uint8_t bits_left = 16;

    /* Parse the buffer into intermediate Runlen/Size + VLI codes */
    parse_syms(HUFF_START_PARSE_SYMS, NULL, NULL);
    for (i = 1; i < 64; i++) {
        // Loop through the 8x8 block zigzag and parse
        r = row_lin2diag[i];
        c = col_lin2diag[i];
        parse_syms(b[r][c], &code_i, &amp_i);
    }

    /* The following converts intermediate codes to huffman codewords */

    /* Find DC amplitude and category (size) */
    dc_amp = b[0][0] - prev_dcval;
    dc_amp = amp_to_vli((uint32_t) dc_amp, &size, VLI_DC);

    /* Add DC huffman code (VLC) for size, then VLI for amplitude */
    codelen = get_dc_code_len(size);
    codeword = dc_lht_codes[size];
    bits_left = add_bits2buf(bits_left, &buf, codeword, codelen);
    bits_left = add_bits2buf(bits_left, &buf, dc_amp, size);

    uint8_t* p = codes - 1;     // point to start of codes
    uint16_t* a = amps - 1;     // point to start of amps
    while (p != code_i) {       // loop until we've processed all codes

        /* Add AC huffman code (VLC) for size, then VLI for amplitude */
        code = *(++p); // index into table
        codelen = get_ac_code_len(code);
        codeword = ac_lht_codes[code];
```

```
        bits_left = add_bits2buf(bits_left, &buf, codeword, codelen);

        // skip EOB and ZRL codes (because they don't have amps)
        if (code != 0 && code != 151) {
            amp = (*(++a));
            size = amp >> 12;
            amp &= 0x0fff;
            bits_left = add_bits2buf(bits_left, &buf, amp, size);
        }
    }
    free(codes);
    free(amps);


    // Return sizeof buf buffer (in bytes).
    return (buf - buf_start +1) * 2;
}
```

JPEG Huffman Variable Length Decoder

```
/*
 * Decodes a given huffman encoded 8x8 block.
 *
 * @prev_dcval: The previous DC value that came before this block.
 * @buf: The 16bit word buffer that holds the huffman encoded bitstring.
 * @outb: The 8x8 output array to which the decoded DCT will be written to.
 *
 * Returns the amount of data in bytes written to buf.
 */
static uint32_t
dehuff(int32_t prev_dcval, const uint8_t* restrict buf8, int32_t outb[8][8])
{
    const uint16_t* restrict buf = (const uint16_t* restrict) buf8;
    uint8_t bitoffset = 0;
    uint16_t word = 0;
    int len = 0;
    int idx = 0;
    int rl = 0;
    int size;
    int32_t amp = 0;
    uint8_t lin_idx = 1;
    uint8_t dig_row;
    uint8_t dig_col;
    const uint16_t* restrict buf_s = buf;

    memset(outb, 0, 64 * sizeof(int32_t));

    // Read DC codeword
    bitoffset = get_word_from_buf(
               (const uint16_t** restrict) &buf,
               &word, bitoffset);
    size = dc_size_from_codeword(word, &len);
    bitoffset += len;

    // Read DC amplitude
    bitoffset = get_word_from_buf(
```

```
                (const uint16_t** restrict) &buf,
                &word, bitoffset);
    bitoffset += size;
    outb[0][0] = prev_dcval + vli_to_amp(word, size);

    int codes_n = 127;
    while (codes_n-- != 0) {
        // Get huff codeword
        bitoffset = get_word_from_buf(
                    (const uint16_t** restrict) &buf,
                    &word, bitoffset);
        idx = ac_lht_idx_from_codeword(word, &len);
        bitoffset += len;

        // If it's zero, we're done (EOB)
        if (idx == 0) break;

        // Extract runlength and amp size from index
        rl = idx / 10;
        size = idx % 10;
        if (size == 0) {
            rl--;
            size = 10;
        }

        amp = 0;
        if (idx != 151) {
            // If not a Zeroblock (ZRL), get amplitude
            bitoffset = get_word_from_buf(
                        (const uint16_t** restrict) &buf,
                        &word, bitoffset);
            amp = vli_to_amp(word, size);
            bitoffset += size;
        }

