AOScope program

V1.4.4 last update 2/1/21 by Mark Bauer

This program was created so that students could demonstrate various concepts for electrical engineering labs when they don't have access to what you would expect to find in the typical lab.

Limitations:

To create this project, I needed to use a processor that was already widely available at a very low cost. Having software that would work on various platforms without needing involve a huge amount of effort. The Arduino platform has existed for a long period of time and already is supported on MacOS and Windows operating systems. This platform also supports the serial plotter function that permits graphical information to be displayed without me writing code that will be obsolete every time the operating system gets updated. I started off with the Arduino Uno compatible board because there are many clones of this board available at very low costs. I will be using the Arduino Nano for demonstrations here as it is available for less than $5 and it also will plug directly into the solderless breadboards we use. The program will work on any ATMEGA328 based Arduino running at 5V and 16Mhz.

The 328 processor is significantly limited, but we manage to get as much out of it as possible. The biggest limitation is the lack of RAM space at 2048 bytes. The next limitation is the speed of the A/D converter. To hit my goal of 10K samples per second on two channels, I needed to reduce the number of bits to only 8 bits. The source code is given for those that wish to make changes to it and there is a section at the end that talks about how the program works.

I started this project by looking online for what had already been written. Although one of the programs claimed to hit 10K samples per second, it wasn't anywhere near that. But it did give me ideas. I started off wanting to hit at least 10K samples per second with a minimum of 2 channels. Once I got that running, I kept adding features until I ran out of resources, one of which is time.

List of features so far:

- 2 8 bit A/D converters running at 10K samples per second
- 4 digital channels
- 512 sample buffer for each of the 6 channels
- Adjustable pre-trigger so you can see what happened before the trigger
- Selectable trigger source with selectable level, rise, or fall.
- Auto or manual trigger hold off.
- Second trigger for time measurements with selections for levels and channels
- 3 Clock outputs with various restrictions
- Scroll mode for continuous display, adjustable speed
- Measure between two points, with limits
- FFT, but don't expect too much

Quick command list:

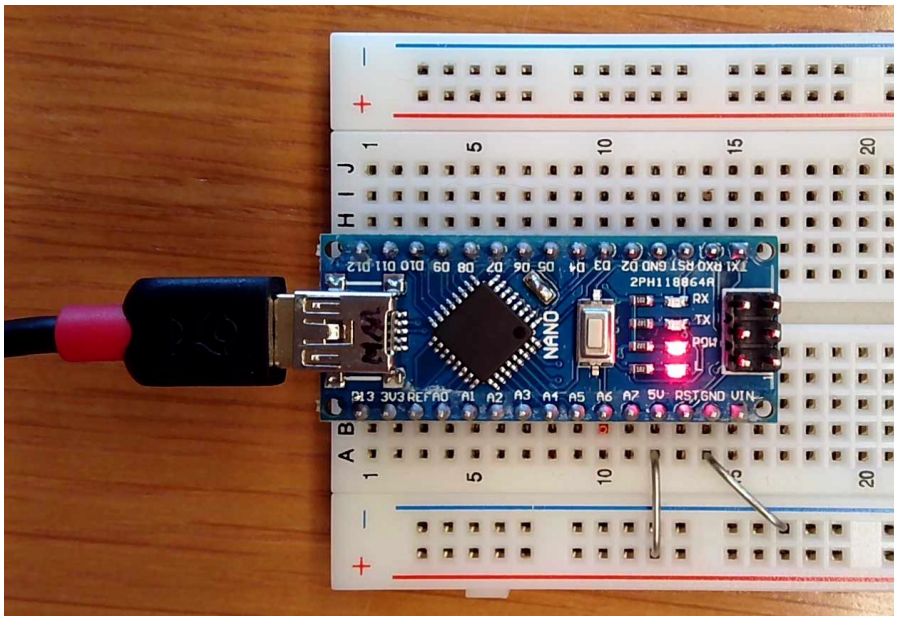
The return key by itself should cause a single sample

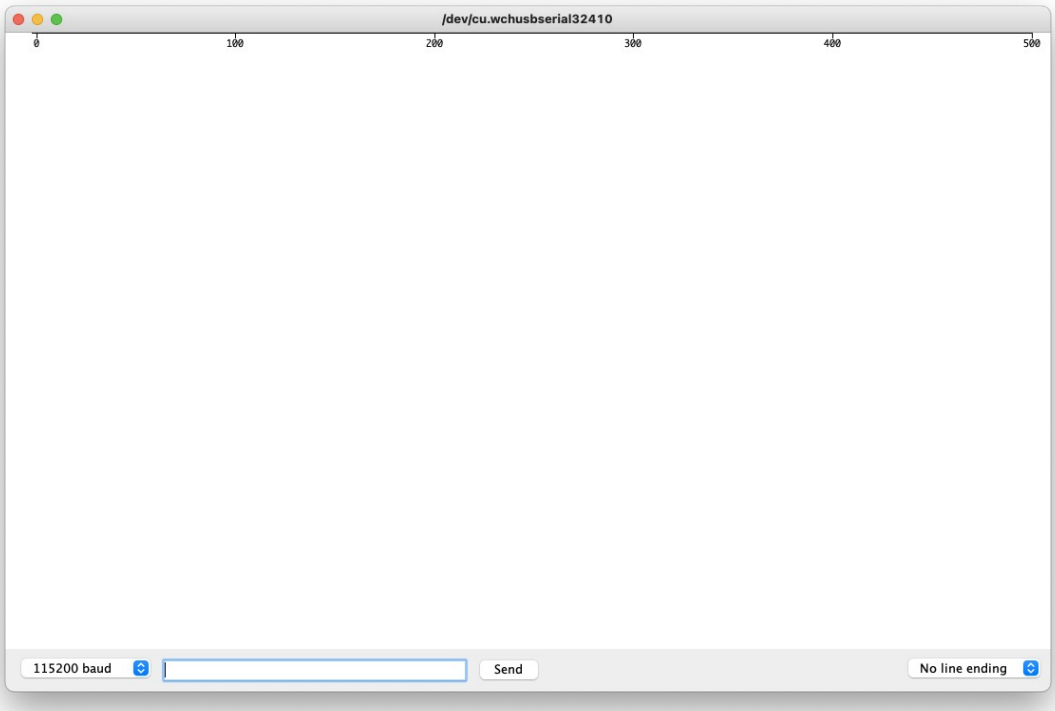
CA	Clock A output is on D10
CAF n	Set the clock frequency to n 1 -> 4M square wave output
CAD n	Set clock divider to n
	n=0 off
	n=1 125nS per tick
	n=2 1uS per tick
	n=3 8uS per tick
	n=4 32uS per tick
	n=5 128uS per tick
CAP n	Set clock period in number of ticks
CAH n	Set number of ticks output is high per period
CB	Clock B output is on D5
CBF n	Set the clock frequency to n 35 -> 4000000 square wave output
CBD n	Set clock divider to n, same as Clock A
CC	Clock C output is on D12 fixed divider at 100uS
CCF n	Set the clock frequency, limited range 1 -> 1K
CCP n	Set the clock period in 100uS ticks 2→65535
CCH n	Set number of ticks high in each cycle
Cs n	If you have a MAX7426 filter connected, it will give you a sine wave output at the requested frequency. Clock A and B are both used to generate this.
LH	Clock C output (D12) is set to high level
LL	Clock C output is set to low level
LPn	Clock C is given one pulse. Pulse duration is in 100uS ticks
L	One pulse with same duration as before.
P n	Pre trigger to 400 samples. Range 0 → 65535
TL v	Trigger level to a voltage mV. Range 0→Vcc (5000)
TF v	Trigger on falling edge (optional set trigger level)
TR v	Trigger on rising edge (optional set trigger level)
TX	Trigger to free run
TH n	Trigger hold off time to n seconds 0 is off. Range 1→ 60
TS n	Trigger source 0→ 5
	n=0 Analog port 0 A0
	n=1 Analog port 1 A1
	n=2 digital port 2 D2
	n=3 digital port 3 D3
	n=4 digital port 4 D4
	n=5 digital port 5 D6
	n=6 Clock A output D10
	n=7 Clock B output D5
	n=8 Clock C output D12
ML v	Measure level to a voltage in mV 0->5000
MF v	Measure on falling edge with optional voltage
MR v	Measure on rising edge with optional voltage
MX	Measure off
MS n	Measure source 0 → 8 same as trigger source
SC n	Drop the sample rate from 10K samples per second to 10K / n per second S2 would give us 5K S1000 would give us 10 samples per second
SN	Change to no scroll
SS	Change to smooth scroll mode
Vn	Set calibration of voltages, will keep it over reset.
F n	FFT mode: 0 is off, 1 is magnitude only, 2 is magnitude and phase
?	Show version information

Getting started with an example

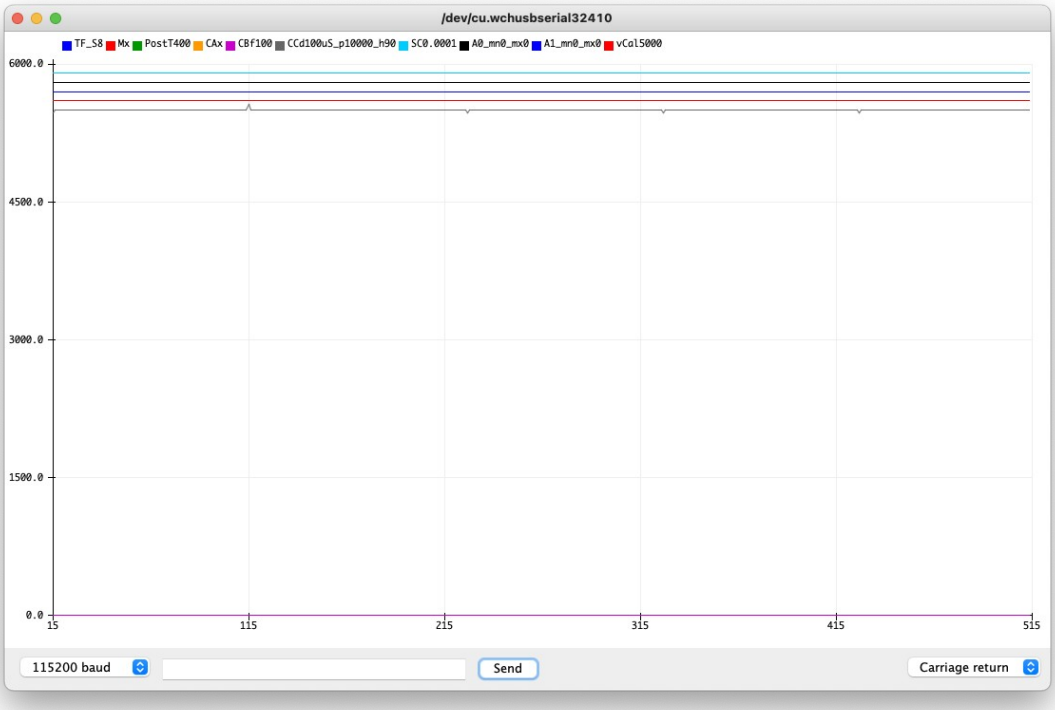
The program is written for the ATMEGA328 based Arduinos. I have tested it with many different ones and all of them seem to work just fine. If you are looking for an inexpensive one, I suggest the Arduino Nano.

