FW.Hardijzer.nl

[illegible]

96×48 full-color LED Matrix

LED Panels

A few years ago I built a red-only 32 pixels high, 96 pixels wide LED Matrix, and due to all the positive responses I sought out to do it again the year after with a bigger better matrix. I did some research into affordable solutions, and as usual ended up with Chinese vendors. I got my hands on about 10 32×16 RGB LED panels with a 1cm pixel pitch, and a HUB75 connection, quite similar to the ADAFruit 32×16 matrix. ADAFruit had a bunch of information on them, and there are several other places where they're being used, so I figured I'd give it a shot. I even bought a Digilent Basys 2 FPGA development board, as these boards are apparently best driven by an FPGA, and I was willing to pick that up.

However, when I started working on it initially, I found out that the Basys 2 did not even have enough memory for 24-bit color, and with my microcontroller of choice at that moment (TI Stellaris) I did not get more than 16-bit color without flickering. While the panels looked fun, with 16-bit color everything still looked very basic, and very limited. Definitely not the neat full-color panels I had in mind. I figured I'd either have to shell out for a much more expensive FPGA, or buy one of the purpose-built LED drivers from china, both of which were out of my budget for this project. The panels disappeared in a box somewhere, only to be stumbled upon at least 2 year later.

Result

Here's a quick video of the final results:

Project revival

Seeing as STM32 chips are much more powerful than Arduino's or AVRs, they've been appearing on loads of Chinese development boards, and are relatively inexpensive, I've collected quite a few STM32 boards in my "to play with sometime" box of electronic gadgets. I'd gotten my first introduction to ARM chips on the TI Stellaris Launchpad, and seeing the rise in popularity of STM32 chips, I was itching to get my feet wet. I'd been writing some simple LED blinking on an STM32F1 discovery board, played with the DMA, and it was time for something bigger. Going through the box of electronic playthings, I stumbled upon the LED panels, and figured I'd give them one more try.

Either I'd get them working this time, or would get rid of them as they were quite big and no use like this anyway.

Basic driving of a HUB75 LED panel

MY HUB75 consists of basically two identical parts stacked on top of each other. I'll describe one part of 32×8, and you should be able to extrapolate to the 32×16 by yourself 😉

The connection of HUB75 has 6 data pins (R1, G1, B1, R2, G2, B2), 4 row select pins (A,B,C,D), and 3 control pins (CLK, STB/LAT, OE).

Of these the OE, or Output Enable pin is the easiest, it simply controls if anything lights up at all. If you pull this to ground, LEDs might light up, if you leave it high, nothing wil light up.

Second easiest are the A, B, C, D pins. In my panels, which are described as 1/8th scan, only the first three of these are used. These pins select which row is active. If for example A, B, and C are all low, the first row might display something, and all other rows are off. If A is high, and B and C are low, the second row might display something, whereas the rest will again be off. You're supposed to switch between all 8 rows so fast that to the human eye it seems like they are all on at once.

Next up are the data pins, and the CLK and STB/LAT pins. Inside the panels are, for each channel, two 16-bit shift registers.Everytime the CLK pin goes high a bit is shifted in, and once the STB/LAT pin goes high, anything in the shift register is displayed on the LEDs. So let's say you shift in 100…000 on the R1 line, and set ABC to be 000, the top-left LED will light up RED. If you then change ABC to be 100, the top-left LED will turn off again the the one below it will light up.

So, basically, you clock in data, disable the output, latch in the data, set the row selectors, enable the output, display this data for a while, and then do the next row. Ideally you'd clock in the data for the next row while you're displaying the former, but let's not get ahead of ourselves.

Back to the bigger 32×16 panel: R1,G1,B1 are for the top half, R2, G2, B2 for the bottom half. Also, the panels have a HUB75 output on the other end that allows you to chain things. So if you chain 2 panels, you'd have to shift in 64 bits each time.

More colours: PWM

At this point you might be wondering "But hey, you mentioned full-colour, but up until now I've only seen you discuss on or off?". That's right, to get different brightness values, we should use pulse-width modulation. So at this point not only should we drive this thing fast enough so you don't notice there's only one row on at a time, we should do it so fast that we can vary how often a pixel is on, without having the human eye notice.

More on HUB75 driving

For more information on how precisely to drive a HUB75 display, the following articles greatly helped me understand it:

- Adafruit RGB LED Matrix at Rayslogic.com
- RGB LED Panel Driver Tutorial by Glen Akins

Actually getting it to work: different approaches

Bit-banging with naive PWM

My first goal was to get the panels to display anything. A simple loop to just read each pixels R, G, and B values, do a comparison against a looping value from 1 to 255, and shift out a bit. Initially on the STM32F1 Discovery this worked, but I could only get up to 4 bit grayscale on a single panel without noticeable flickering, while the goal was to get something at least 24bpp on a lot more panels. Switching to the STM32F4 Discovery, and with it increasing the clockspeed from 24mhz to 168mhz, gave me some extra leeway, but at just a single panel 8-bit grayscale already gave visible flickering on my phone's camera.