        // at this point we've decoded the next zero runlength and
        // the amplitude following this runlength. Skip RL indexes
        // and then place amplitude.
        lin_idx += rl;
        if (lin_idx >= 64) break;
        dig_row = row_lin2diag[lin_idx];
        dig_col = col_lin2diag[lin_idx];
        outb[dig_row][dig_col] = amp;
        lin_idx++;
    }
    if (bitoffset >= 16) buf++;
    return (buf - buf_s + 1) * 2;
}
```

ESP32 TCP Packet to SPI Relay Code

```
while (1) {

            // Get TCP Data
            int to_recv = next_fr_size != 0 && next_fr_size < SPI_MTU? next_fr_size : SPI_MTU;
```

```
#ifdef DEBUG_FRAME
            printf("Tryign to recv %d\n", to_recv);
#endif
            int recvd = 0;
            char* write_ptr_copy = write_ptr;
            do {
                int r = recv(sock, write_ptr_copy, to_recv - recvd, 0);
                write_ptr_copy += r;
                recvd += r;
#ifdef DEBUG_FRAME
                printf("Received %d. Have %d/%d", r, recvd, to_recv);
#endif
            } while (recvd != to_recv);

            // Check if a new frame started
            if (strcmp(write_ptr, sig) == 0) {
                got_first_frame = 1;

                // If the amount of data we need to transmit for
                // current frame is not zero, we are lagging.
                if (next_fr_size > 0) {
                    // Drop the next frame.
                    printf("Dropping Frame...\n");
                }
                else {
                    // Get the next frame size from the rxbuffer which looks
                    // like "FRSTART\0XX[...]" were XX is a UINT16 and [...]
                    // is the frame data
                    next_fr_size = (int) *((uint16_t*) (write_ptr + strlen(sig) + 1));
                    next_fr_size += strlen(sig) + 1 + sizeof(uint16_t);
                    spi_consumed_fr_size = next_fr_size;

                    // Reset Read/Write pointers
                    read_ptr = rx_buffer;
                    write_ptr = rx_buffer;

                    // Collect Debug information
#ifdef DEBUG_FRAME
                    gettimeofday(&fr_et, NULL);
                    float elapsed_ms = (((fr_et.tv_sec - fr_st.tv_sec) * 1000000)
                            + (fr_et.tv_usec - fr_st.tv_usec)) / 1000.0f;
                    gettimeofday(&fr_st, NULL);

                    printf("Next frame size %d! Elapsed since last: %0.2fms\n",
                            (int) next_fr_size, elapsed_ms);
                    int sum = 0;
                    for (int i = 0; i < debug_frame_i; i++) {
                        sum += debug_frame[i];
                    }
                    printf("Frame Huffman Coded DCT Checksum: %d\n", sum);
                    debug_frame_i = 0;
#endif
                }
            }
            if (got_first_frame == 0) {
                write_ptr = rx_buffer;
                continue;
            }

            // move write_ptr to next block
```

```
            write_ptr += recvd;
            recvd_total += recvd;
            if (write_ptr - rx_buffer > TCP_RX_SIZE)
                printf("Buffer ofverflow\n");

            next_fr_size -= recvd;

            // Check if current frame still needs bytes to be sent over SPI.
            if (spi_consumed_fr_size > 0) {
#ifdef DEBUG_FRAME
                printf("spi_consumed_fr_size: %d\n", spi_consumed_fr_size);
#endif
                if (spi_consumed_fr_size >= SPI_MTU &&
                        write_ptr - read_ptr >= SPI_MTU) {
                    // We have enough data, so
                    // write the data that came in into the spi buffer
                    memcpy(spibuf, read_ptr, SPI_MTU);

                    // send the chunk over SPI
                    spi_slave_transmit(VSPI_HOST, &t, portMAX_DELAY);

                    read_ptr += SPI_MTU;
                    spi_consumed_fr_size -= SPI_MTU;
#ifdef DEBUG_FRAME
                    int sum = 0;
                    for (int i = 0; i < SPI_MTU; i++) sum += read_ptr[i];
                    printf("ChecksumMTU: %d\n", sum);
                    memcpy(debug_frame + debug_frame_i, spibuf, SPI_MTU);
                    debug_frame_i += SPI_MTU;
#endif
                }
                else if (spi_consumed_fr_size < SPI_MTU &&
                        write_ptr - read_ptr >= spi_consumed_fr_size) {
                    // empty the buffer and pad with zeros
                    memset(spibuf, 0, SPI_MTU);
                    memcpy(spibuf, read_ptr, spi_consumed_fr_size);

