POSCOPE - SOFTWARE OSCILLOSCOPE

GERRIT SLOPSEMA

Abstract. We outline the generic structure of oscilloscopes and the particular structure of Poscope, a software oscilloscope. Poscope, coupled with the NI USB-6009, acts as an oscilloscope suitable for simple tasks and limited frequency ranges.

1. Introduction

The oscilloscope was invented in 1897 by Karl Ferdinand Braun[3]. Braun was experimenting with electron beams and found that a screen coated with phosphor illuminates when bombarded with the electrons. From this he made the first cathode ray tube. He applied this to measuring voltage versus time and created the oscilloscope. Braun was awarded a Nobel Prize due to his work in wireless telegraphy.

An oscilloscope is, conceptually, a very simple device. It takes a time-varying voltage input and displays the data as a graph of voltage vs. time. A simplified schematic[1] of a generic oscilloscope can be seen in Figure 1. In this schematic, the voltage input is fed through a voltage attenuator which increases or decreases the voltage to fit the voltage levels to the volts per division setting on the scope.

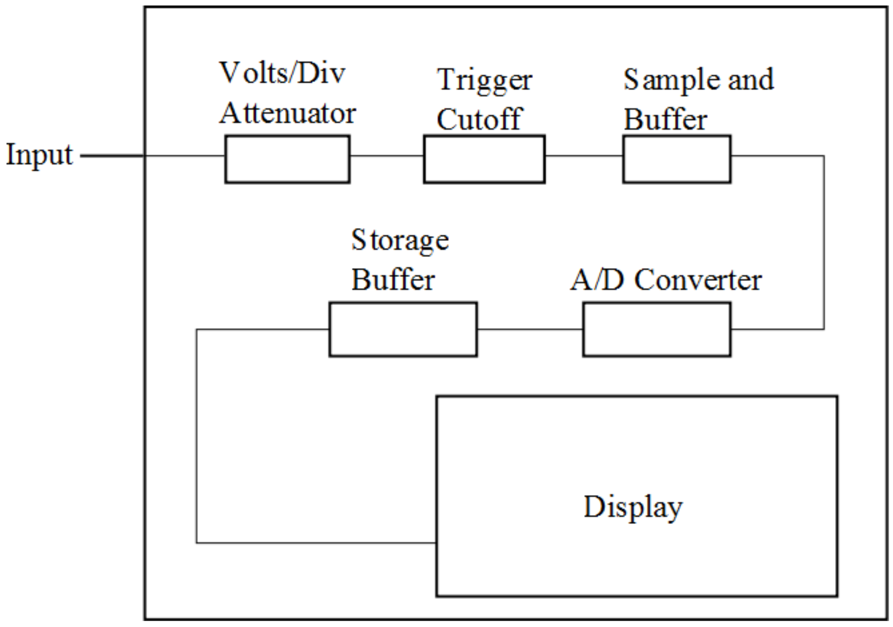

Figure 1. A generic oscilloscope schematic.

Date: December 9, 2011.

The data is then routed to the trigger cutoff where the scope reads until a voltage equalling the trigger level is found[1]. There are two methods of triggering: rising slope (or edge) or falling slope (or edge). In the first case the scope searches for the trigger value when the voltage slope is positive. In the second case the scope searches for the trigger value when the voltage slope is negative. The scope only sends the data on after these two conditions (either positive slope and voltage greater than trigger or negative slope and voltage less than trigger) are met. A diagram of this can be seen in Figure 2.

