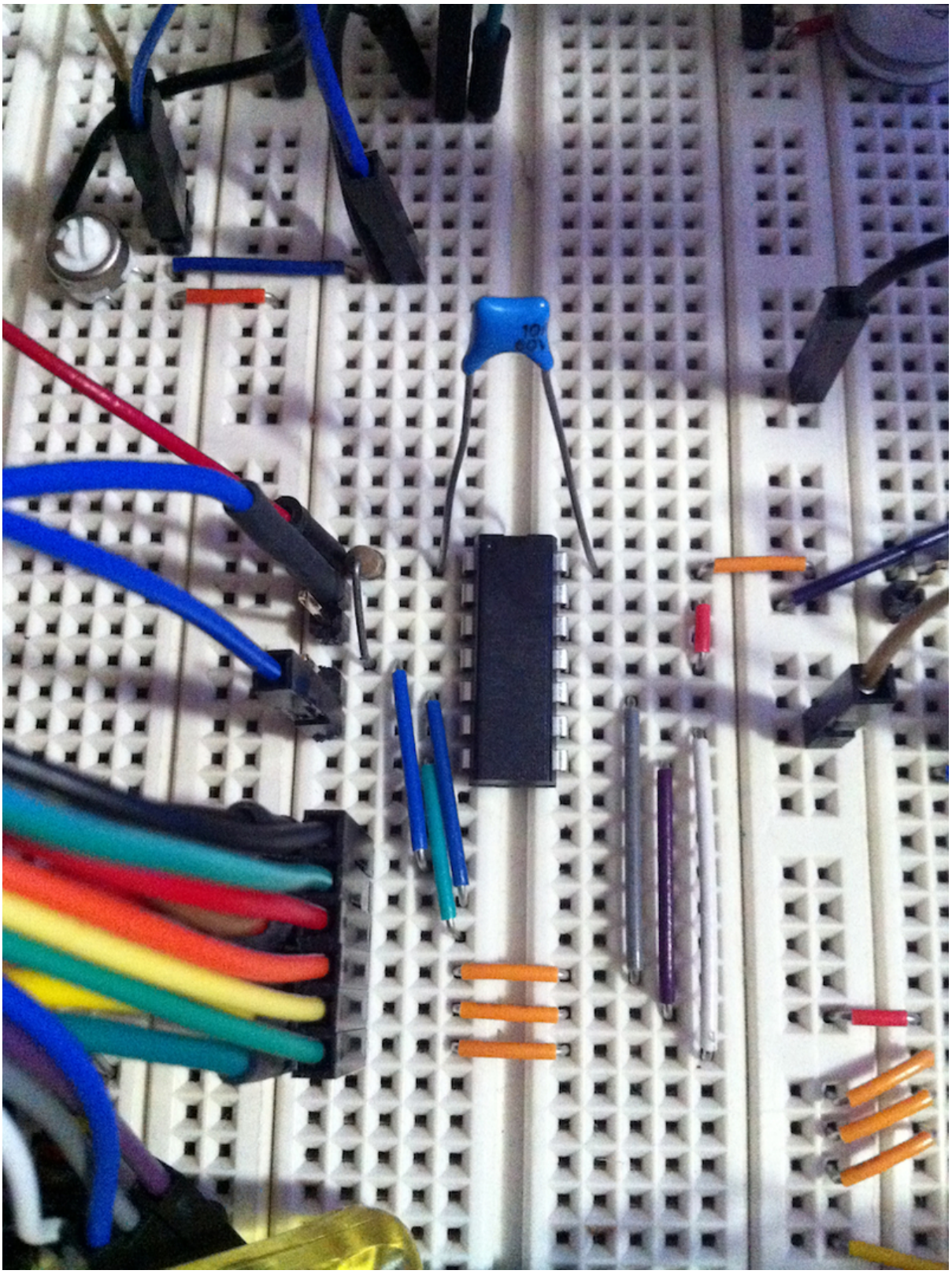
# The Pixif

Created by David Littell



https://learn.adafruit.com/the-pixif

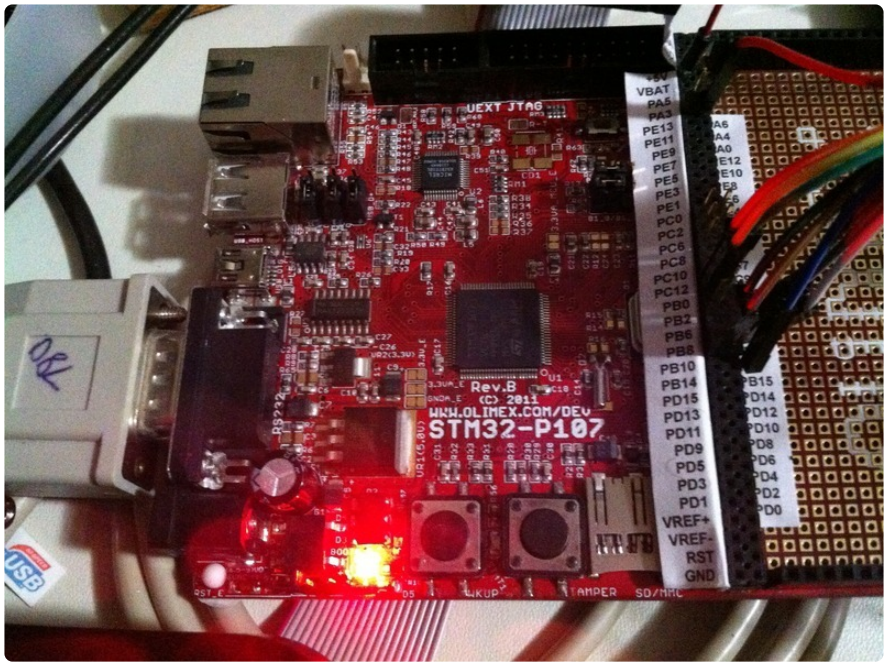
Last updated on 2024-06-03 01:38:10 PM EDT

# Table of Contents

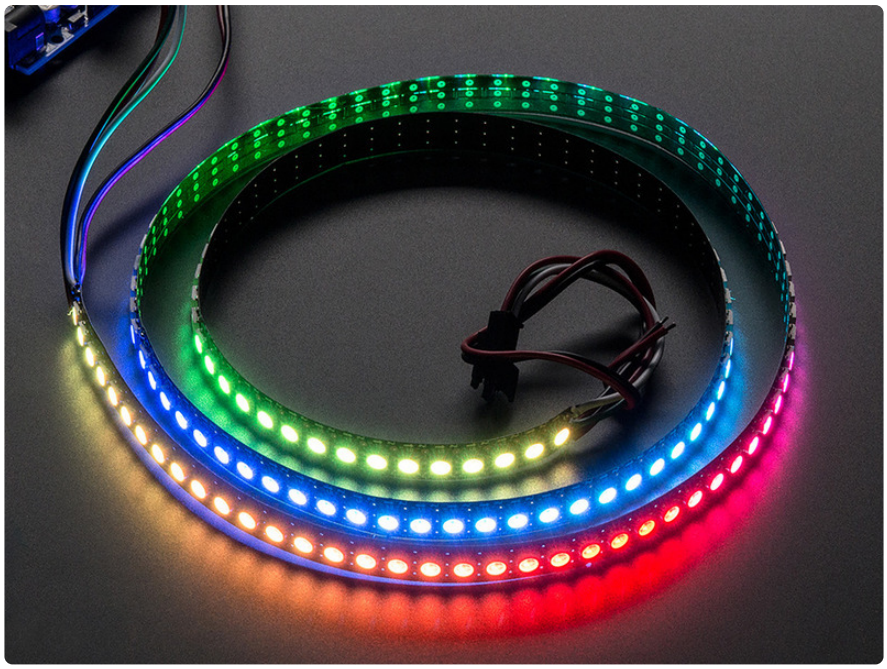
# From Here to There

I've learned quite a bit about NeoPixels through developing Madison's NeoClock (https://adafru.it/Ce6) and the PICsellator (https://adafru.it/Cex) but I wanted to take a step up in processing power. The PIC was certainly an incredible platform for those projects but I also wanted to work on something a bit more "forgiving" (from a floating-point calculation perspective). Finally, I wanted to have a simpler interface to the NeoPixels that didn't require as many bit-twiddling babysitting processor cycles. I've studied a number of different approaches (fancy SPI and DMA tricks with specially encoded memory to create the WS2812 bitstream, etc.) but my little brain craves simple. So the challenge became, as ever, to use what I've learned to create something I need. The gauntlet is thrown...