                    // send the last chunk over SPI
                    spi_slave_transmit(VSPI_HOST, &t, portMAX_DELAY);

                    read_ptr += spi_consumed_fr_size;
                    spi_consumed_fr_size = 0;

                    write_ptr = rx_buffer;
#ifdef DEBUG_FRAME
                    int sum = 0;
                    for (int i = 0; i < spi_consumed_fr_size; i++) sum += read_ptr[i];
                    printf("ChecksumMTU_Fin: %d\n", sum);
                    memcpy(debug_frame + debug_frame_i, spibuf, SPI_MTU);
                    debug_frame_i += SPI_MTU;
#endif
                }
                else {
                    // Not enough TCP data available
                }
            }
            else {
                // No new data needed being written, we're chillin until next frame
            }
```

```
#if BENCHMARK
            // Error occured during receiving
            if (recvd_total > 10e6) {
                gettimeofday(&et, NULL);
                int elapsed = ((et.tv_sec - st.tv_sec) * 1000000) + (et.tv_usec - st.tv_usec);
                double elapsed_seconds = (double) elapsed / 1.0e6;
                ESP_LOGI(TAG, "Bytes Received: %d", recvd_total);
                ESP_LOGI(TAG, "Elapsed Seconds: %f seconds", elapsed_seconds);
                ESP_LOGI(TAG, "Speed: %f KB/s",
                        (double) recvd_total / elapsed_seconds / 1024);

                ESP_LOGE(TAG, "recv failed: errno %d", errno);
                break;
            }
            // Data received
            else {
                rx_buffer[recvd] = 0; // Null-terminate whatever we received and treat like a
string
                recvd_total += recvd;
                //ESP_LOGI(TAG, "Received %d bytes.", recvd_total);
            }
#endif

        }

        if (sock != -1) {
            ESP_LOGE(TAG, "Shutting down socket and restarting...");
            shutdown(sock, 0);
            close(sock);
        }
    }
```

PSoC6 CM4 Mainloop

```
while (1) {
               /*
                * Check if:
                * 1. ESP32 has data and is waiting to accept an SPI transaction.
                * 2. We are in the vertical blanking zone OR we've already started
                *             transferring the first chunk of a frame and we have clearance
                *             to get the next.
                */
               if (spi_ready /*&& (port9_fb[0] == 0xff || spi_ok != 0)*/) {

                      spi_ready = 0;

                      /* If we haven't encountered a frame start symbol, write to beginning of
buffer */
                      if (in_frame == 0) huffman_data_i = 0;

                      /* Do SPI transaction */
                      cyhal_spi_transfer(&mSPI,
                                    transmit_data,
                                    SPI_MTU,
                                    huffman_data + huffman_data_i,
                                    SPI_MTU,
                                    0x00
```

```
                        );

                        /* Check if we're not in a frame and have encountered a frame start
symbol */
                        if (in_frame == 0 && strcmp((char*) huffman_data, sig) == 0) {
                                huffman_data_i = SPI_MTU;
                                spi_ok = 1;           // Give clearance to pull SPI as quickly
as possible
                                                      // (without having to wait for
vertical blanking)
                                in_frame = 1;  // We are now receiving subsequent frame data.
                        }
                        else {
                                /* Check if we've received an SPI_MTU Block ending in zeros. If
this is the
                                 * case, then we know the frame is done. Note that it would
probably be better
                                 * to parse the size of the frame when encountering a start
symbol instead of
                                 * relying on this.
                                 */
                                if (huffman_data[huffman_data_i + SPI_MTU - 3] == 0 &&
                                                huffman_data[huffman_data_i + SPI_MTU - 2] == 0 &&
                                                huffman_data[huffman_data_i + SPI_MTU - 1] == 0) {

