# Data Capture on Mac or PC

When you are working with microcontroller boards like the Arduino family, it can be difficult to see what's going on in the code, especially if your code is real-time. For one thing, the Arduino IDE doesn't offer any way to examine the code while it's running, and even if it did, you can't set breakpoints very effectively in any code that has to run real-time, simply because the breakpoints affect the timing of the code execution.

There are a number of ways to monitor the activity of your code without affecting its operation. At least, not affect it all that much. Some of the things I've done in an attempt to see what's happening include:

* Add some LEDs that come on when certain decisions are made in the code.
* Write a signal to one or more digital port bits and watch those on an oscilloscope.
* Use a PWMgenerator running very fast to implement a DAC and look at the analog output on a scope. I have an article on this coming soon.
* Send information out the serial port and capture it on a Mac or PC.
* Use a bluetooth link to do the serial port idea but without wires.
* Use Microchip's MPLAB X IDE

For this article we're going to look at a simple way to get data from an Arduino-family project into a PC or a Mac. We'll show how to send the data over the serial port line and get that data into a Python app. Using the Python code we can examine the data on the console or send it to a file, or maybe process it with some algorithm in Python.

Real-time applications often have sensors for voltage, current, digital input pin state, remote devices on IIC busses, and much more. The first example (NAD MAYBE THE ONLY ONE) shows how to take some (simulated) analog input data, format it and send it over the serial link. We'll look at how to grab the incoming data on a Mac or PC with a python program that can also write the data to a file.

Most of my own real-time applications deal with voltages, or currents, as inputs fed through some sort of voltage divider to an analog input. To keep things simple, the example code generates a pair of sine waves digitally and we sample the values as though we were getting the data from an analog input using analogRead. In later article I'll show how to do very fast analog sampling suitable for looking at motor drive waveforms or some audio signals.

For this example we have two varying values that we want to capture. When post-processing real-time data we generally need to know the time the data was sampled, so in the example we send a record containing three values: time, voltage 1, and voltage 2.

To keep the example simple we'll just send the raw values as though they were from the analog to digital converted (ADC) and not worry about actual voltages, since we can easily do that conversion in Python later when we capture the data, or in our post-processing code in MATLAB or Octave, or whatever.

We have a choice as to how we send data over the serial link. Data from the ATmega ADC is essentially an unsigned value in the range 0 to 1023. That's just 10 bits of data so we usually store that in a uint16\_t (or just unsigned int) on the processor. If we want to send that over the serial link, we could choose to do so by sending each sample as two bytes. But for our fist example we'll format the data into an ASCII string and send it as a comma-separated line of text. This makes the data bigger but allows us to see it on the Serial Monitor to verify that the code is working. We will look at sending binary data a bit later (MAKE SURE WE DO).

There is no real-time clock on an Arduino so the best we can do is capture the time since boot. This is done very accurately on the ATmega processor by counting CPU clock cycles. We can access the data easily with a called to millis() to get the time in milliseconds, or to micros() to get the time in microseconds. In both cases the time returned is an unsigned 32-bit value.

The worst case is to use microseconds snice the ASCII representation is longer.

So for out example we want to send the time in microseconds, and two ADC values that can range from 0 to 1023. So how big is the line of text? We need to know this to estimate how fast we need to clock the serial interface. Let's say the app has been running for 10 minutes and we are getting analog values near the upper limit. Out data might look like this:

600000000,1020,1019<lf>

Where <lf> means the newline character ('\n' in the code). The entire string is 20 characters long, including the newline at the end. 20 characters is 160 bits. But over a serial link each character has a start bit and at least one stop bit, so a single character needs 10 bits and our full string will need 200 bits.

Now let's say we want to sample the data every 10 milliseconds. That's really not all that fast for a real-time application but it will illustrate what we need. That means that every second we'll send 100 of our text lines which adds up to 20,000 bits. So our serial link needs to run at 20,000 baud or faster to keep up with the data rate.