To help develop a wire-wrap MC68000 system I used an STM32-based board (https://adafru.it/efk) (another story for another time) and had studied it enough to feel pretty comfortable using it for my "next step up" in processing power. The STM32 family (https://adafru.it/efl) has some very nice features, I had already been down the tool-chain cobbling path, and the price was right (already owned!) so the question became how to get from here:




to here:



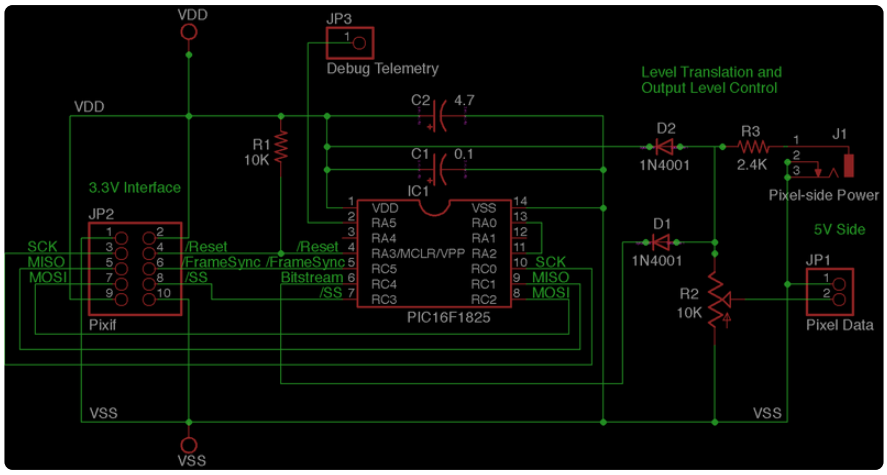
I need some sort of simple interface to NeoPixels...a pixel interface...a Pixif!  I had a name!  That's the hardest part, right?  ;-)

The PICsellator showed that I could efficiently drive NeoPixels and have a nice, clean interface in the PIC's memory (basically just a vector of RGB values for the pixels' colors).  How could I now have that simple interface appear in another processor's memory with minimal impact to that processor's (calculation and memory) bandwidth?  My first thoughts were to use the DMA-supported SPI Main in the STM32 to drive a modified PICsellator as a SPI Secondary.  That certainly seemed workable but possibly a bit limiting (no pun!) if I wanted to expand - the prospect of babysitting multiple SPI Mainss and their DMA engine(s) wasn't overly appealing.  How would it go if the pixel interface (the Pixif!) was the SPI Main, ran independently and retrieved its pixels' color data autonomously?  Sometimes it can be very enlightening to turn over a problem and work it backwards...  For me, this was one of those.

---

# Connect the Dots...

The Pixif's hardware is very simple, with almost all of the "special fancy stuff" being done by the PIC's software.  I wanted a straightforward interface to the STM32 board and was able to accomplish that by using a normal 4-wire SPI with the addition of two discrete signals.  The interface to the host comprises those 6 signals along with power for the PIC.  Note that even though the Pixif's PIC operating voltage range is 1.8V to 5.5V, the output level translation hardware constrains the lower voltage to be around 3.3V.  So, Pixif VDD greater than 3.3V: OK.  Lower, not so much.  See the Output Level Translation and Tuning page for more information on what's going on in that area.



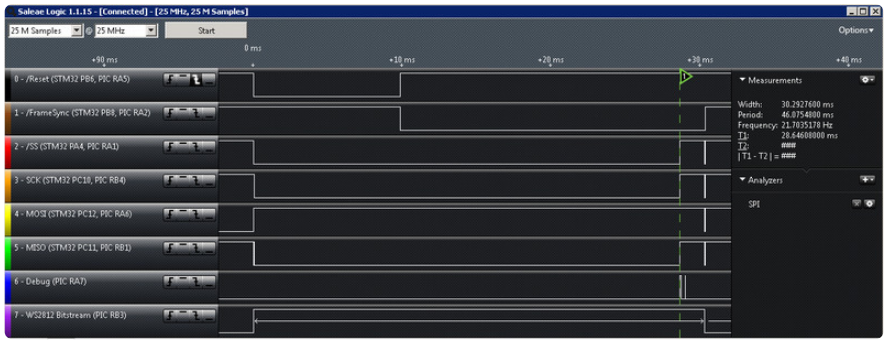

Of the two discrete input signals, /Reset gives the host processor ultimate control over the Pixif - for whatever reason, start over at will! ;-)