There were a couple of problems with this approach:

- There was some major ghosting going on between rows. Apparently this is a common problem with these kinds of matrices, and is easily solved by limiting the number of row-switches to a minimum.
- 6-bit grayscale is nowhere near full-colour.
- The processor has a full-time job driving this display, which means that there are no clock cycles left to actually do some animating.

Binary code modulation, precalculating

The first fix is to switch from naive PWM to binary code modulation. With normal PWM I refresh the display 255 times, and if a pixel was supposed to be 127/255 lit, I'd turn it on for the first 127 times, and leave it off for the rest. Binary code modulation is a better fit, instead of 255 equally long display periods, I use just 8. One of let's say x, one of 2x, one of 4x, and so on. For a 127/255 lit LED, we'd split up the 127 in it's binary counterparts: 1 + 2 + 4 + 8 + 16 + 32 + 64, and turn it on in the periods corresponding to those, and off during the others. The LED ends up being lit for the same amount of time, yet with binary code modulation I only have to shift out data 8 times, and switch rows 8 times.

Furthermore, seeing as now for each row there are only 8 different streams being pushed out, I could precompute them, thus cutting the inner loop of shifting the data outback from about 20 clockcycles per bit to 5.

This finally pushed me near 8-bit grayscale, but 9-bit unfortunately still gave flickering on my phone camera (but barely noticeable to my eye).

Thoughts on this method:

- The ghosting wasn't nearly as bad, but still visible. Something that I can live with, but I'd rather get rid of it at some point.
- While 8-bit grayscale sounds good, seeing as your monitor is 8-bit grayscale too, in reality it isn't at all. The human eye responds to brightness logarithmically, and normally your monitor adjusts for that. Our duty-cycle algorithms don't, which means we've got only a few different intensities in the lower end, and a lot in the higher end. We should aim for at least 9-bit grayscale, to get something resembling full colour.
- Due to the timing, there are now chunks of time where the processor isn't doing anything, but as it's scattered all over the place, it's hard to do anything useful with it.
- The time of the lowest significant bit was at this point limited by the time it took to shift out an entire row.

FSMC + DMA + Output Enable

To free up the processor, it was time to use Direct Memory Access: a peripheral that can pump a block of memory to a peripheral, without the processor having to do anything. There'd be a DMA interrupt (whenever it's ready copying the data), and a timer interrupt, and once these both have been fired the processor would do a little bit of work to set up the next bit of data to shift out. Apart from that, it was free to do anything it wanted, like create a nice animation.

I could use the DMA to write directly to the GPIO, but that would mean embedding the clock signal in the data, and doubling the memory required. Luckily the STM32F4 has an FSMC , or Flexible Static Memory Controller , peripheral. Most memory chips have a few address lines, some data lines, and a write and read strobe. Whenever you toggle the read strobe, it'll read whatever is at the address pointed to by the address lines, and put it on the data lines. Whenever you toggle the write strobe, it'll write whatever is on the data lines to the address on the address lines. The FSMC was made specifically to drive memory like this, by mapping an address range to it. The STM32's main processor can just write or read from an address, and the FSMC will talk to the memory for it. For our matrix, we need a few data lines, and a clock pulse, which maps nicely on the data lines and the write strobe. So we can set up the FSMC, write a couple of bytes to the address range for it, and the FSMC will shift it out for us and take care of the clock signal. Nice!

Another little trick I thought up is to actually use the Output Enable pin for something. Shifting out the data takes a set amount of time, and initially I thought that with the binary code modulation the least significant bit couldn't be shorter than the time that shifting out took. However, what if we could turn on the row for only a fraction of the time it takes to shift out the data. Yes, the display won't be at absolute maximum brightness, but it would allow us to make the time for the LSB much smaller, and as such for the entire refresh rate to be much smaller. It wasn't difficult to wire the timer counter-compare pin to the output enable pin on the HUB75, and the results were stunning. I could finally do 10-bit grayscale!

Some setbacks, though:

- The FSMC was very very fickle. The write strobe was idle-high, while the clock was supposed to be idle-low. Usually it worked, but whenever I would do anything over any other pins, like let's say receive data over UART, there would be major glitches appearing.
- While I could theoretically now set the LSB-duration really small and allow for more bit grayscale, the lower I set it, the more the ghosting

reappeared.

Final method: DMA + GPIO + Loop fix-up

While in theory the FSMC was perfect for this, and maybe with a little inverter on the clock pin it still might, I still opted to ditch it in favour of just DMA'ing to the GPIO pins. This doubled the memory consumption, but seeing as it's much much more stable, I think that's worth the trade-off.

Finally I also re-organized the loop. Initially for each bit in the bit code modulation, I would loop through all rows, each time introducing a teeny tiny bit of ghosting on the next row. Instead, I opted to loop over the rows, and then do all the bits at once. This cut the number of times I looped over the rows drastically, and the ghosting is now no longer visible with the naked eye. I'm sure it's still there, but at least now it's so minor it no longer bothers me.