Plug in your Arduino Uno (or compatible) into your computer. Download the Arduino development environment from https://www.arduino.cc and bring up that program. There are also resources there if you have problems getting that program up and running. Download the AOscopeV1_4.ino program from markus.unl.edu. Open the AOScope program with the Arduino environment. You should see the source code at this point. Upload the program to your Uno. The Uno should be blinking fast indicating that it is running. I am using an Uno Nano. The pin numbers listed on the top of the board are the ones we will be referencing. If you are using a different Arduino, the order may be different but the labels are what is important. The Nano plugs directly into the bread board making working a bit easier. I also suggest just leaving it plugged in as bending pins is easy to do when removing it.



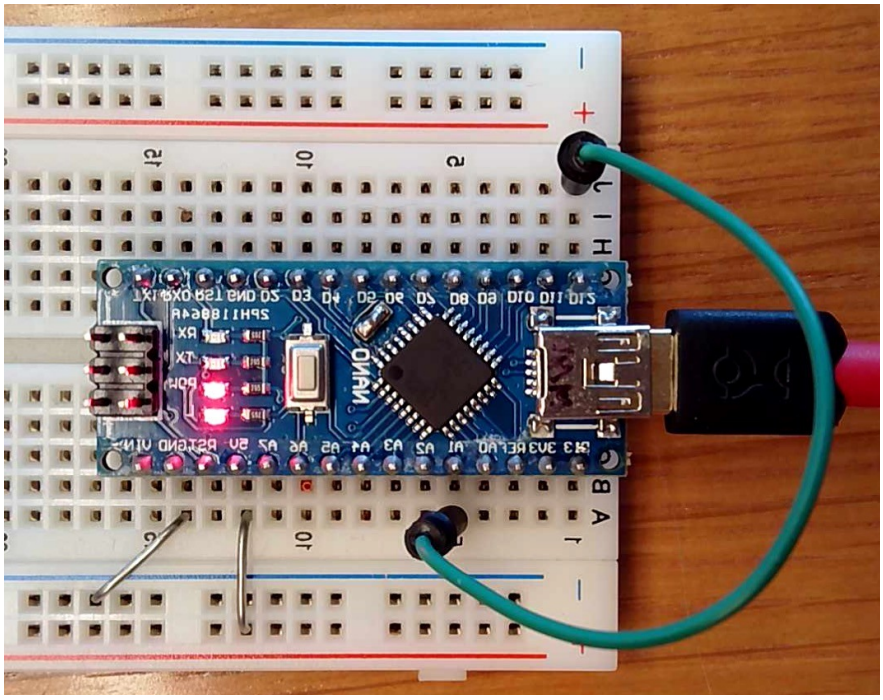


In the Arduino environment under Tools clock on the Serial Plotter option. It will open up another window. Notice that in the lower right corner it says “No line ending”. Click on that and change it to “Carriage return”. Change the baud rate to 115200 if it not already set to that. Now just click on the “Send” button.

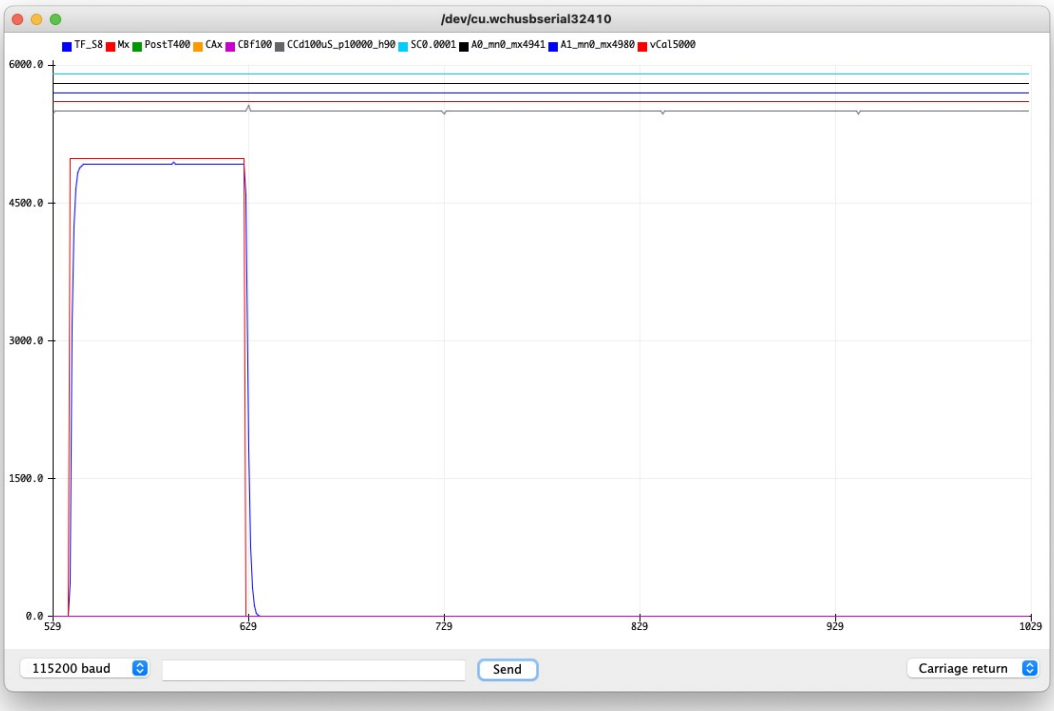


Nothing is connected to any of the input pins so the plot is not very exciting. Notice near the top of the plot there is a bunch of cryptic information. This tells us the settings. By default two of the clock outputs are running. Clock A is off as noted by “CAoff”. Clock B is in frequency mode of 100 Hz as noted by “CBf_100”. Clock C is in pulse mode with “CCd100uS_p10000_h90”. The “d100uS” part tells us that each tick of the clock is 100uS in length. The “p10000” indicates that the period is 10,000 ticks at 100uS or 1 second. The “h90” indicates the pulse is 90 ticks at 100uS or 9mS. The number along the X axis really don’t tell us anything important, just remember that there are 500 points from the left edge of the plot to the right edge.

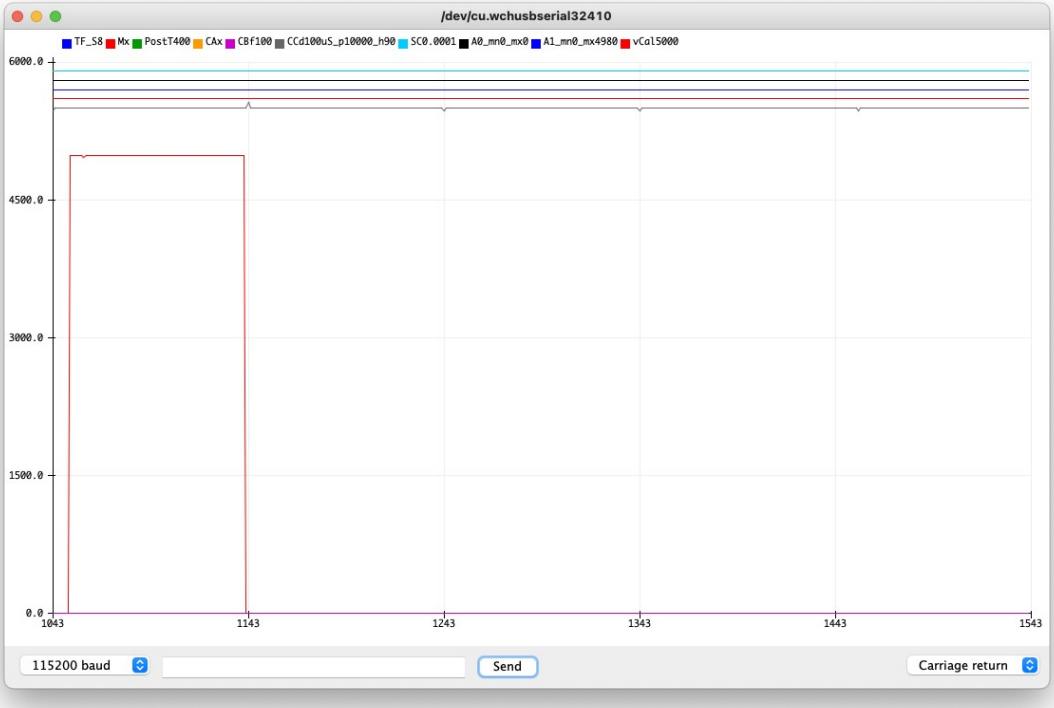
Add a jumper from the output of clock C (D12) to the analog port 1 (A1).



Click on the Send button, it should redraw the plot.



We now have a plot. There is a square pulse now. But there is also a not quite so square pulse that lags the original. This is caused by the A0 port not being connected to anything and the voltage on it just drifts around. Add a jumper from A0 to ground. Redraw the plot by clicking Send again.



We should have a single square pulse. You can see faint vertical lines that divide the plot into 5 sections. Each section is 100 points or 10mS. The full width of the plot is 50mS. By default, clock C is pulsing with a repetition rate of 1 per second with a pulse height of 9mS. Note that the square wave is just short of 10mS, likely 9mS. Notice that as we hit the Send button and it redraws the plot, the pulse always ends up in the same spot. That is the function of the trigger. Before explaining how that works, we need to talk about about how digital scopes work in general.