                                        in_frame = 0;
                                        spi_ok = 0;
                                        // frame is done. Process it.
#ifdef DEBUG
                                        uint32_t frame_dt = micros() - last_fr_time;
                                        last_fr_time = micros();
                                        printf("Time since last frmae: %dms\n\r", (frame_dt /
1000));

                                        int sum = 0;
                                        for (int i = 0; i < huffman_data_i + SPI_MTU; i++)
                                                sum += huffman_data[i];
                                        printf("Huffman checksum: %d\n\r", sum);
                                        uint32_t start_t = micros();
#endif

                                        /* Huffman Decode YCbCr channels */

                                        // +10 because we have a 10 byte signature at the
beginning.
                                        huffman_data_i = decompress_channel(ychannel,
                                                        huffman_data+10, IMG_WIDTH, IMG_HEIGHT);
                                        huffman_data_i += decompress_channel(cbchannel,
                                                        10+huffman_data+huffman_data_i,
CHROMA_WIDTH, CHROMA_HEIGHT);
                                        decompress_channel(crchannel,
                                                        10+huffman_data+huffman_data_i,
CHROMA_WIDTH, CHROMA_HEIGHT);

                                        /* YCbCr convert to RGB and draw fill the shared VGA
buffer */
                                        dec_ycrcb(ychannel, cbchannel, crchannel,
                                                        SUBSAMPLE_CHROMA, 0, 10, IMG_WIDTH,
IMG_HEIGHT);
#ifdef DEBUG
```

```
                                    uint32_t end_t = micros();
                                    printf("Decompression time: %dms\n\r",
(end_t-start_t)/1000);
#endif
                                }
                                /* If frame didn't end, next frame will be written to this idx */
                                else huffman_data_i += SPI_MTU;
                        }
                }
        }
```

VGA and DMA Initialization code (CM0+, PSOC6)

```
void
init_dma(DW_Type* dma_hw, int channel, cy_stc_dma_descriptor_t* descriptor,
                const cy_stc_dma_descriptor_config_t* dconfig, const
cy_stc_dma_channel_config_t* channel_config,
                uint32_t* _port_addr, uint32_t* src_addr)
{
        cy_en_dma_status_t dma_init_status;

         /* Init DAM Descriptor */
    dma_init_status = Cy_DMA_Descriptor_Init(descriptor, dconfig);
    if (dma_init_status!=CY_DMA_SUCCESS) { handle_error(); }

    /* Initialize DMA channel */
    dma_init_status = Cy_DMA_Channel_Init(dma_hw, channel, channel_config);
    if (dma_init_status!=CY_DMA_SUCCESS) { handle_error(); }

    /* Set source and destination for descriptor 1 */
    Cy_DMA_Descriptor_SetSrcAddress(descriptor, (uint32_t *) src_addr);
    Cy_DMA_Descriptor_SetDstAddress(descriptor, (uint32_t *) _port_addr);

    /* Setup Channel */
    Cy_DMA_Channel_SetDescriptor(dma_hw, channel, descriptor);
    Cy_DMA_Channel_SetPriority(dma_hw, channel, 0UL);
    Cy_DMA_Channel_Enable(dma_hw, channel);

}

void
init_vga()
{
        /* Find Port register address */
    port9_addr = (uint32_t*) Cy_GPIO_PortToAddr(9);
    port8_addr = (uint32_t*) Cy_GPIO_PortToAddr(8);

    /* Initialise DW0 and DW1 DMA controllers */
    init_dma(DMA1_HW, DMA1_CHANNEL, &DMA1_Descriptor_0, &DMA1_Descriptor_0_config,
                &DMA1_channelConfig, (uint32_t*) port9_addr, (uint32_t*) port9_fb);
    init_dma(DMA2_HW, DMA2_CHANNEL, &DMA2_Descriptor_0, &DMA2_Descriptor_0_config,
                &DMA2_channelConfig, (uint32_t*) port8_addr, (uint32_t*) port8_fb);

    /* Create DMA Line interrupt service routine
     * This interrupt reloads the DMA controllers with updated addresses.
     */
```