Displaying something useful

At this point the display would function fine as a sort of digital photo frame, seeing as I could display an image and keep it that way, but actually displaying video was not really possible. Seeing as almost everything at this point is done in DMA, there are plenty of clock cycles to generate images, but that's no what I wanted: I wanted to be able to drive it over the network.

Raspberry Pi + SPI

The first approach was using a raspberry pi. I set up a simple double-buffer on the STM32F4, and set up DMA to copy anything it received over SPI to the buffer, and once it was filled to actually display that buffer. The raspberry pi would then do all the precalculation, and simply spit it out over SPI. This worked great, but there was one setback: it was wired. Why would I want to run an extra ethernet cable to my LED matrix, when we live in a world with WiFi?

OpenWRT on TP-Link WR703N + USB

Luckily I had a cheap TP-Link WR703N router available which would be up to the task. It was a bit slower than the raspberry pi though, so it probably wouldn't be able to do all the precalculation too. Fortunately the STM32F4 has lots and lots of clock-cycles left over to do that for us. Also the TP-Link WR703N has a USB port, and the STM32F4 discovery board has a nice USB port directly to the microcontroller. It's only Full-speed (USB1.1), but that should be enough right?

Nope, apparently it isn't. There is apparently way too much overhead on the USB port to get a halfway decent framerate, and it required a USB2.0 USB hub inbetween as the TP-Link's USB port was horrible at driving USB1.1 devices. Oops

Uart output on the TP-Link WR703N

There is another way to pipe out data on the TP-Link though: UART! It's hidden inside, so you'll have to manually solder to it, but it should be up to the task.

After some soldering I could get data out, but it was limited to 3megabaud. Not quite good enough for fluid framerates 🙁 The weird thing is that apparently the serial clock on this thing runs at 25mhz, so I had no clue why it would crap out above 3mbaud. After some searching I found that in the drivers it was artificially limited, probably because noone ever uses a serial port over 3megabaud, and it might not be able to reproduce baudrates above that accurately. After a quickly patching the driver and recompiling OpenWRT, I could get it to work on higher baudrates. Seeing as the serial clock was running at 25mhz, I figured 5megabaud would be easy for it to generate, and indeed the signal is rock-solid!

I wrote a quick little program that would get Art-Net data over UDP, and wired up GLEDiator to pump out some data. It worked like a charm!

Final hardware

The final hardware consists of:

- STM32F4 Discovery board, about €15

- TP-Link WR703N, about €20
- 9 HUB75 32×16 RGB 10mm pitch panels, about €18 each (incl shipping)
  (note: This is not the seller I bought them from. I was unable to find the seller that originally sold them to me)
- A cheap 5v 20A power supply, about €17
  (note: I did not double-check if this is enough current for the entire display to light at once!)

Compared to the dedicated hardware offered by adafruit for $300, I think my solution for less than a fifth of that (€45) is pretty good 🙂

Furthermore any computer or laptop with a network interface can drive this without needing extra PCI cards or DVI ports.

Final code

All code can be found on my Github repository, with instructions on how to compile and wire everything. Furthermore in the utils directory there is a file to generate a GLEDiator Art-net patch file, and a file to adjust the matrix parameters (brightness and gamma).

I've licensed all code with an MIT license, so feel free to reuse it in it's entirety or in parts for your own projects. Do note that I would highly appreciate it if you sent me a small e-mail about your project if you do, though!

January 4, 2015 Frans-Willem Hardijzer Electronics

7 thoughts on "96×48 full-color LED Matrix"

Pingback: RGB LED Matrices With The STM32 and DMA | Hackaday

David Langer
January 6, 2015 at 9:08 AM

Saw your post featured on Hackaday. This is awesome. I picked up a few 32×16 rgb panels a few months ago and have been tinkering around with them with a spartan 6 fpga. I've gotten full 24 bit color working on it but only for text and simple frame animations. Havent even attempted full video display yet but after seeing this post, i am going to to pick up some more panels and giving it a shot. Looks fantastic!

Markus Bauer

January 7, 2015 at 11:15 AM

Hey,
awesome project! How do you get the YouTube-Video to your TP-Link-Router? Or better, how do you convert your YouTube-Video to a Art-Net-Source?

Greetings

Frans-Willem Hardijzer

January 7, 2015 at 5:13 PM

I use a program called GLEDiator with its screen capture functionality. Not perfect, but it works 🙂

Pingback: RGB LED Matrix galore! | stijnvandrunen.nl

belouet

May 8, 2015 at 3:24 AM

HI,

Very good project with STM32 and TP link, but I don't find how connect it ? do you have picture for that

Sincerely
Eric

Ndrew Poon

May 13, 2015 at 12:13 AM

Hello there,

Saw your post on other website, I am wanted to produce a T-shirt that has a flexible LED matrix and control by Android.

Could I contact you by anyway?

Thanks!
Ndrew