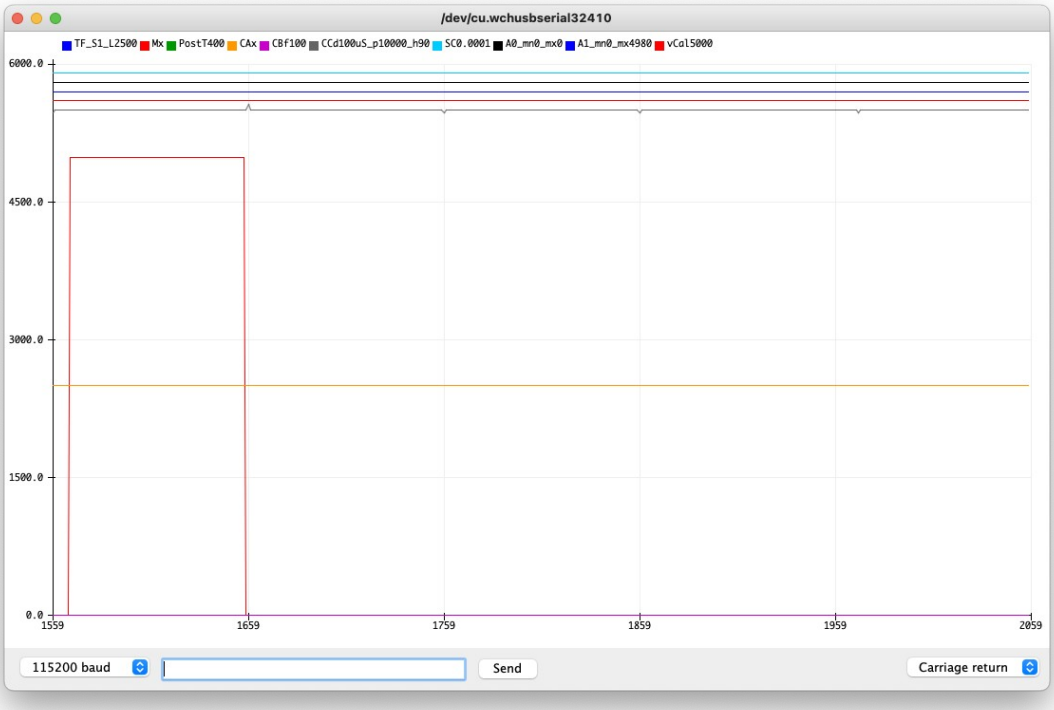
The Scope

I am talking about a digital oscilloscope at this point, you may never work with the older analog ones at this point. A scope gives us a picture of how voltages change over time. If a voltage changes really slow, we could get the information with just a volt meter. If a voltage changes too fast, we would never be able to see it. Scopes measure the voltage many times per second. A good modern scope may measure more than a billion times every second. Those voltage measurements are translated into points on the screen where the Y axis is the voltage and the X axis is time. On a real scope, you can adjust scale factors for both of those things. Microprocessors have A/D converters that can take a voltage and convert it to a number. The speed at which they work has a limit. The processor we are using, the ATMEGA328 was selected because they are cheap and you can get them just about anywhere. It has significant speed limitations so with some fancy programming, we are able to get it to run 20,000 samples per second. Wanting to watch 2 channels, we only get half that speed or 10,000 samples per second. At that speed, anything over 2-3 Khz will be hard to see. The plot output is only 500 points wide, so that represents 50mS. When the program is sampling, you can see the LED blink at a rather high rate of speed. The data is being put into a ring buffer that only keeps the last 512 values.

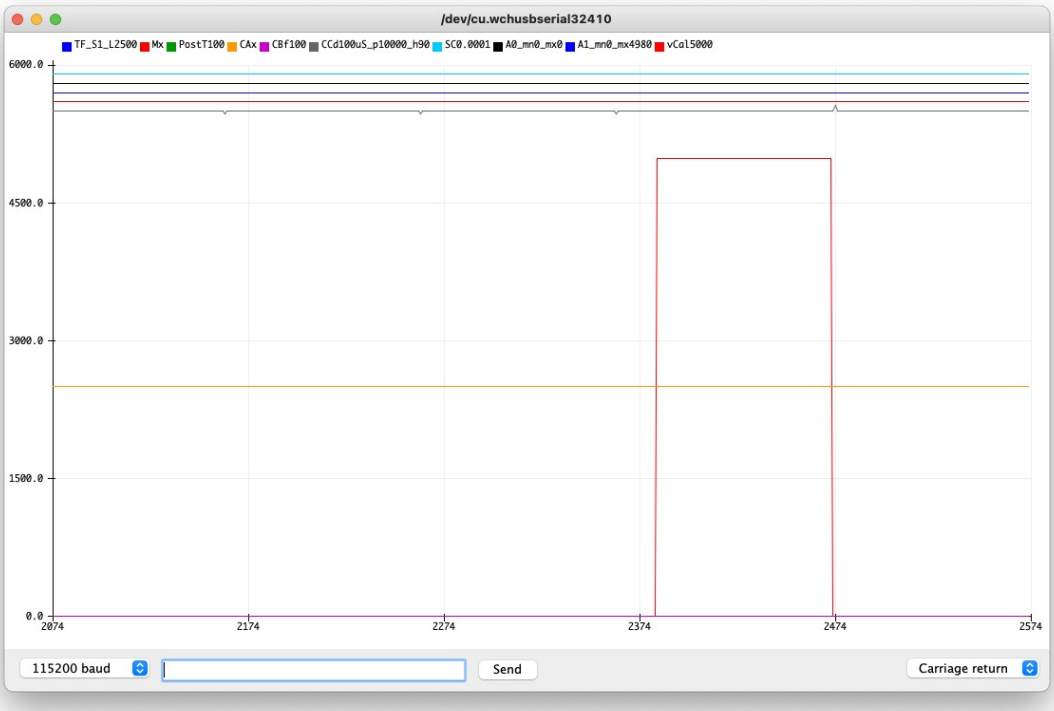
The Trigger

Every oscilloscope will have some sort of trigger function. When events occur either randomly or infrequently, we need to be able to catch the event. This AOScope program has a significantly adjustable trigger system. To understand how it works, you need to think about in terms of the program is sampling at 10,000 times per second. We need something telling the program when to start and stop. In our example we have clock C putting out a pulse once per second and the pulse only lasts 9mS. The width of the plot on the screen is only 50mS so the odds of just randomly catching it is rather low. We can resolve it in a couple of ways. We can just wait until something happens then start sampling. That something is the trigger event. This program as with most scopes allow adjustment as to what is an event. We can select which signal we wish to watch and also what the signal needs to do to trigger an event. We can see what the program is set to by looking at that cryptic status line. The default setting is T_F_S8, this the signal Falling on source number 8 (clock C output). Let’s change it to watch the A1 line we are connected to. For a command type “TS 1” and hit return. This tells the program to watch

the A1 line for the trigger. Notice on the status line it now says T_F_S1_L2500. Hit an additional return to force it to redraw the screen.



You will also notice a new horizontal line at 2500mV. We are telling it to trigger when the voltage on the A1 line falls from above 2.5V to below 2.5V. Also notice that the A1 signal drops from a high signal to a low signal 100 counts from the left edge of the screen (400 from the right edge). The way the program works is that it is almost always sampling the inputs. We need to know those voltages anyway to check for a trigger. When an event occurs, if we were to stop sampling at that point, we would only see what happened before the trigger event. If we started to sample at the trigger event and took the next 500 points, we would only see what happened after the trigger. This is adjustable with the P command. In the status line you see a PostT400. That tells you the program is setup to keep recording 400 points after the trigger event. Enter the P100 and press the return a second time to get it to redraw the display.

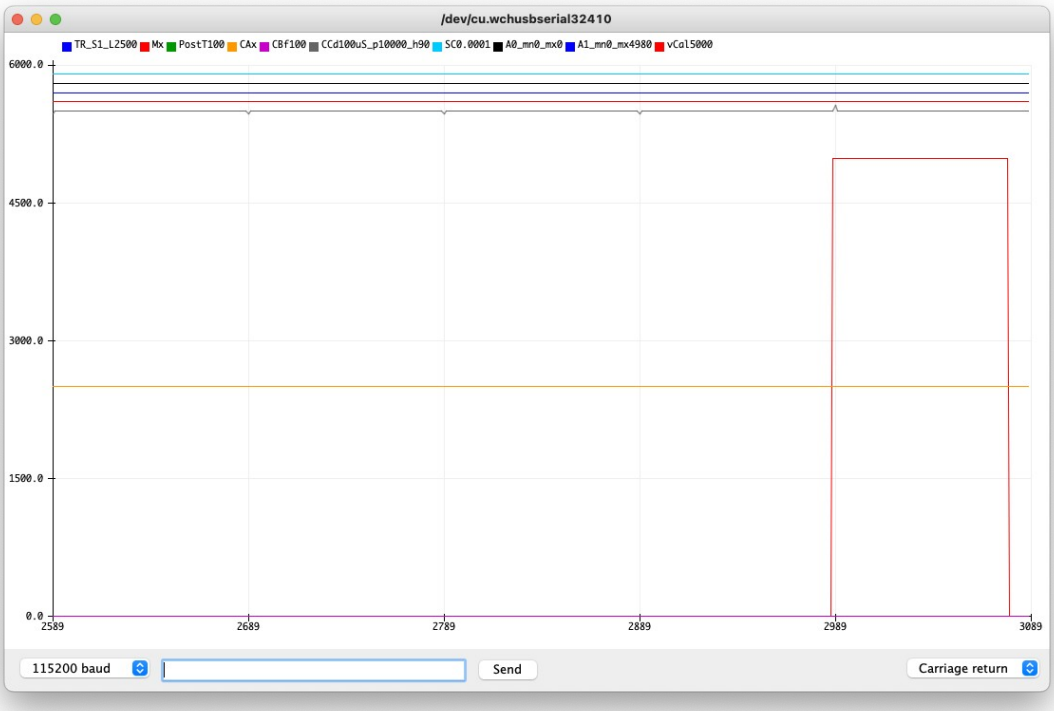


Notice the pulse is now located 100 points from the right edge of the plot. This allows us to choose to see what happened before and what happened after a trigger event.

If I wanted to trigger on the rising edge of the pulse, I could use the TR command to flip it to that. You can set the level at the same time as you set TR or TF by putting a voltage level after the command. You can turn off the trigger with TX. It will just trigger instantly.

There are 9 different sources to the trigger.