```
        cy_stc_sysint_t DMA_LINE_DONE_ISR =
        {
                        .intrSrc      = (IRQn_Type) NvicMux1_IRQn,
                        .cm0pSrc         = HSYNC_T_IRQ,       // Fire on each HSYNC pulse
                        .intrPriority = 0u,
        };


        /* Disable all other IRQs for safety */
        for (int i = 0; i < 0xff; i++) __NVIC_DisableIRQ(i);

    /* Initialize and enable the interrupt */
    Cy_SysInt_Init(&DMA_LINE_DONE_ISR, &dma_line_done);
    __NVIC_EnableIRQ(DMA_LINE_DONE_ISR.intrSrc);

    /* Start PWM Timers which will trigger DMA
     *
     * HSYNC_T  -- Generates Hsync pulse that goees into the VGA monitor
     * HBLANK_T  -- A slightly delayed Hsync pulse that goes off after Hsync goes low.
     *                         When this PWM pulses, DW0 gets triggered to start the next
horizontal
     *                         line transfer. This creates the back porch.
     * HBLANK_T2 -- A copy of HBLANK_T. This triggers DW1 transfer. (We need separate
     *                         TCPWM blocks because each only connects to one DataWire)
     */
    Cy_TCPWM_PWM_Init(HSYNC_T_HW, HSYNC_T_NUM, &HSYNC_T_config);
    Cy_TCPWM_PWM_Enable(HSYNC_T_HW, HSYNC_T_NUM);

    Cy_TCPWM_PWM_Init(HBLANK_T_HW, HBLANK_T_NUM, &HBLANK_T_config);
    Cy_TCPWM_PWM_Enable(HBLANK_T_HW, HBLANK_T_NUM);

    Cy_TCPWM_PWM_Init(HBLANK_T2_HW, HBLANK_T2_NUM, &HBLANK_T2_config);
    Cy_TCPWM_PWM_Enable(HBLANK_T2_HW, HBLANK_T2_NUM);

    Cy_DMA_Enable(DMA1_HW);
    Cy_DMA_Enable(DMA2_HW);

    /* Create and register vertical sync ISR */
    cy_stc_sysint_t VSYNC_RISE_ISR =
    {
                .intrSrc      = (IRQn_Type) NvicMux0_IRQn,
                        .cm0pSrc         = (IRQn_Type) VSYNC_T_IRQ,
                        .intrPriority = 0u,
    };

    /* Initialize and enable the interrupt */
    Cy_SysInt_Init(&VSYNC_RISE_ISR, &vsync_rise_isr);
    NVIC_EnableIRQ(VSYNC_RISE_ISR.intrSrc);


    Cy_TCPWM_PWM_Init(VSYNC_T_HW, VSYNC_T_NUM, &VSYNC_T_config);
    Cy_TCPWM_PWM_Enable(VSYNC_T_HW, VSYNC_T_NUM);

    Cy_TCPWM_TriggerStart(VSYNC_T_HW, VSYNC_T_MASK);

    /* Magic number */
    CyDelayCycles(470);

    Cy_TCPWM_TriggerStart(HBLANK_T_HW, HBLANK_T_MASK);
```

```
    CyDelayUs(31);
    CyDelayCycles(92);
}
```

[Crypto] Python Generalized Permutation Bitmask Generator

```
def get_masks(start, end):
    """
    Given a starting permutation and an ending permutation, compute the
    masks for it recursively.
    @start: array with elements. For example, ['A', 'B', 'C', 'D']
    @end:   array with permutated elemented. For example, ['D', 'C', 'B', 'A']
    """
    n = len(start)
    if set(start) != set(end):
        return print("Error! Not a permutation!!")