The second input, /FrameSync, allows the host processor to control the rate at which the Pixif retrieves a set of RGB bytes for the NeoPixels. A Pixif "frame" is the entire set of color byte values for all of the NeoPixels the Pixif is configured to drive. For example, 32 NeoPixels results in a 96-byte frame, 144 NeoPixels yields a 432-byte frame, etc. As shown later in the logic analyzer traces, whenever the Pixif (as the SPI Main) asserts the SPI /SS signal it will merrily clock in whatever's present at the MISO signal until it's received the complete RGB byte frame. (This highlights the importance of DMA support for the SPI Secondary on the host processor so it can spend its time elsewhere.) The Pixif continuously does its SPI operations with interleaved NeoPixel bitstream generation, so the normal rate of /SS assertion is a function of the configured number of pixels. But because the Pixif checks the /FrameSync level before asserting /SS the /FrameSync signal can be used to throttle how often the Pixif retrieves the RGB frame. For example, the logic analyzer traces show operation with a 144-pixel strand and /FrameSync constantly asserted following the Pixif configuration. Instead of being constantly asserted, if /FrameSync is pulsed (low) at 60 Hz the frame rate would be reduced from ~188 Hz to 60 Hz with the SPI /SS being asserted immediately following the /FrameSync pulse. Of course the /FrameSync assertion rate must be lower than what the free-run (/FrameSync constantly asserted) /SS rate would be. But what is that rate? And where did this weird 188 Hz rate come from? Glad you asked...

The bitstream sent to the NeoPixels by the Pixif runs at ~650 kHz. Because the Pixif interleaves the retrieval of the RGB bytes via SPI with generating the NeoPixel bitstream (study those logic analyzer traces!) the overall rate of data transfer is determined by the bitstream rate (~650 kHz), not the SPI clock rate. So, 144 pixels at 24 bits/pixel at 650 kHz yields ~188 Hz. (Note that the SPI clock rate need only be fast enough to keep the bitstream "pipeline" full of the frame's RGB bytes.)

# I Love LA!

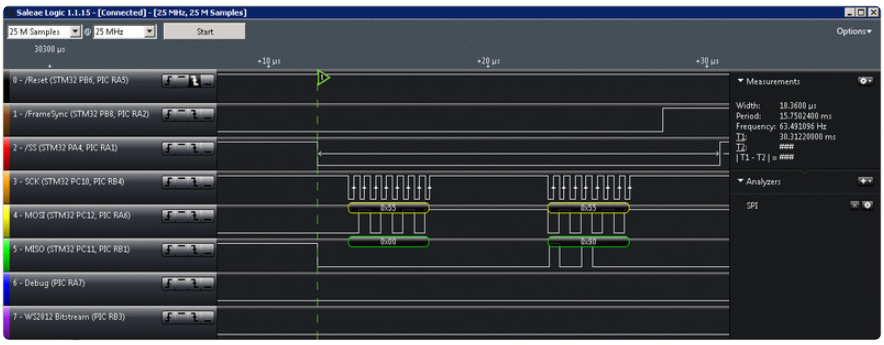
...a Logic Analyzer, that is.  The Pixif interacts with its host processor via SPI.  Because the number of NeoPixels it can drive is configurable (up to a maximum determined by its 16-bit counters) it needs to know that number before it starts fetching RGB color byte data.  The host provides the number of pixels to the Pixif during a Reset and Configuration Sequence:




First, the host fully resets the Pixif using its /Reset input.  The logic analyzer trace above shows /Reset being asserted for 10 mSec - that's just an arbitrary duration I coded on the host.  It could be much, much shorter and still work because the /Reset signal directly resets the PIC.