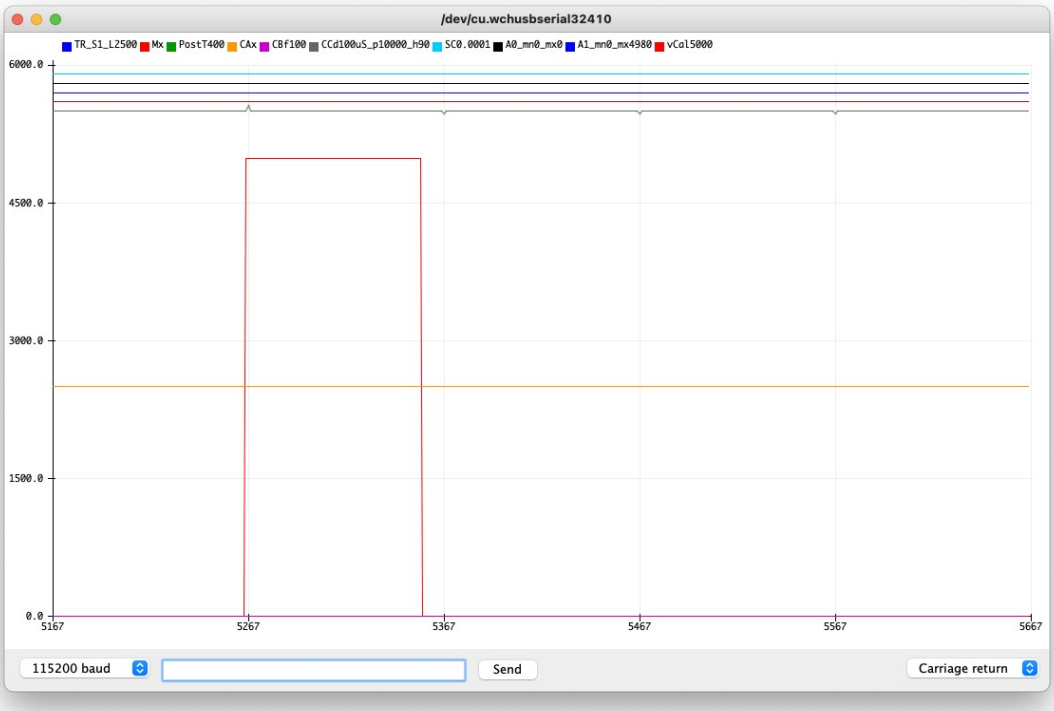
0	Analog port 0	A0
1	Analog port 1	A1
2	digital port 2	D2
3	digital port 3	D3
4	digital port 4	D4
5	digital port 5	D6
6	Clock A output	D10
7	Clock B output	D5
8	Clock C output	D12



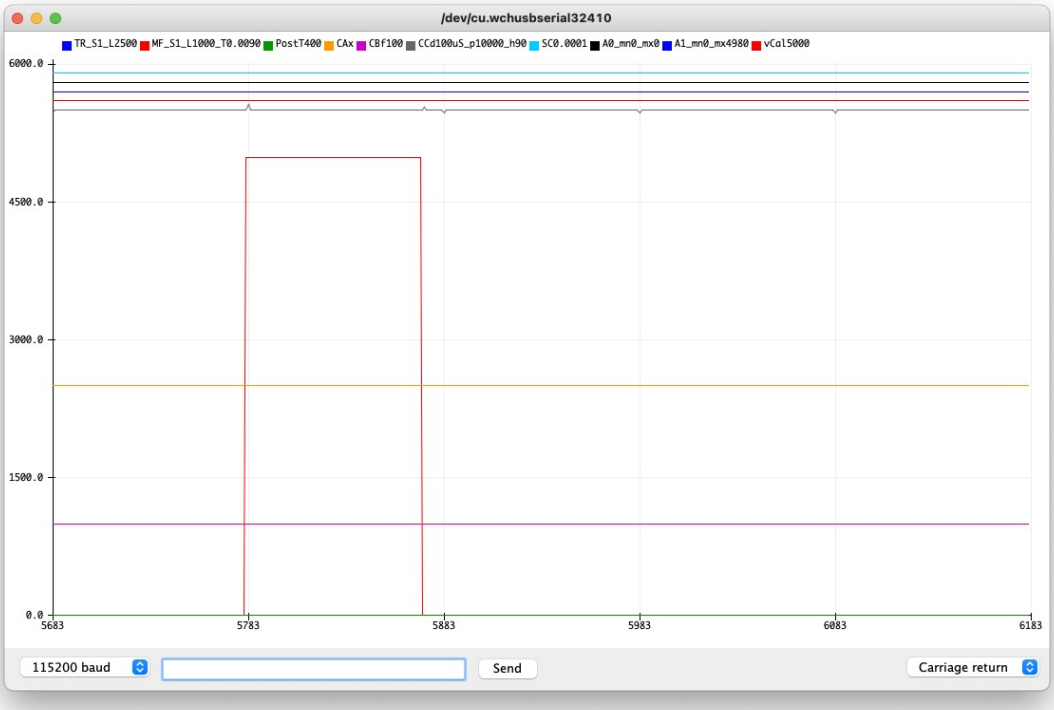
Notice there are 5 horizontal lines at the top of the plot. The top 4 are the digital inputs. The 5th one tells us where the trigger is. Notice there is a small upward spike where the trigger occurred.

Measure Point

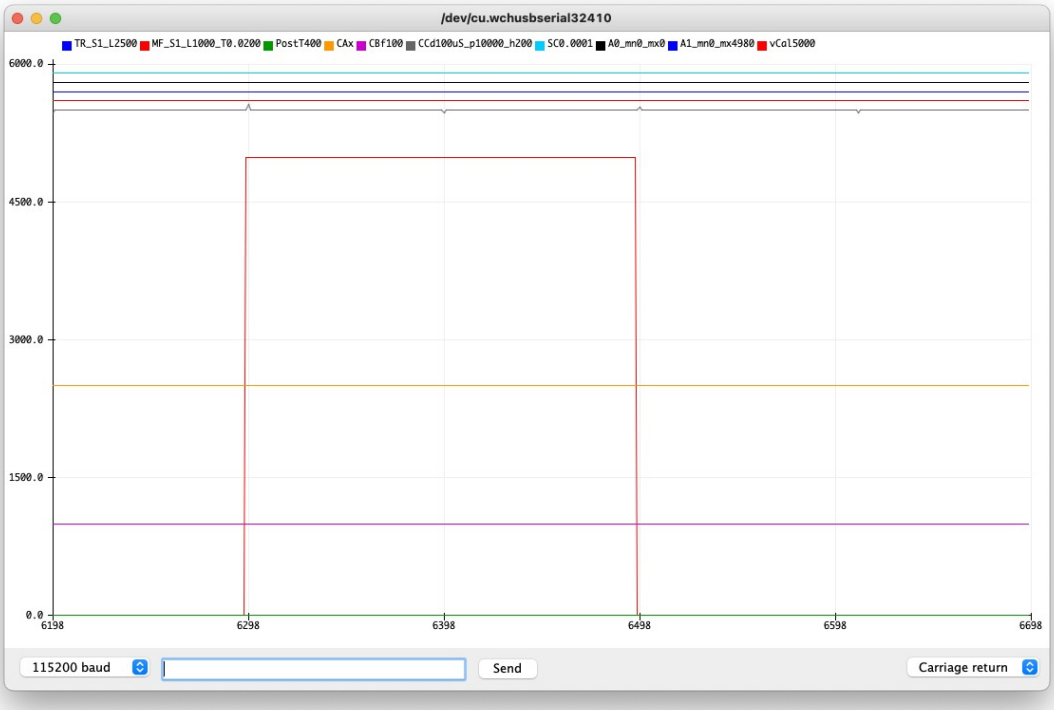
One of the frequent uses of a scope is to measure the time between two different events. This program can do that also. The measure point works just like the trigger event. In the previous example, we already know the width of the pulse we are generating is 9mS, so we will start by measuring that. We should still have a wire between the clock C output on D12 and the channel 1 input on A1. I suggest resetting the processor by the reset button, or just unplug and plug it back in. By default clock C should be one pulse every second that is 9mS wide. I want to time between the rising edge of the pulse and the falling edge of the pulse. We are feeding the pulse into A1 line so we need to switch the trigger to channel 1. Enter the TS1 command. Note that for input “TS1”, “TS 1”,and “ts 1” all work equally well. We also want it to trigger on the rising edge not the falling so enter TR. If I hit an extra return, I get the following plot.



The commands for the measurement modes are nearly identical to the trigger modes. Note that the status line shows M_X to indicate that it is off. We want to watch for falling signal on A1, so we use the MF command to indicate falling and MS1 to use the voltage on channel 1. Note that the voltage level for the measurement point is set to 1000 or 1.000V. Hit another return to refresh the plot.



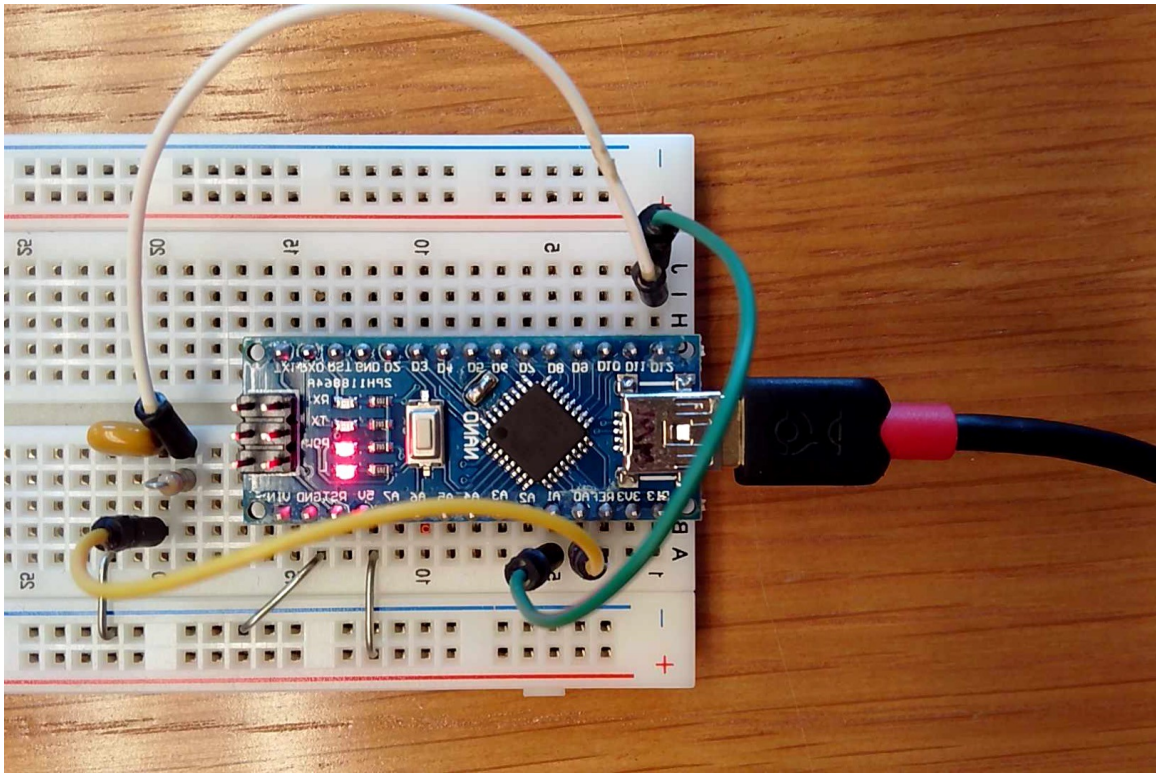
Note that the status line includes M_F_S1_L1000_T0.0090. This shows us that we are measuring to the point at which the voltage on channel 1 falls to 1.0V and the time from the trigger event to this crossing is 9mS. These lines are nearly vertical so the voltage level doesn't matter much. Let's make the pulse longer. In the command box, enter CCH200 and press Send. Press Send a second time to refresh the plot. We just told the program that Clock C High time is 200 pulses or 20mS and that matches the updated status line.