    # base case. A simple Mux.
    if n == 2:
        return [[0, 0]] if start[0] == end[0] else [[0, 1]]

    start_s1 = start[0:n//2]
    start_s2 = start[n//2:n]
    end_s1 = end[0:n//2]
    end_s2 = end[n//2:n]

    # These are lists of tuples, each tuple represents a Mux
    start_s = [(x, y) for x, y in zip(start_s1, start_s2)]
    end_s = [(x, y) for x, y in zip(end_s1, end_s2)]

    # Masks keep track of which Muxes have been flipped
    mask_start = [0] * (n // 2)
    mask_end = [0] * (n // 2)

    # It is our goal to make set(start_s,0) == set(end_s,0). To do this,
    # we assert that the first mux position at start_s[0] is correct.
    # Then all the other mux positions follow uniquely. We can solve this
    # using a ping pong game. We know start_s[0][0] is correct, hence the
    # we need start_s[0][0] to be in the ending subset. Thus, we "want"
    # start_s[0][0]. In the ending set, once we set start_s[0][0], we know
    # that the other elemnt that gets replaced by start_s[0][0] in end_s
    # is "unwanted" in start_s. So we go back to start_s and flip again.
    s1, s2 = 0, 42
    unwanted = None
    wanted = start_s[0][0]
    past_wanted = set()     # keep tab on past wanted sets, to avoid cycles
    pings = 0

    def ping_back(arr, mask):
        nonlocal wanted, unwanted, pings
        pings += 1

        # Check if we've already encountered this item. If so, we're
        # free to assert that any next item is in the correct position
        if wanted in past_wanted:
            for x in start_s:
                if x[0] not in past_wanted:
```

```
                wanted = x[0]

        for i in range(len(arr)):
            if arr[i][0] == unwanted:
                # A unwanted element is in a wanted position. Cross Now.
                arr[i] = arr[i][::-1]
                mask[i] = (mask[i] + 1) % 2
                wanted = arr[i][0]
                unwanted = None
                break
            if arr[i][0] == wanted:
                # A wanted element is in a wanted position. Hence, it's
                # compliment must be unwanted.
                unwanted = arr[i][1]
                past_wanted.add(wanted)
                wanted = None
                break
            if arr[i][1] == unwanted:
                # An unwanted element is in an unwanted position. Hence,
                # its complimemt must be wanted.
                wanted = arr[i][0]
                unwanted = None
                break
            if arr[i][1] == wanted:
                # A wanted element is in an unwanted position. Cross Now.
                arr[i] = arr[i][::-1]
                mask[i] = (mask[i] + 1) % 2
                unwanted = arr[i][1]
                past_wanted.add(wanted)
                wanted = None
                break

    # while set([x[0] for x in start_s]) != set([x[0] for x in end_s]), play pingpong
    # the upper bound to this is n//2
    for _ in range(n//2):
        # play the ping pong game until the sets are equal
        ping_back(end_s, mask_end)
        ping_back(start_s, mask_start)

    # generate the final subwords based on the correct sets
    start_s1 = [x[0] for x in start_s]
    start_s2 = [x[1] for x in start_s]
    end_s1 = [x[0] for x in end_s]
    end_s2 = [x[1] for x in end_s]

    # create the first layer mask. Note that only the latter half matters, as 1
    # signfiies the destination index to go into when we do the delta shift.
    # Everything else (higher half) should be zero
    zero_subword = [0] * (n//2)
    masks = [zero_subword + mask_start]

    # recurse deeper to generate all the inner layers
    mask_sublayer_left = get_masks(start_s1, end_s1)
    mask_sublayer_right = get_masks(start_s2, end_s2)

    # interweave the separate sublayer masks to create the full n-length
    # masks for each sublayer
    masks += [x + y for x, y in zip(mask_sublayer_left, mask_sublayer_right)]