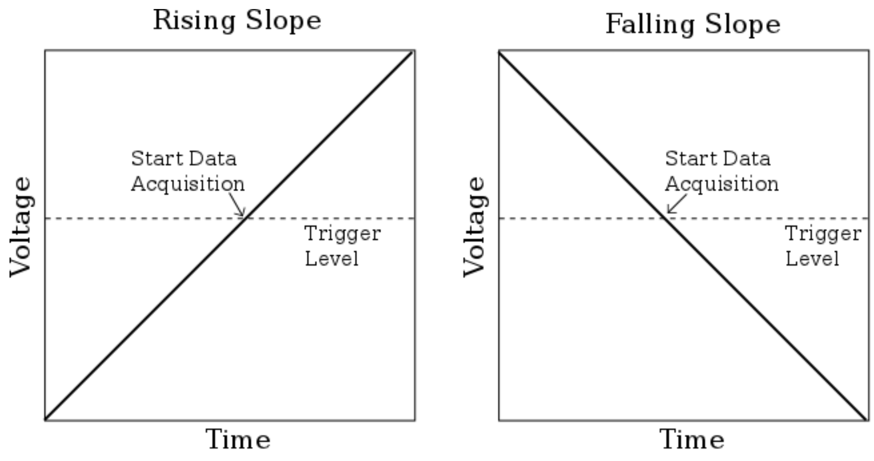

FIGURE 2. Rising and falling slope triggers.

After the voltage reaches the trigger level the scope sends the voltage stream to the sampler. The sampler takes the continuous analog voltage signal and *samples* data points from it. For example, in a stream of voltage 500 ms long the sampler may record 1000 voltage levels, each spaced equally in the 500 ms timeframe (reading at a rate of $\frac{500}{1000} = 0.2\frac{\text{samples}}{\text{ms}} = 2000\frac{\text{samples}}{\text{sec}}$ in this case). The sampler stores these values as a discrete approximation to the continuous voltage stream.

Once the data is obtained the scope can display it on the screen. The scope must accomodate two variables in this display: the volts per division and time per division. In effect, the volts per division dictate how tall the voltage graph appears on the screen. The scope's screen has a fixed number of horizontal lines acting as reference for the voltage levels. Thus the scope must display the data relative to these lines. For example, if the screen has 5 horizontal lines and the volts/div setting is at 1 volt/div, then the maximum voltage difference viewable on the screen is 5 volts. The scope must scale the voltage image according to the size dictated by the volts/div. The same idea applies to the time/div variable, only along the horizontal axis.

2. Methods

2.1. **General Structure.** The Poscope program is a software oscilloscope written in the Python programming language. It uses NI-DAQmx, a software library provided by National Instruments for access to National Instruments devices, NumPy, a scientific computing library, and Qt/Qwt, a library for graphical user interfaces (Qt) and graphical plotting (Qwt).

Poscope uses a device from National Instruments (NI USB-6009) to read voltage data. A voltage source is connected to the NI USB-6009 (with the positive into analog input

"AI0" and negative into "GND") and the NI USB-6009 is connected to a computer via USB. Poscope reads from the NI USB-6009, manipulates the data according to the user's specifications, and displays it on a graph.

Poscope's internals differs from Figure 1 in the order in which the data is processed. All the steps are present, however the order is as follows:

(1) Voltage is input to the NI USB-6009.
(2) The NI USB-6009 samples the voltage (sample rate controlled by Poscope) and converts from analog to digital.
(3) Poscope reads the data from the NI USB-6009 into a buffer.
(4) Poscope applies the trigger cutoff.
(5) Poscope manipulates the data to account for scaling, volts/div and time/div, and vertical positioning settings.
(6) Poscope graphs the data on the screen.

The above constitutes one plot cycle.

2.2. **Definitions.**

We will use a few variables defined here:

n	Number of data points graphed per plot cycle.
r	Sample rate of the NI USB-6009 (in samples per second).
t_{div}	Time per division (in seconds).
n_t	Number of time divisions on the graph/screen.
V_{div}	Volts per division (in Volts).
n_V	Number of voltage divisions on the graph/screen.

Note that n/r is the time duration of the sample and $n_t t_{div}$ is the time duration displayed on the graph.