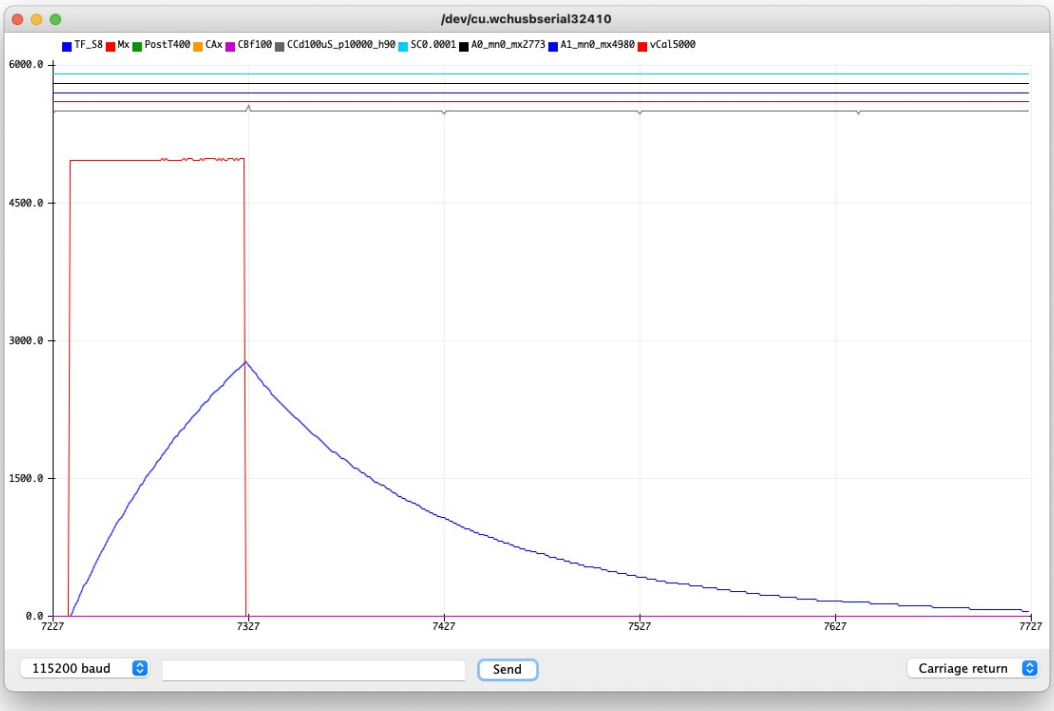
Voltage Calibration

The actual voltage levels are listed in mV and assume the voltage of the processor is known. It defaults to 5.000V but that is seldom the case. You can tell the program the actual voltage by measuring Vcc with an external meter and using the V command to enter it in. It is in mV so no decimal point here. This Vcc is saved in non-volatile memory so you don't need to enter it every time it is reset. You can see what the voltage is set to at the end of the status line. It should default to vCal5000 and it is limited to 4.5V → 5.5V. The A/D conversion is only using 8 bits so voltages are not exact.

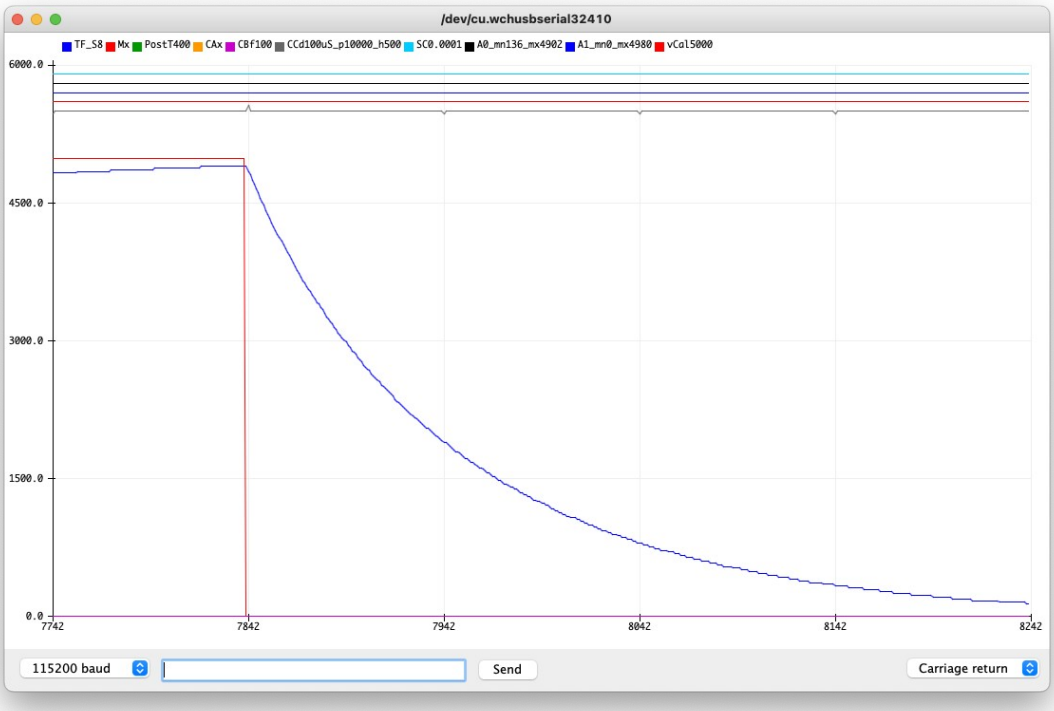
Running digital signals into an analog input doesn't really demonstrate what we can do. Connect a second wire to the Clock C output (D12) to a 10K resistor. Remove the wire from A0 to ground. Connect the A0 input to the other side of the 10K resistor. Also connect a 1uF capacitor to the A0 input. Ground the other side of the capacitor.



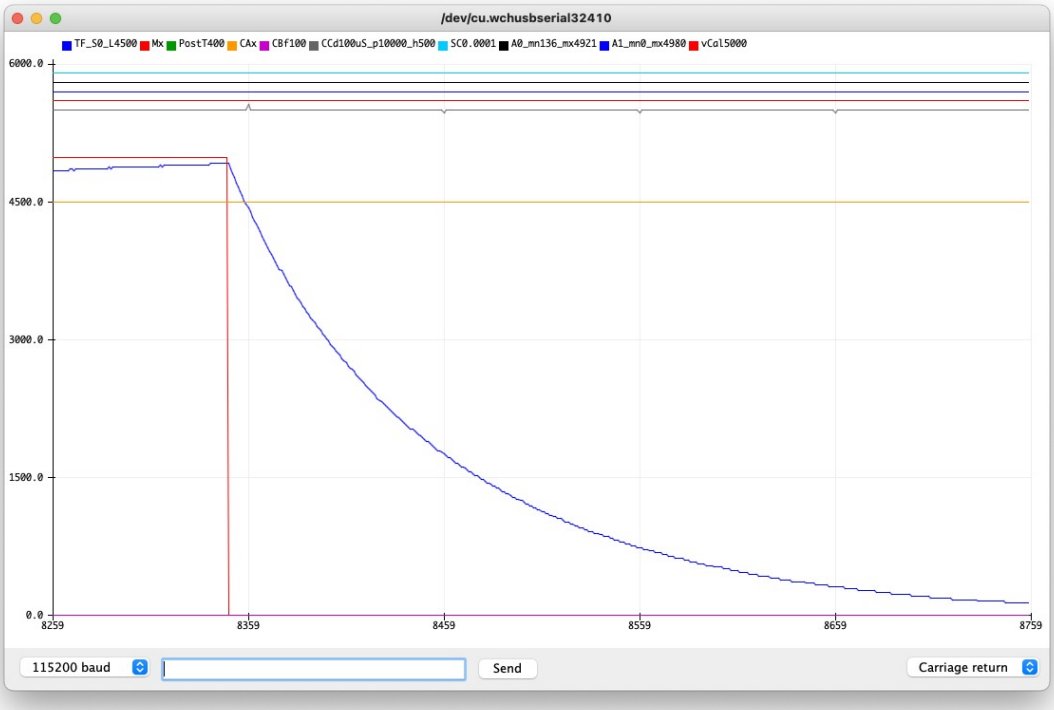
Reset the processor to get everything back to defaults and press an extra return to get a plot.



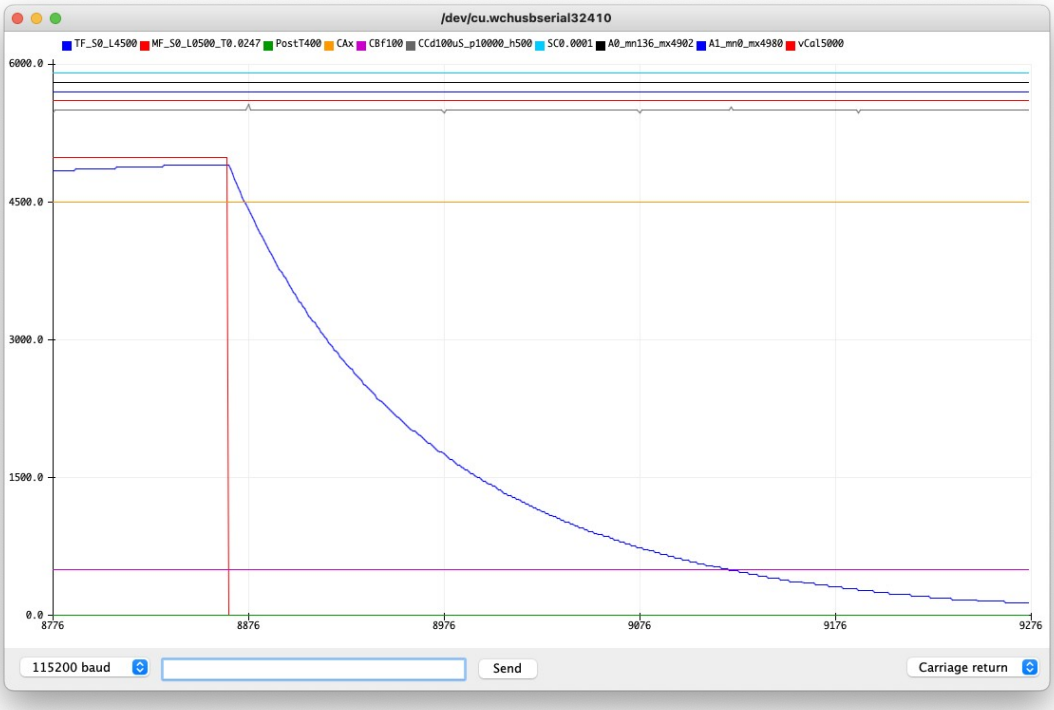
We get the pulse that we did in previous examples plus we now get the charge and discharge voltage of the capacitor. This pulse still happens only once per second and is only 9mS long. The cap charges to nearly 3 volts in the 9mS. I would like it to use a plot to estimate the size of the capacitor. To make it a bit better, I would like the cap to have more charge on it. The trigger is when the pulse drops, so I should just be able to make the pulse much longer. Use the CCH500 command to stretch the pulse to 50 mS.