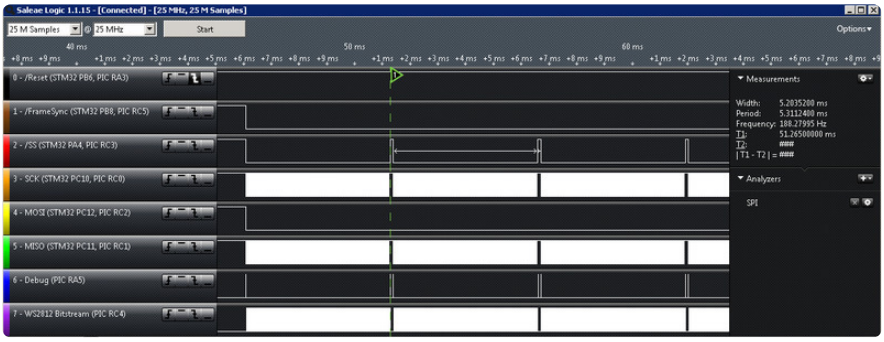
Following /Reset being negated (at 10 mSec), the Pixif waits for the /FrameSync input to be asserted by the host indicating the Pixif should perform a 2-byte SPI exchange.  It's this exchange that determines how many pixels the Pixif will drive.  As shown in the trace following the /Reset negation it takes the PIC a little time to get happy (about 18 mSec), recognize /FrameSync, and initiate the first SPI exchange.  The green flag in the trace indicates when things begin to get serious: SPI /SS is negated, the /FrameSync assertion is noted, and the SPI pixel count exchanges are done (at about the 30 mS point in the trace).

The first SPI exchange (the "Configuration" portion of "Reset and Configuration") simultaneously informs the host of the maximum number of pixels the Pixif can drive and the Pixif of the number of pixels it should actually drive.  If, for some reason, the host doesn't like the result (maybe it expected to be able to drive more pixels than the Pixif supports and sent down a number larger than the Pixif's maximum) it can simply assert /Reset and restart the process.  Here's the detailed trace of the Configuration portion with the number of pixels to be driven set to 144:



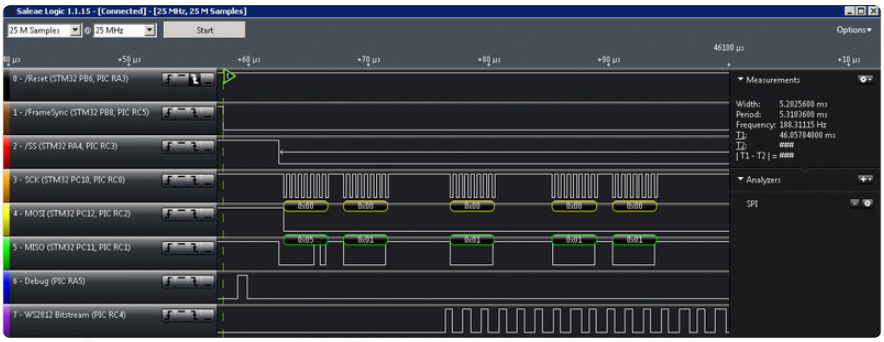

One important note in this trace is the negation of /FrameSync at about 28 mSec. Following the exchange of pixel counts the Pixif waits for /FrameSync to be negated by the host. The Pixif then waits for a subsequent /FrameSync assertion to begin its retrieval of the pixels' RGB color bytes. This sequence allows the host to do any needed final preparation before allowing the Pixif to begin fetching pixel color data.

Once /FrameSync is asserted by the host the Pixif will assert SPI /SS and begin retrieving the RGB color bytes for the pixels. Here's a summary view:




This trace shows the first 3 full frames (and a portion of the 4th). Because the host leaves /FrameSync asserted the Pixif continues to retrieve all of the RGB color bytes its connected pixels. The green flag on this trace indicates the negation of /SS and the beginning of the "quiet time" reset interval needed by the NeoPixels. Note that the WS2812 Bitstream is being generated at the same time as the SPI exchanges are being done. Pulling this off on an 8-bit PIC was certainly one of the more challenging aspects of this project!