2.3. **Triggering.**

The NI USB-6009 has triggering capabilities[2] but I was unable to make it work. It is, however, a simple procedure to implement in software. I considered a few triggering methods, some complicated and some easy, and settled on a simple method which works for *most* cases. Assuming the voltage data is periodic and the period of the data is less than $n_t t_{div}$, Poscope requests $2n$ data points from the NI USB-6009. Once Poscope has the data, $d = \{d_1, d_2, \ldots, d_{2n}\}$, it searches through d until it finds a d_{i-1} and d_i such that d_{i-1} is less than the trigger level and d_i is greater than the trigger level. This is an example of a rising-slope trigger. Once these d_{i-1}, d_i are found Poscope reads the next n data points from d and uses that as the data to graph.

Because we assume the period of the data is less than $n_t t_{div}$ we are guaranteed that there are n data points after d_i. In the "worst case" scenario (with t_{div} set accordingly), the graph will display n data points (that is, $n_t t_{div} = n/r$). However, since the period of the data is less than $n_t t_{div}$, the period is also less than n/r. So if the voltage level reaches the trigger level in one period, this "double-buffer" method catch the trigger level and have n data points to display on the graph.

2.4. **Vertical Position.**

The data read from the NI USB-6009 is not always centered around 0. For example, with a sine wave input from a function generator, the recorded voltages may vary from 1 V to -2 V, rather than the desired 1.5 V to -1.5 V. To determine the proper vertical shift required make the voltages symmetric about the horizontal axis, we use the following algorithm:

```
1  amplitude = (max(data) - min(data)) / 2
2  correction = amplitude - max(data)
3  data = [x + correction for x in data]
```

Because the data is periodic, it has a set amplitude. Since we want the data to be centered horizontally around the time-axis, we want the maximum of the data to be at the amplitude. So, line 1 gives the amplitude and line 2 gives the deviation of the maximum from that amplitude, the correction. Then each data point is corrected by adding the correction to it. The result is a shift up or down so that the data is centered around the horizontal axis.

2.5. **Plot Scaling - Time.**

The NI USB-6009 can sample data at varying rates. The maximum sample rate[2] is $r_{max} = 48{,}000\ \frac{\text{samples}}{\text{second}}$. Further, the number of samples graphed, n, is variable. Both must be considered when determining how much data to use for the plot. Usually some of the data read from the NI USB-6009 will be discarded. This is due to the time/div setting. The following algorithm determines the number of data points to use:

```
1  # The time duration of the actual data sample
2  # =
3  # (number of samples) / (sample rate)
4  time_duration = n / r
5  # The maximum plottable time
6  # =
7  # (time per division) * (number of time divisions on graph)
8  time_maximum = t_div * n_t
9  if time_duration > time_maximum:
10     index_maximum = int(time_maximum * r)
11 else:
12     index_maximum = n
13 plottable_range = data[0:index_maximum]
```

Lines 9 - 12 determine the maximum index that can be graphed. This can be explained by considering a dataset $d = \{d_1, d_2, \ldots, d_m\}$. If the data is recorded at a rate $r\ \frac{\text{samples}}{\text{second}}$, then the duration of the read is m/r seconds. However, the graph can show a maximum time duration $t_{max} = t_{div} n_t$ (the time per division times the number of divisions). So, if the time represented by the data is greater than t_{max} (line 9) then we must find the data range within d which can be plotted. The index of this d_i is given by $i = \lfloor t_{max} r \rfloor$. If, however, the time duration of the read is less than t_{max}, then we simply display the available data. This will result in a portion of the graph being empty. An improvement on this would be to read more data until t_{max} is met. Thus the plottable data is $d_{plottable} = \{d_1, d_2, \ldots, d_i\}$. Line 13 is the pythonic way of taking a range of an array.