I would like to time the drop between 2 known voltages. So I want the trigger to be at a known point. I just picked 4.5 volts as something easy to see. So to fix the trigger we use TS0 to say watch channel 0. Then commands TF and TL4500 (or a single command TF4500) to say we want trigger falling at a level of 4.5V.



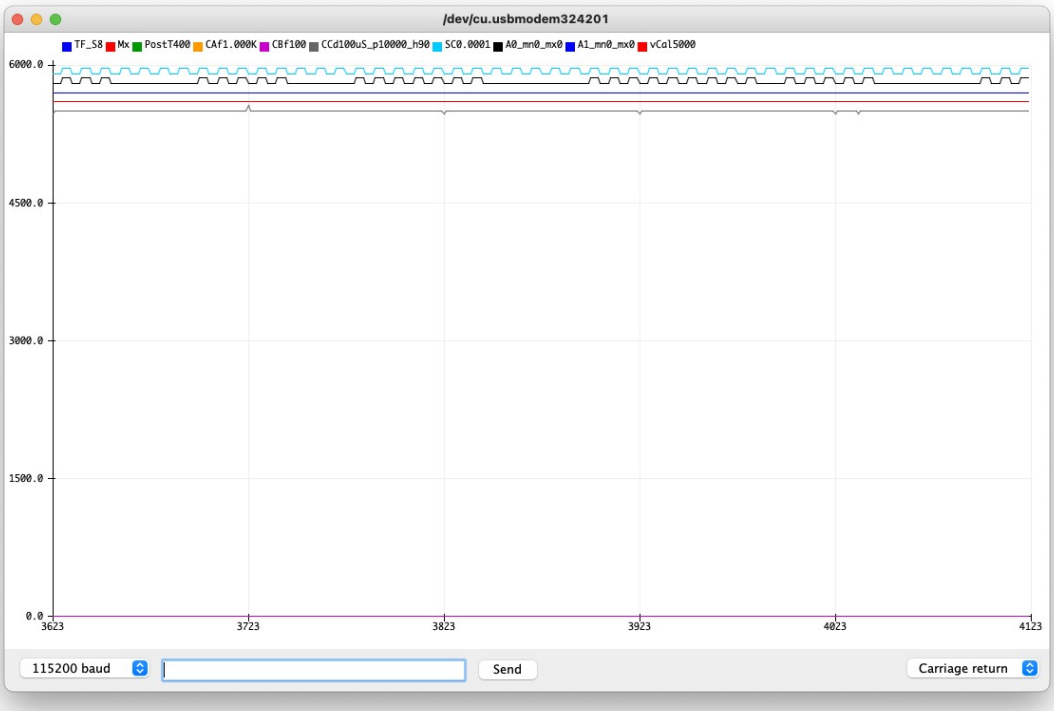
I now need to set the measure point to be at another voltage of the A0. I will just pick falling through the 0.5 volt level. So commands are MS0, MF, and ML500.



Although your values will likely vary some, in the status line we have M_F_S0_T0.0247. That is telling me that it took 24.7mS for the voltage to go from 4.5V to 0.5V. You can see in the trigger status line a large tick mark pointing up at the trigger point and a small tick mark pointing up at the point where the measurement is being taken. Assuming the 10K resistor is actually 10K we can calculate the value of the capacitor. 0.5/4.5 = exp(-t/rc). t/rc = 2.197 t = 0.0247 so rc=0.01124 10K for r gives us 1.12uF Within the tolerance for the parts we have.

Digital Inputs

The 4 digital inputs can only detect if a signal is high or low. They have been numbered 2 → 5 and show up on the serial plot output as the top 4 lines. To demonstrate the output, connect a jumper between the clock A output (D10) and the digital input 2 (D2). Reset the processor to get it into a known state and issue the command “CAF1000”. This sets the clock A frequency to 1KHz. Hit the extra return to get a plot.



Notice the top line is what we expected, the second line with pulses only sometimes is due to it floating. You can either just connect the unused inputs to ground or ignore the traces.

Clock Outputs

There are three different clock outputs labeled A, B, and C. Each has different restrictions.

Clock A is hardware based on Timer 1 in the ATMEGA328. The output is on the D10 line of the Arduino. It is a 16 bit timer with an adjustable scale factor on the front end to adjust the time per tick. If you just need a square wave clock, you can use the CAF command to set the output to a square wave at the specified frequency. To get a 1KHz square wave, CAF1000 would work just fine but the software will allow some flexibility on how you input values. For example if you wanted 1,200 Hz, you could enter it as “CAF 1200” or “CAF 1.2K”. It also understands M for megahertz. The frequency range low end is 1Hz and the top end is 8MHz. If you want control of the duty cycle, you can set the divider, period, and ticks high. The input to the timer is fixed at 8Mhz. This would give us 125nS per tick. There is a divider that we can adjust. It is set with an integer value from 1-5. The following table shows the time period for the different divider values.

n=0	off
n=1	125nS per tick
n=2	1uS per tick
n=3	8uS per tick
n=4	32uS per tick
n=5	128uS per tick

Once the divider is set to where we want it, we can set the period of the waveform with the CAPn command where n is the number of clock ticks. This timer can be used to control a hobby servo motor. After connecting the power and ground, connect the signal wire from the servo to the clock A output (D10). The hobby servo needs a period of 20mS and a pulse high time of 1 → 2 mS. I can use a divider of 2 so I would use the CAD2 to set the divider. I would set the period to 20mS with CAP20000. The period high time could be set at 1.5mS with CAH1500. This should center the servo motor. I can then give more CAH commands with values between 1000 and 2000 to give me the range desired. Clock A is a 16 bit timer so values in the range of 1 → 65535 are allowed.

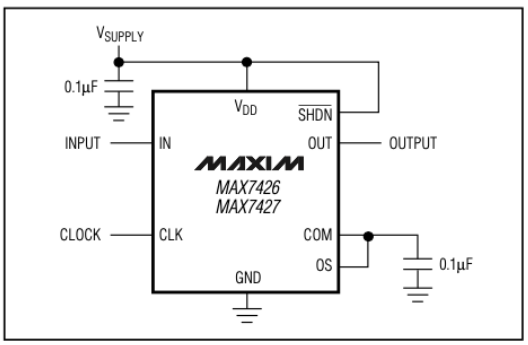
Clock B is based on Timer 0 and is only an 8 bit timer and its output is on D5. It has the same adjustable scale factor as clock A, but the counts for the registers are limited to a range of 0->255.

Clock C is a software based pulse generator with an output on D12. It has a fixed tick time of 100uS. It uses 16 bit variables to determine period and pulse high time.

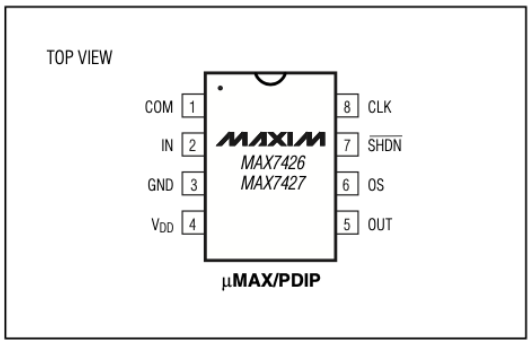
Sine Waves

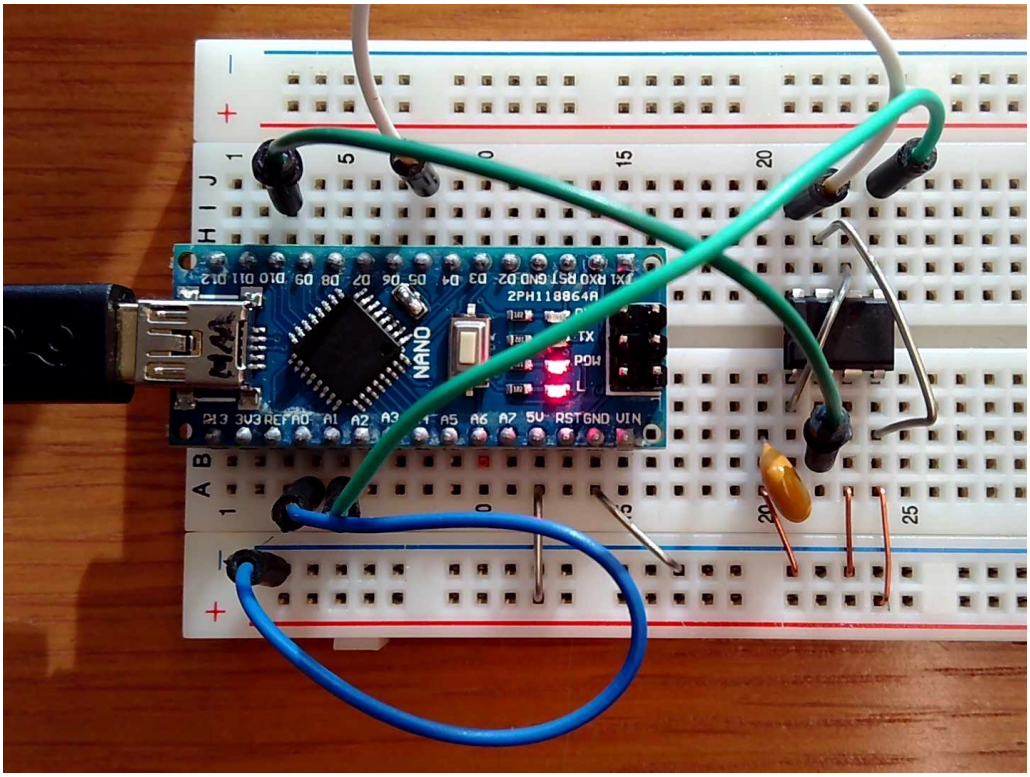
All of the outputs on the processor are digital, we can not generate a sine wave with just what we have. Although there are a couple of ways to generate sine waves, we added one additional part, the MAX7426 switched capacitor filter. We use the concept that a square wave is the sum of odd harmonics of the fundamental frequency. Basically if I filter a square wave, I can make a sine wave. That is what we will do here. Be careful connecting up the MAX7426. If you wire it incorrectly, you can destroy it. We will also use both clock A and clock B to get what we want. To wire it up use the following information. Connect the Vdd to the +5V line and the GND G pin. The CLK line needs to be wired to clock B output (D5) and the input needs to be wired to clock A output (D10). Connect the OUT of the filter back to the A0 pin so we can see what it looks like. Connect the A1 pin to ground so it doesn't follow along.