Here's the detail of the the first few SPI exchanges following the /SS assertion:

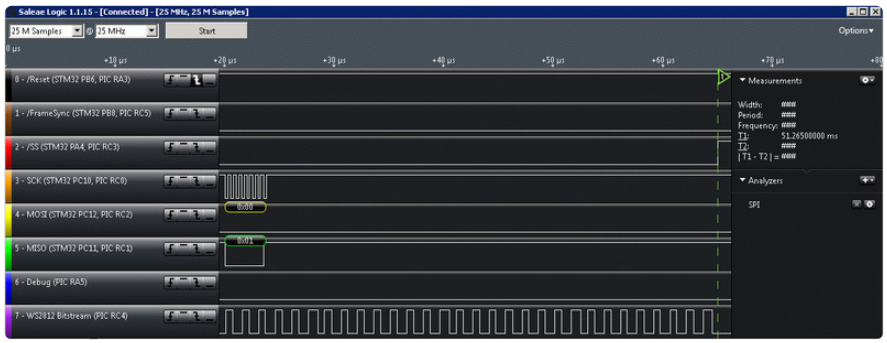


Remember simple? The other side of that coin is sometimes you have to do not-so-simple things to be able to do simple things. The above trace shows the SPI exchanges for the first 5 color bytes and the resulting WS2812 Bitstream created from the first 2 color bytes. But, why the delay in starting the bitstream if the received SPI color bytes are just handed off to the bitstream generation mechanism? Remember from the Data Sheet that the WS2812B color byte order is Green-Red-Blue? Ugh. Well, the entire created universe has always thought of color values as Red-Green-Blue and I wasn't about to burden my host-side software with any byte re-ordering silliness. So I buried that silliness in the Pixif. Here's the magic: the color byte order in the host's memory is RGB, so the transmitted order via SPI is RGB. That's cool. But the WS2812 bitstream order, being GRB, means the Pixif has to hold off generating Red's bitstream until after Green's. So, for each received RG portion of an RGB triplet the Pixif can't start the triplet's bitstream until after the Green byte is received via SPI. You can see this in the trace as the first SPI byte is a Red value (0x05) and the second is Green (0x01). If you decode the 16 bits of the WS2812 Bitstream trace you see what is sent to the pixels: Green followed by Red. (Blue's bitstream encoding follows immediately after Red's beyond 46100 uSec. It really is there - trust me! ;-)

It all hangs together and results in a coherent bitstream for the pixels because I carefully chose the SPI clock rate to blend with the bitstream clock rate and used the PIC's USART to create a "bit pipeline" to buy enough time for the PIC to handle the SPI exchange concurrently with feeding the bitstream generation mechanism. I wanted a SPI clock rate slow enough to not have electrical problems when run across a few inches of wire (does this antenna make my clock look bad?) and allow the host-side DMA to easily blend with all the other host-side memory bandwidth consumers. But that had to be balanced with the fact that a SPI clock rate that's too slow would result in gaps in the generated bitstream (a "pipeline stall" or "bubble") and all sorts of whackiness from the pixels! Yikes!

Whew!

To finish up our deep-dive trip to LA here's what happens at the tail end of an /SS assertion:



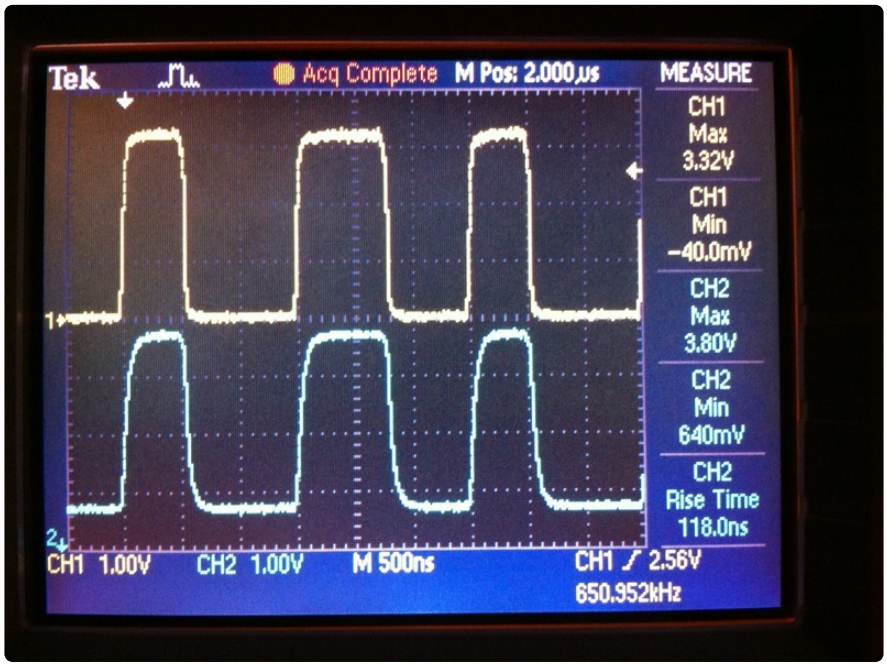
The above trace shows the SPI exchange for the very last color byte (Blue) of the very last pixel along with the ongoing bitstream being generated from previous color bytes. Notice the Pixif stops clocking SPI after the last pixel's last byte but continues to "drain the pipeline" of the bitstream mechanism. The last pixel's Blue value appears in the bitstream, /SS is negated (@ the green flag), and the WS2812B reset interval begins. Following the reset interval the fun starts all over again!