3. Results

Poscope is a working example of a software oscilloscope. It accurately displays the waveforms provided it. The performance is adequate, however the framerate of the plot is low. This is due to the time required to sample enough data, process that data, and display it. Using the default settings (timer = 10 ms, $n = 1000$, $r = 10000$) the program will restart the plot cycle every 10 milliseconds, read data for another $2n/r = 20$ milliseconds ($2n$ due to double-buffer triggering), process the data for an indeterminate amount of time, and generate a static plot figure, again using an indeterminate amount of time.

Using a conservative estimate of 20 milliseconds for the time required to process the data and generate the graph, each frame takes a minimum of 40 milliseconds to complete. This leads to a framerate of $1000/40 = 25$ frames per second. The plot animation appears relatively smooth at this framerate, however it could stand for improvement.

Both the read times and the processing times can be reduce, however. By reducing n, we can reduce both the read time and the processing time. Further, increasing r will reduce the read time, but will also cause the data to represent a shorter time duration. It will, however, give a higher resolution to the data.

The performance of the trigger could be improved by using a *continuous read* rather than a fixed-size read. That is, instead of using double-buffer triggering wherein we read $2n$ datapoints, we could set the NI USB-6009 to send a stream of data points. Rather than first reading twice the amount of data necessary, the program could look at each data point individually, compare it to the trigger level, and discard it after use. Only when it found a value greater than the trigger would Poscope shut off the continuous stream and request n more data points. There would thus be less data storage and, on average, the NI USB-6009 would not end up reading all $2n$ data points. The NI-DAQmx interface offers such a function however the NI USB-6009 does not support it. This places practical limitations on the device, as certain waveforms or plots may appear choppy.

4. Conclusion

The software presented here is an adequate oscilloscope for simple tasks and limited frequency ranges. Its limitations lie in speed of operation, features, and its ability to read data *only* from the NI USB-6009.

5. Future Work

Poscope can be extended in many areas. The first is functionality. Currently Poscope supports only 1 input (one channel) however the NI USB-6009 has 8 analog inputs and 8 digital inputs. It is therefore possible to have up to 16 channels on the scope. The vast array of options and configurations available on hardware oscilloscopes can also be coded in.

There are many techniques described in Hickman[1] which conventional digital oscilloscopes use to enhance the quality of the samples. Included in these are data smoothing and data averaging, both of which considerably help the discrete samples appear more continuous when graphed and reduce noise. This would allow for a smaller number of samples taken and speed up the data processing and graphing, all of which would speed up the whole program.

The program is written in Python, an easy to use but relatively slow language. It is possible to make Poscope faster by translating it to C++. This is also relatively simple, as much of the functionality (Qt, Qwt, and NI-DAQmx) are themselves written in and intended for C or C++ programs. The Python versions (PyQt4 and PyDAQmx) are one-to-one mappings from the originals to their Python bindings: all the function names are identical or follow some systematic renaming scheme, and all the function arguments are identical.

The method used for plotting the data is also not optimal. It is a simple solution, used because of the short time-frame of the project. Currently Poscope creates a completely new plot for each read. This is quite inefficient. A more suitable method of displaying the graph is to use a specialized graphics library, something like OpenGL (an advanced graphics library capable of intricate 3D animation).

6. Acknowledgments

Poscope is composed of several free, open-source parts. The graphical user interface is written using the Qt GUI framework and the Qt Widgets for Technical Applications (Qwt), both converted for use with the Python language by the PyQt4 project. All of Qt, Qwt, and PyQt4 are supplied by Nokia. The data is processed using NumPy, an excellent numeric library for Python. The NI USB-6009 is controlled by the proprietary (but free to use) NI-DAQmx driver library from National Instruments. The driver is useable from Python via the unofficial PyDAQmx module.

7. References

[1]Hickman, Ian. *Digital Storage Oscilloscopes.* Oxford: Newnes, 1997.

[2]National Instruments Corporation. *NI USB-6008/6009 User Guide and Specifications.* 2008.

[3]The Nobel Foundation. *Nobel Lectures, Physics 1901-1921.* Amsterdam: Elsevier Publishing Company, 1967.