Typical Operating Circuit

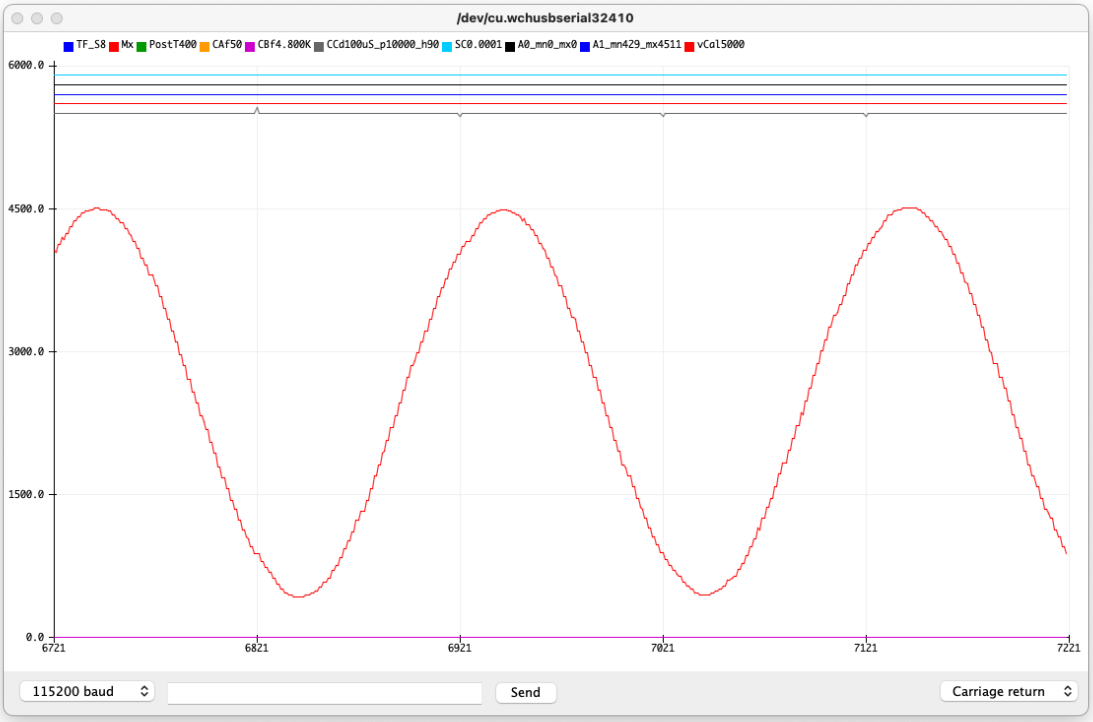


Pin Configuration





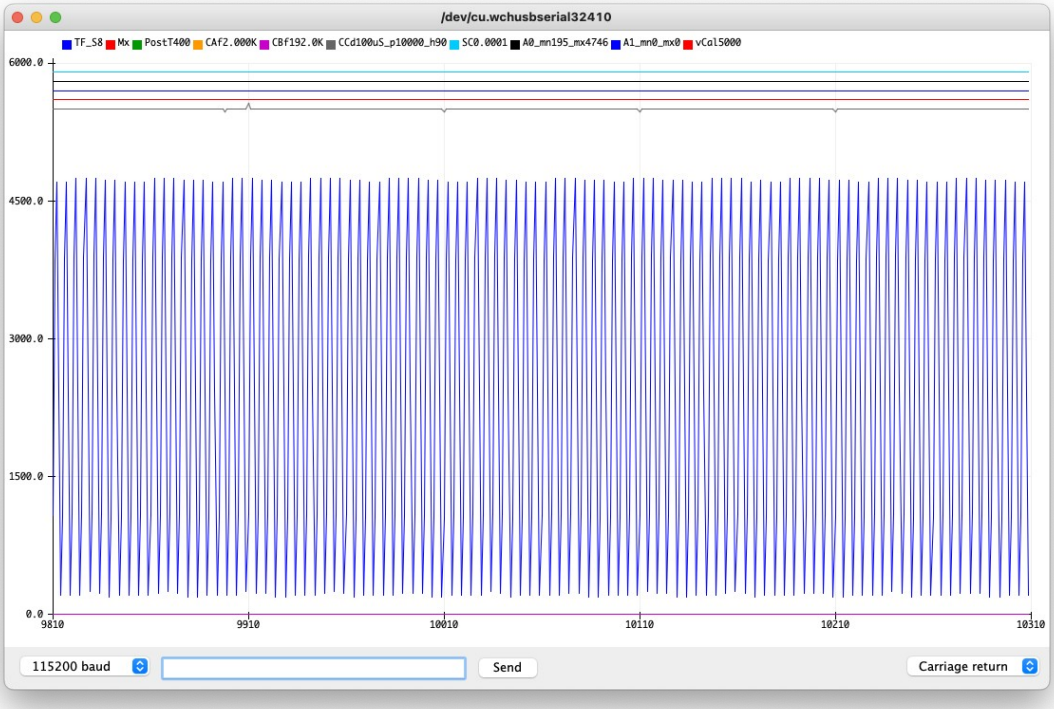
In the serial plotter, reset the processor so you know what state everything is in. Issue a “CS50” command to give us 50Hz on the output. Redraw the plot.



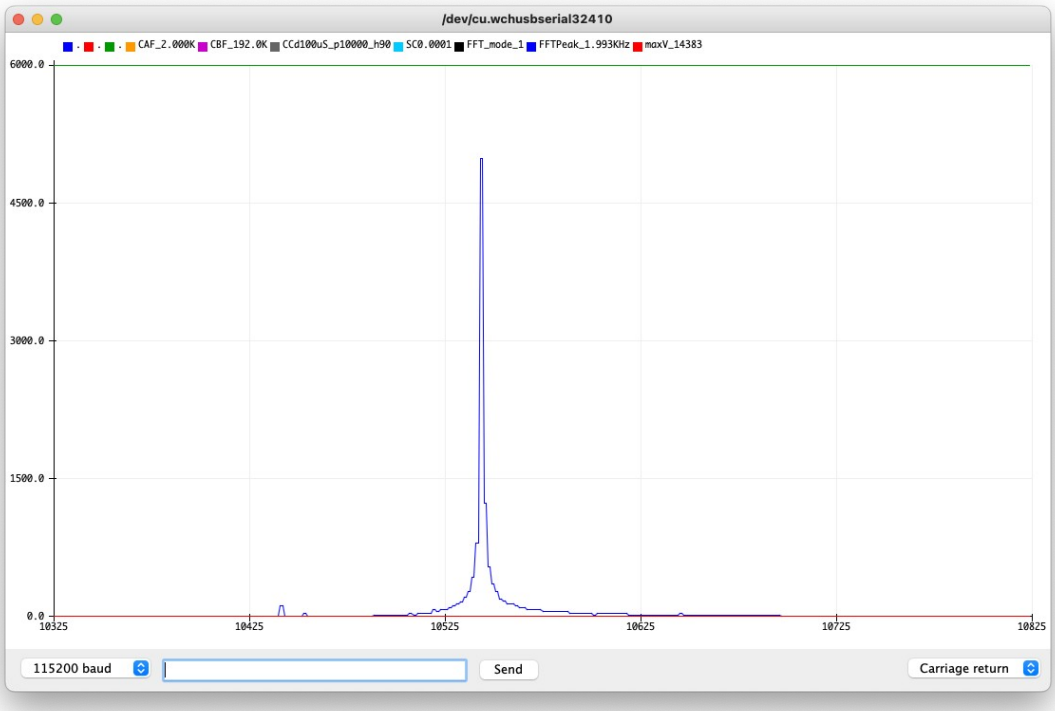
The frequency limits are significant. Going up to 1K already makes the ability to get any phase information difficult.

FFT

Frequency analysis was added more as a challenge than a serious useful tool. Running very low on memory required compromises to be made. The 512 data points gets cut in half to 256 and we can only look at the A0 channel. From the previous setup, run the output of the filter chip into A0. Use the CSF2.0K command to set the sine wave output to 2KHz. Re-plot it.

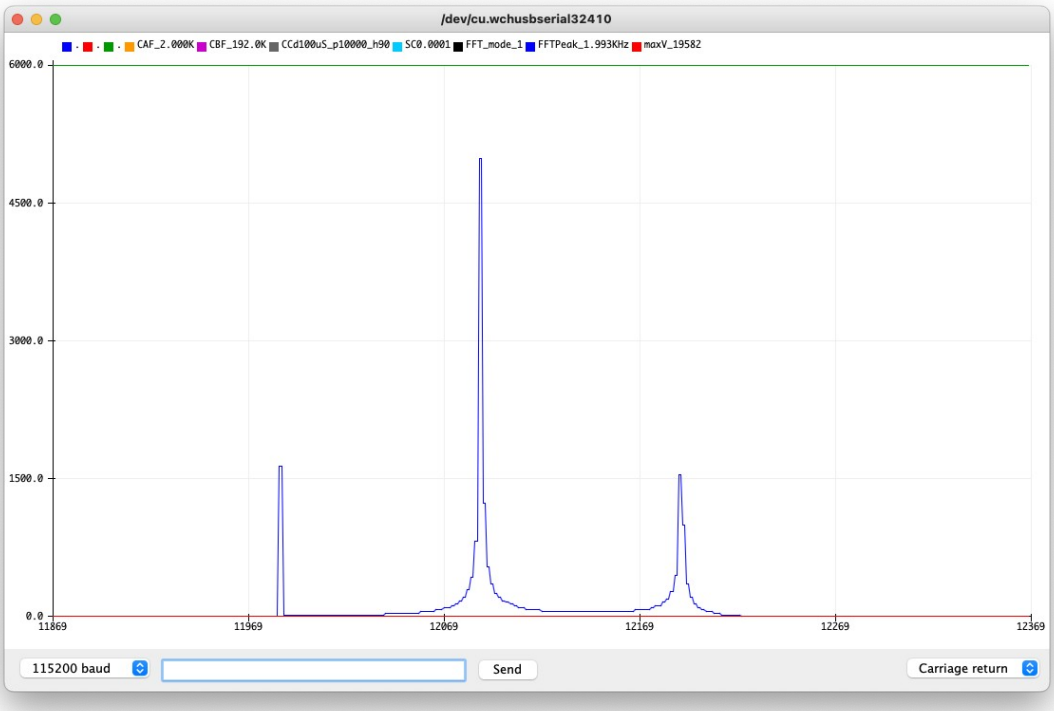


Now issue the F1 command and re-plot it.