    # create the mask for the final layer.
```

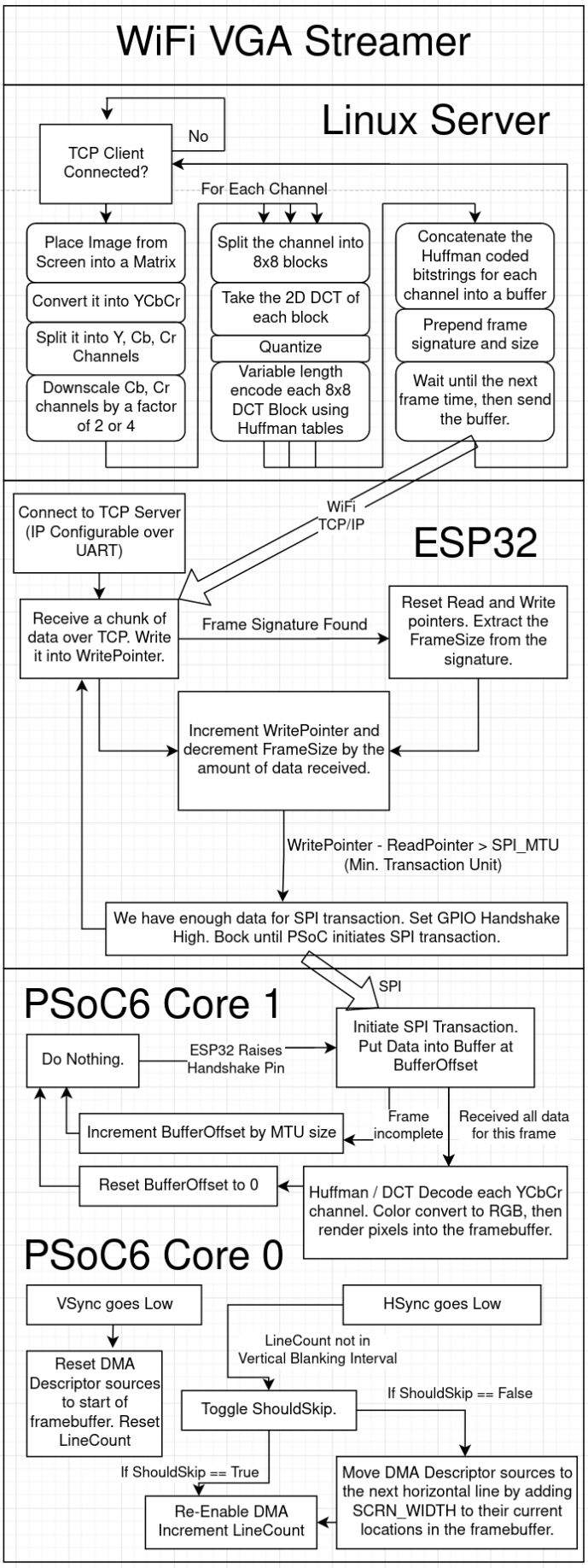
```
    masks.append(zero_subword + mask_end)

    return masks
```

My DES Implementation (Which relies on my generated masks)

```
#include "des.h"
#include "des_data.h"

uint64_t key_scheduler(uint8_t iter, uint64_t* cd);
uint32_t apply_f(uint32_t r, uint64_t k);

/**
 * encrypt_des_blk - Encrypt a 64-bit block with DES.
 * @block: A 64-bit block of data
 * @key:   A 64-bit key
 */
uint64_t
encrypt_des_blk(uint64_t block, uint64_t key)
{
    block = perm_bitstring(block, 64, perm_mask_ip);

    uint32_t l_i = (0xffffffff00000000 & block) >> 32;
    uint32_t r_i = (0x00000000ffffffff & block);
    uint64_t cd_i = perm_bitstring(key, 64, perm_mask_pc1) & 0x00ffffffffffffff;
    uint32_t tmp;

    key = 0;
    int i;
    for (i = 0; i < 16; i++) {
        key = key_scheduler(i + 1, &cd_i);  // compute Key_i
        tmp = r_i;                          // go to next
        r_i = l_i ^ apply_f(r_i, key);      // layer in the
        l_i = tmp;                          // feistel network
    }

    block = (((uint64_t) r_i) << 32) | ((uint64_t) l_i);
    return perm_bitstring(block, 64, perm_mask_ipi);
}

#define smask1   0x0100000010000000 /* set bit D[n+1] (C[0]) and bit C[n+1] */
#define smask1i  0x00fffffeffffffff /* discard C[0] and irrelevant bits     */
#define smask2   0x0300000030000000 /* set D[n+2:n+1] (C[1:0) and C[n+2:n+1]*/
#define smask2i  0x00fffffcffffffff /* discard C[1:0] and irrelevant bits   */

/** key_scheduler - Given the current iteration CD blocks, update CD
 *                  and  return the next key.
 */
inline uint64_t
key_scheduler(uint8_t iter, uint64_t* cd)
{
    // CD = 56bit number. C = 28 high bits. D = 28 low bits.
    uint64_t out = *cd;

    if (iter == 1 || iter == 2 || iter == 9 || iter == 16) {
        // rotate C and D left by 1
```