If we take a look at the Serial Monitor that is part of the Arduino IDE, we see that it defaults to running at 9600 baud which is pretty slow (although I can remember when a 9600 baud modem was almost too fast to comprehend). Looking at the baud rate window we see that we can go a lot faster than 9,600. I usually run at 115,200 for no other reason than it's pretty fast and happened to be the speed an app I downloaded a long time ago used. You can experiment with faster speeds of course. The reality is that the only actual serial data controlled by this rate is between the main ATmega processor on the board and the USB interface device. From there on it's all running over a USB connection that is limited by whatever else is on the same bus and how fast the USB interface to our computer is.

The example code for the ASCII version can be loaded into an Arduino family board. If you load it and look at the serial monitor (at the correct baud rate) you'll see data like this:

24294368,956,268

24304544,952,256

24314720,944,244

24324896,940,236

24335072,932,224

24345248,924,216

24355424,916,204

24365608,912,196

A word of caution here: The Serial Monitor doesn't really like to run for a long time with huge amounts of data zooming by. I think the Java code runs out of memory at some point and the entire IDE grinds to a halt. We'll fix that in a minute, but for now, upload the data\_cap\_ascii\_example and then start the Serial Monitor to see the data just for a while. Then close the Serial Monitor.

## Getting the Data in Python

You can capture the data from the serial port using some Python code. The serlog.py code included here will open the serial port you specify then copy the ASCII data records to a file.

To see how to use the app, print out it's help like this:

python serlog.py -h

The file is named with a date and time stamp so it's unique. The filenames look like this:

2022-01-10-103840-log\_data.csv

And the data in the log file is just the ASCII text we're sending, like this:

2327692,196,912

2337868,204,916

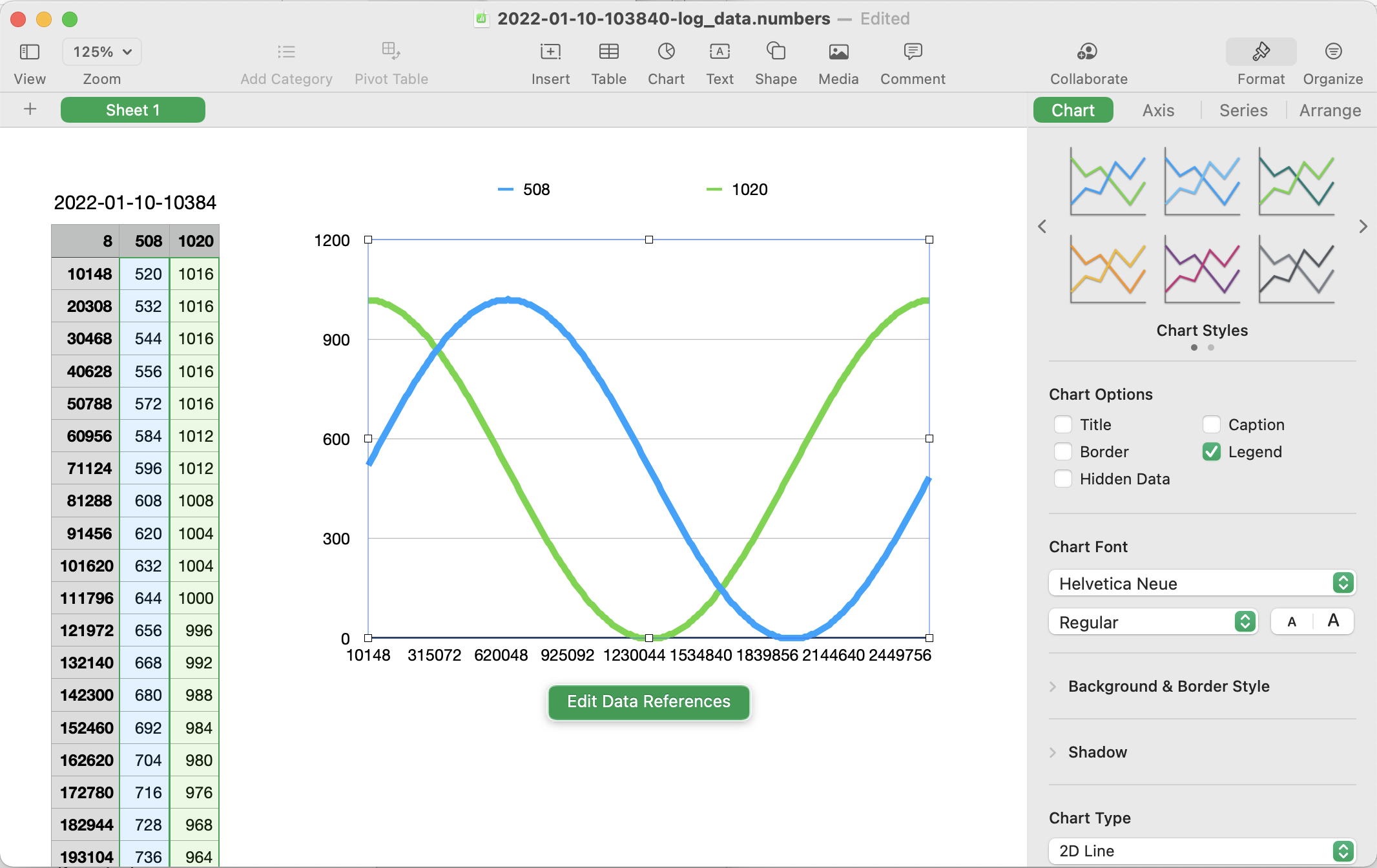
2348044,216,924

Use the -v option to see the data on the screen as it's logging it.

## Processing the Data

There are a lot of ways to process the data from the CSV file. You can open the file directly on a Mac in Numbers, on a PC in Excel, or with Google Sheets. And if you're doing some real data processing you can import it into MATLAB or Octave.

Here is a screenshot from Numbers of the data from the test app:



## Binary Data

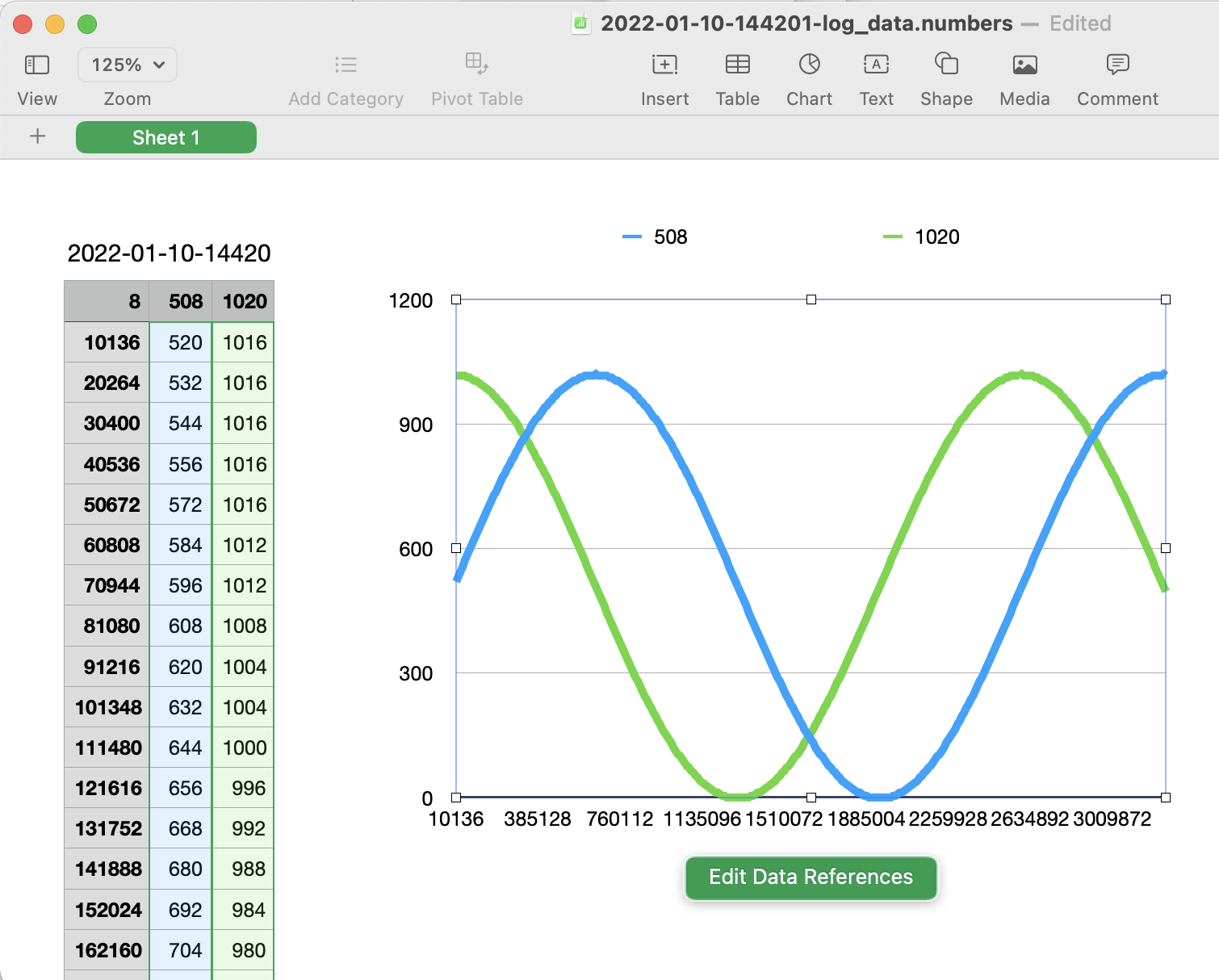
If you have a lot of data to send and want to be a bit more efficient, you can send the data over the serial link in binary. There are lots and lots of ways to achieve this but we'll go for something simple as an example.

When we sent ASCII data as lines of text we had a format that was both easy to send and easy to receive. Sending binary data is also very easy - in fact it's easier than sending text since we don't need to format it as ASCII characters first. But receiving it it a bit more complicated because an arbitrary stream of binary data has no obvious start or end points. We need to mark the start in some way and perhaps the end as well so that when we start the receiver mid way into a transmission, it has some chance of synchronizing with the individual records.

The simplest thing we can do is send a number of bits at the start that cannot naturally appear in the data. If the data is really random, that's not possible, but in our simple example here it certainly is. Our analog samples are only 10 bits but we can package those into 16-bit words conveniently. The largest value is can send is 1023 or 0x3FF. The timestamp is a 32-bit value, which gives us a range of a little over an hour if we are using microseconds and almost 50 days if we are using milliseconds. So we can probably use 32 bits of 1s as a sync marker and keep mostly out of trouble.

We won't get any fancier than that here expect to say that the receiver will wait until it's seen the start marker (4 bytes of 0xFF) and then collect the data record. After that, the next thing it sees should be the start marker again. If it doesn't see that then it's not in sync and it can start again. We're going to assume that the receiver knows the exact structure of the transmitted data. If that's not the case, for example if the data format is variable in some way, then the protocol will need to be more complex but we can't fit that into one small article, so I'll leave that to you to ponder.

When you upload the data\_cap\_binary\_example don't try to watch the data in the Serial Monitor. All you will see are funky characters :).



The Python code to receive the binary data is a little more complex. We can't just ask the serial interface to read a line of text as we did in the ASCII case, but we can write a function to read one of our records and then use the struct library to decompose the fields in the record. Having done that we can convert it to CSV and send it to the log file.

The code in serbinlog.py does all of that, and you can see the resulting plot from Numbers for a short sample of the data that was transmitted in binary.

In a later article I'll be showing how to do data logging like this over a Bluetooth connection so that we can get rid of the wires.