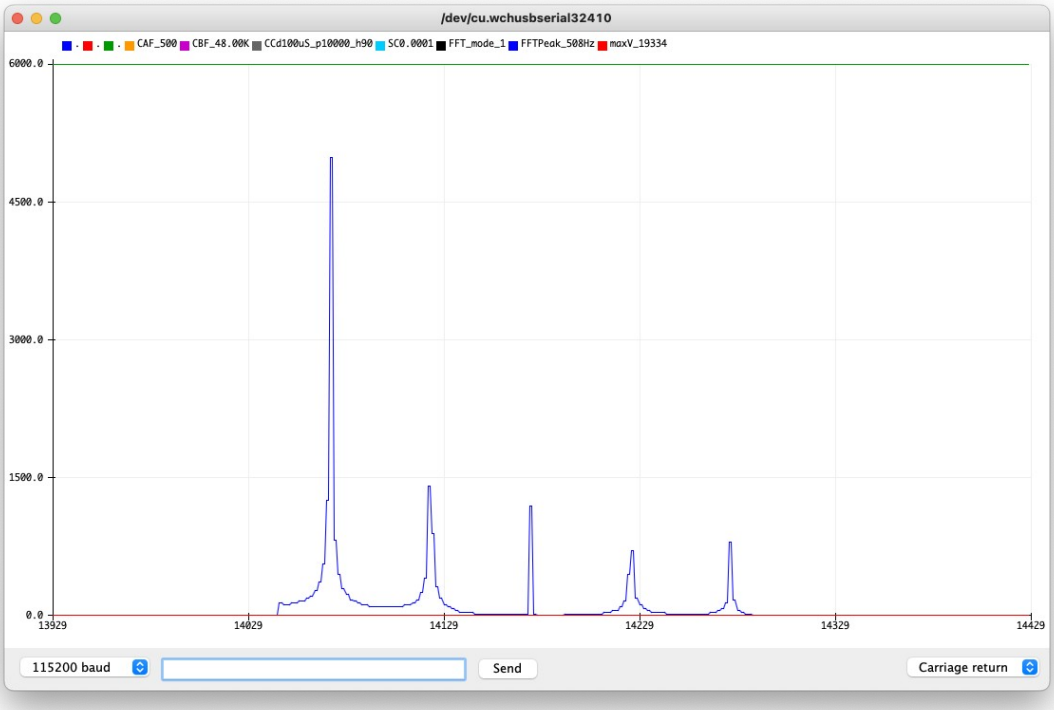


Notice the status line has changed significantly. It does tell us that we are in FFT mode 1 and that the peak is at 1.993KHz. The maxV is a bit less useful. But the plot does show one peak really close to the expected 2KHz.

Now change feed the clock A output (D10) directly into the A0 input.



Even though we are feeding in the same frequency, it is a square wave, we now see other peaks. Use the CSF500 to drop the frequency down to 500Hz and re-plot it.



How does it work?

Even though this compiles and runs with the normal Arduino environment, I needed to be able to control exactly what was going on so I do not call any of the Arduino libraries. Instead of using setup() and loop() routines, I use main() as this bypasses code not needed. I also always place routines in a specific order. The main() routine is the last thing in the file. In general if I call a subroutine, it will be located earlier in the file. This prevents the need for function prototypes. All registers to setup hardware are done directly. Execution starts with main() where we start by setting up the timers and the serial I/O port. The important timer here is Timer2, it is set to interrupt every 100uS, but interrupts are not enabled yet. The A/D converter is setup next with the 16 Mhz system clock divided by 32 to give me the 500Khz I want and also take over A0 and A1 for the analog input. Interrupts are not enabled yet. Variable initialization is setup next, here is where the default settings are located. The rest of the I/O ports are also setup to be in the states we want. Interrupts are enabled at this point and we push the version information out the serial port. At this point we enter the while(1) loop and never leave it. Everything after this is triggered by interrupt routines.

To hit the 10Khz samples on two channels requires the A/D conversion to be way less than 50uS. Under normal operation the 16Mhz clock would be divided by 128 to give you 125Khz for the A/D clock. The conversion take 13 ADC clock cycles at 125Khz or 104uS, this is already less than the 10Khz on just one channel. The A/D converter on the ATMEGA328 processor is a 10 bit machine. We can clock it over the 200Khz recommended rate if we only need fewer bits. I set the pre-scale to 16Mhz / 32 or 500Khz but only using 8 bits. The settings allow for a 26uS conversion time. I do need a bit of time for updating buffers, so reading both ports and processing takes close to 60uS. The A/D conversions are interrupt driven and is running most of the time.

With only 2048 bytes of ram, I decided to use two ring buffers, each 512 bytes in length to hold the results from the A/D converter channels. I use ring buffers so that we can continually read values and stuff them into memory. I later figured out that I still had enough memory to add a few more features. I added 4 digital inputs and they each have their own ring buffers. I do bit stuff the bytes so I only needed 64 bytes for each for a total of 256 bytes. These 6 buffers takes up 1,280 bytes of my 2048 available. There also a bunch of global variables to keep track of everything going on. As of version 1.4.2 we are using 1795 bytes out of the 2048. Those other 253 bytes are needed for local variables and and stack. You will get a compiler warning about being low on RAM, but haven’t seen any problems with running out.

The ATMEGA328 also has 3 timers built in, TIMER0 and TIMER2 are 8 bit and TIMER1 is 16 bit. I wanted to keep TIMER1 for clock output. Almost everything is interrupt driven starting with the system tick timer. We use TIMER2 to generate the 10KHz system tick that triggers each data sample cycle. This 100uS tick interrupt routine starts the A0 A/D conversion cycle and samples all 4 of the digital channels in one shot as they are on the same port. When the A/D conversion is done it generates its own interrupt. In this routine, if just completing A0, it starts the conversion on A1 and returns to the main loop. Once the A1 A/D conversion finishes, it generates another interrupt to the same vector. This time if adSampling is true we load up all of the ring buffers and advance the pointers. We also take a look at all of the trigger and measure settings. The trigger and measure state machines are also located in this area. It makes the routine look really long, but it should still be very fast. Once the system has been triggered and the count of samples satisfied, the adSampling is turned off and we wait to unload the data. For a rising trigger, the measured value on the channel must measure at least one sample below the selected level before it starts looking for a point higher or equal to the trigger level. Falling trigger works the same way with comparisons reversed.

When a trigger event occurs, there is another value that determines how many points are to be collected after the event. This If you set the value to 512 then it will rewrite the full ring buffer and all of your data displayed will be from after the trigger. By setting this value to 400, you get to see what happened 100 points before the trigger. You can set this value to 65535 and it will collect 6.5535 samples worth of data, but you will only get to see the last 500 points. The default value is set to 400 but you can adjust that with the P command. While it is waiting for a trigger event, you can still enter commands and adjust the level.

While the A/D system is sampling channels the LED is blinking at about 10hz (0.1024S or 9.675hz). With only 512 locations to fill up and 100uS sample rate, we fill the buffer in 51.2mS. This is the maximum speed of the system. We can go slower by setting the skip parameter and forcing the program to ignore reads. The SC n command can set the skip parameter to what ever you wish. If you set it to 50 or more, you can also force the output into a smooth scroll mode with the SS command. The SN will return it to normal mode. Setting SC 10 will sample 1,000 times per second where as setting to 10,000 will get you to once per second.

Characters from the keyboard also generate interrupts and are processed as they come in. Once it hits the 0x0D Return character, the line is passed to the processCommand() routine.

Commands are 1 or 2 character sometimes followed by an argument. A space between the command and the argument is optional. If the command is 2 characters, you cannot put a space between them.

After a trigger has completed the display is updated. You need to hit a carriage return with no arguments to get it to trigger again. There is a command TH n to force a re-trigger after n seconds. So you can have the display auto update after 5 seconds or so. The system isn't that fast so 5 seconds works ok.

There is a measure feature that works just like the trigger function. It will tell you how long between two events. The resolution isn't all that great due to only sampling at 100uS. Again the condition must be false before it can measure trigger can be true. If the condition is never met, a value is still printed but is bogus. There is a status line in the plot that needs to be looked at.

Outputs:

There are three different clock outputs labeled A, B, and C. Each has different restrictions.

Clock A is based on Timer 1 in the ATMEGA328. It is a 16 bit timer with an adjustable scale factor on the front end to adjust the time per tick. The user has full control over the divider. If the user enters a specific frequency, the program calculates the best divider to get there.

Clock B is based on Timer 0 and is only an 8 bit timer. It has the same adjustable scale factor as clock A, but the counts for the registers are limited to a range of 0->255.

Clock C is a software based pulse generator with a fixed tick time of 100uS. It uses 16 bit variables to determine period and pulse high time.

The output plot

Looking at other Arduino programs, they pointed out the Serial Plotter option, I didn't know it existed. It appears to plot 500 data points in the X direction. The Y direction is scaled to what ever we need. The A/D converter gives us values from 0x00 → 0xFF. Those are scaled at plot time to the actual voltages using the vCal variable. The Y range I selected covers 0-6000. The important point is that only the last 500 data points are visible even though 512 data points are collected. The only text output options with this is the key at the top of the chart. I will use that to display information. Note that at the bottom, you must select 115200 baud and also "Carriage return".

The top 4 lines of the plot are the 4 digital inputs. The 5th line has tick marks that point up or down. The up pointing tick marks are of two sizes, the larger one indicates where the trigger happened. The smaller tick pointing up is the point at which we measure to. The downward pointing ticks are multiples of 10mS from the trigger point. Below that you will have the A0 and A1 input traces along with the levels for trigger and measure if those are set.

FFT calculations

Good luck with this but I will give you a few hints. It is all done in 16 bit fixed point. I wrote sin and cos functions but they don't use degrees. They use 0→65535 to represent an angle. The two main arrays that hold A0 and A1 signals are 8 bit unsigned ints 512 bytes long each. I couldn't get the FFT to work with 8 bit due to overflow issues. We are out of memory so I couldn't expand it. I used the union function in C to place 16 bit signed integer arrays with a length of 256 over the top of both. One of those is the real part, the other is the imaginary. Because we are only looking at the A0 channel, the memory for the A1 channel is used to hold the real part. Once the values have been converted from 8 bit A0 to the 16 bit real part in the A1 space, A0 is zeroed out. Once the FFT returns, we still need to convert back to magnitude and phase. We also need to scan for the largest value and scale everything back down. The actual output is back to 8 bit values. If you look at the code the square root function is inline and the phase does call an atan2() function. In terms of where everything is called from, it is done right before the plot is output.

Future changes

As I said, we are really out of RAM space. But I might be able to squeeze in enough for a network analyzer. Maybe 32 points at fixed frequencies, but that will wait for another day.