---

# Output Level Translation and Tuning

The STM32 host board provides 3.3V to the Pixif, so the PIC's WS2812B bitstream output voltage range is 0 to 3.3V. Since the NeoPixels are powered at 5V in this project the bitstream voltage from the PIC should be translated to a level more palatable to the WS2812B's. There are a number of ways to accomplish this but for the Pixif I implemented an approach shown in Microchip's 3V Tips 'n Tricks (https://adafru.it/Cez). This document has some neat ideas for handling a number of 3V-to-5V situations. For this project I used the incredibly simple (and cheap!) diode-based circuitry described in Tip #7. (Note that Tip #7's written description has a typo that reverses the references to D1 and D2 relative to the schematic shown in the document's Figure 7-1. The lesson for me (once again!) was to make sure I take the time to understand what a circuit actually does before just plopping it in a design and hoping for the best.)

There are probably better theoretical choices for the diodes (and the pullup resistor value) but in the end I was very happy with the result shown here:




The top trace (CH1) is the bitstream output from the PIC measured at D1's cathode (10x probe). It's what you might expect: precise levels, clean, fast edges, no knees, no

ringing, and minimal undershoot.  One would always hope for such a nice, clean signal at this point because it's about to be mashed some...

The bottom trace (CH2) is the final level-shifted (and output-level-adjusted via R2) signal to the NeoPixels measured at JP1 pin 2 (10x probe).  The logic-low voltage level is exactly as Tip #7 predicted: D1's forward voltage.  The logic-high voltage level (3.8V) is also as predicted by Tip #7 after R2 reduces the level slightly.  (This particular signal level allowed correct operation of the NeoPixels with no risk of the "whiteout" that occurs when the DIN voltage is too high relative to the pixel's VDD.)  I was very happy with level-shifted edges and rates: even in the presence of the probe capacitance the 118 nS rise time gives an ~8.5 MHz edge rate.  Not at all bad for a ~$0.20 level shifter!

---

# Showtime!

For testing, the Pixif was configured to drive a 144-pixel strand with no update frame synchronization.  As shown earlier, this yields an update rate of ~188 Hz.  Because the STM32 has independent DMA support for its SPI Slave there's no impact on the STM32's core processor other than a trivial ~82 kBytes/Second of memory bandwidth (3 bytes per pixel * 144 pixels per update * 188 updates per second).  This leaves all those core cycles for calculating pretty stuff...and slogging through software floating-point calculations!  :-)

To have something interesting to display on the NeoPixels I ported to the STM32 board one of the algorithms from the PICsellator (https://adafru.it/Cex).  Algorithm 5, the "Sine Springboard", is shown here running with a 1 second sleep interval:

(That's a PICsellator in the background running inside two crackled-glass votives - my version of a Lava Lamp!  ;-)  So I'd consider this in-depth exercise a total success - a simple, efficient, reliable interface to independently handle all the NeoPixel twiddling while my core processor concerns itself only with what to scribble in a simple vector of RGB values in memory.  The key to the whole thing was good DMA Support for the SPI - there's just no substitute for good hardware!  (Except, of course, pain in software... ;-)

I'm thinking now of the next projects: Maybe a board design to incorporate a few Pixifs (with only the 5V power for the level translation being common among them).  That would make it easier to replace the software-floating-point STM32 with a faster, hardware-FPU processor (probably a DSP) having multiple DMA-supported SPI Slaves and then take the next leap in software to drive it all.  The key to success in this will be to study the Data Sheets to ensure that the Chosen One has good, no-babysitting-required DMA support for multiple SPI Slaves.  Any ideas?

One of my favorite hangouts is the Glowy things forum (https://adafru.it/efp) - comments, questions, and suggestions welcomed!