```
        out = (out << 1);
        out = (out & smask1i) | ((out & smask1) >> 28);
    }
    else {
        // rotate left by 2
        out = (out << 2);
        out = (out & smask2i) | ((out & smask2) >> 28);
    }

    // update C and D blocks
    *cd = out;

    // return the 48 bit
    out = perm_bitstring(out, 64, perm_mask_pc2);
    out &= 0x0000ffffffffffff;

    return out;
}


/** apply_f - Apply the f(x), where f is defined as in DES Spec.
 */
inline uint32_t
apply_f(uint32_t r, uint64_t k)
{
    int8_t   si;
    uint8_t  cmd, val;
    uint32_t out = 0;
    uint8_t  idx;

    // MSB of r is 1st position. 32 position is LSB.
    uint64_t t = (uint64_t) r;
    uint64_t re = (t & 0xf8000000) << 15 |
                  (t & 0x1f800000) << 13 |
                  (t & 0x01f80000) << 11 |
                  (t & 0x001f8000) << 9  |
                  (t & 0x0001f800) << 7  |
                  (t & 0x00001f80) << 5  |
                  (t & 0x000001f8) << 3  |
                  (t & 0x0000001f) << 1  |
                  (t & 0x00000001) << 47 |
                  (t >> 31);

    uint64_t x = (uint64_t) re ^ k;

    /**
     *  Do the sblock calculations. Convert 48 bit --> 32 bit output.
     */
    for (si = 7; si >= 0; si--, x >>= 6) {
        /* Get 6 bits as cmd word. B8, ..., B1 */
        cmd = x & 0x3f;

        /* Compute the S-Block byte-level index into the table */
        idx = ((cmd >> 2) | (cmd << 4)) & 0x1f;

        /* Find S8, ..., S1 and add byte-level idx, then fetch*/
        val = sblocks[(si << 5) + idx];

        /* If cmd uses upper nibble, shift it */
        if (cmd & 2) val >>= 4;
```

```
        /* Shift and or value into out */
        out >>= 4;
        out |= val << 0x1c;
    }

    return perm_bitstring(out, 32, perm_mask_p);
}
```


WiFi VGA Streamer
Linux Server
TCP Client Connected?
No
For Each Channel
Place Image from Screen into a Matrix
Convert it into YCbCr
Split it into Y, Cb, Cr Channels
Downscale Cb, Cr channels by a factor of 2 or 4
Split the channel into 8x8 blocks
Take the 2D DCT of each block
Quantize
Variable length encode each 8x8 DCT Block using Huffman tables
Concatenate the Huffman coded bitstrings for each channel into a buffer
Prepend frame signature and size
Wait until the next frame time, then send the buffer.
WiFi TCP/IP
ESP32
Connect to TCP Server (IP Configurable over UART)
Receive a chunk of data over TCP. Write it into WritePointer.
Frame Signature Found
Reset Read and Write pointers. Extract the FrameSize from the signature.
Increment WritePointer and decrement FrameSize by the amount of data received.
WritePointer - ReadPointer > SPI_MTU (Min. Transaction Unit)
We have enough data for SPI transaction. Set GPIO Handshake High. Bock until PSoC initiates SPI transaction.
SPI
PSoC6 Core 1
Do Nothing.
ESP32 Raises Handshake Pin
Initiate SPI Transaction. Put Data into Buffer at BufferOffset
Frame incomplete
Received all data for this frame
Increment BufferOffset by MTU size
Reset BufferOffset to 0
Huffman / DCT Decode each YCbCr channel. Color convert to RGB, then render pixels into the framebuffer.
PSoC6 Core 0
VSync goes Low
HSync goes Low
LineCount not in Vertical Blanking Interval
Reset DMA Descriptor sources to start of framebuffer. Reset LineCount
Toggle ShouldSkip.
If ShouldSkip == False
If ShouldSkip == True
Move DMA Descriptor sources to the next horizontal line by adding SCRN_WIDTH to their current locations in the framebuffer.
Re-Enable DMA Increment LineCount