Measurements Laboratory
Geiger Experiment
Software Reference

Mark Orchard-Webb

November 17, 2017

Preface

The LaTeX sources of this document and for the software documented herein can be accessed via Subversion for anonymous read at the following URL.

```
svn://svn.ugrad.physics.mcgill.ca/geiger-339
```

On Linux or Mac you can issue the following command in a terminal

```
svn co svn://svn.ugrad.physics.mcgill.ca/geiger-339
```

You will need to have subversion installed, on Ubuntu you can do this with

```
sudo apt-get install subversion
```

On Microsoft Windows apparently TortoiseSVN is all the rage.

Should installing Subversion not be to your taste, you are welcome to try an experimental HTTP interface at

```
http://svn.ugrad.physics.mcgill.ca/download
```

or you can browse the repository via websvn at

```
http://svn.ugrad.physics.mcgill.ca/public
```

1 Introduction

The software for this experiment is pretty simple. There two parts, the Arduino sketch, `geiger.ino` and the Python program, `geiger.py`.

1.1 `geiger.ino`

The sketch transforms the Arduino in to a device which converts a sequence of positive going pulses into a sequence of numbers. The value of the n_{th} number is the duration in microseconds between the n_{th} and the $n + 1_{\text{th}}$ pulse.

1.2 `geiger.py`

This program connects to the Arduino and records the data sent from it. It will optionally display a histogram of the frequencies of the bins which are generate by the default behavior of the `numpy.histogram()` function.

2 Running the Code

Since `geiger.py` attempts to implement real-time graphics, there is a slight modification required to the configuration of **Spyder**. Click on the menu `Tools`, select `Preferences`, click on `IPython Console` on the left pane, click on the `Graphics` tab which should appear on the right pain. The second *region* in this tab is entitled `Graphics Backend`, and within it you should find a drop-down-list adjacent to the word, `Backend`. This is probably set to `Inline`. Change it to `Automatic`. This should allow plots to open in their own windows where they can be zoomed and updated.

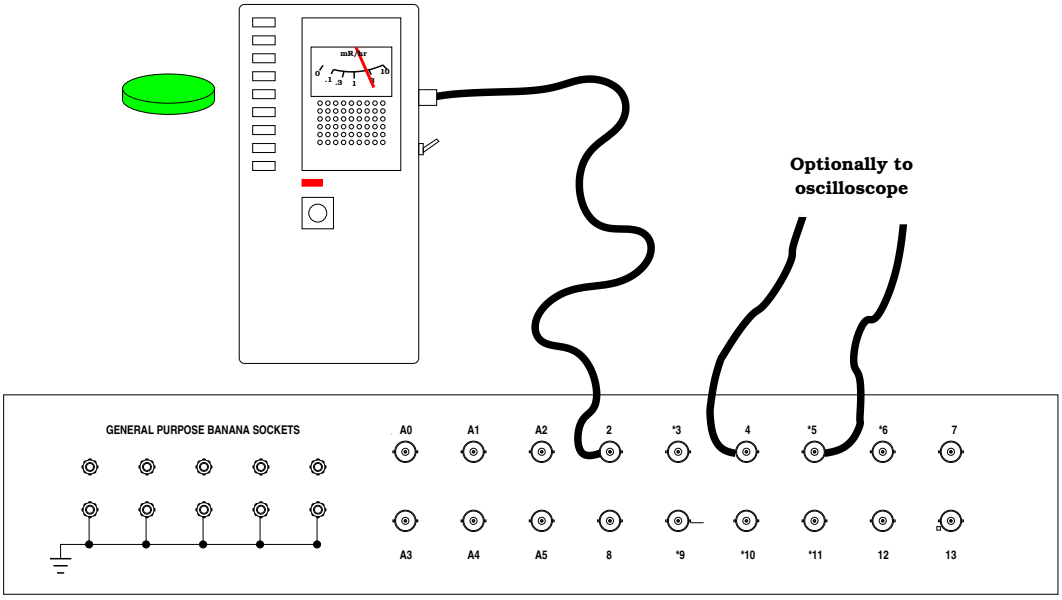

Figure 1: External Connections to Arduino. Geiger counter to digital pin 2, debugging outputs `InISR` from pin 4, `QueueLength` from pin 5.

3 Design of `geiger.ino`

3.1 The Calm Before The Storm

This section is sometime quite dense. If you find it interesting, great. If you are terrified that you are somehow expected to learn this stuff, don't worry about it. Rather than having one set of documents for users and one for developers, I prefer to put it all in one document ... which is why you won't find too many comments cluttering up my code.

3.2 The Storm

The basic idea is that everytime a pulse arrives, we record a timestamp. We can then calculate the interval between this timestamp, `current_micros`, and the timestamp of the previous pulse, `prev_micros`.

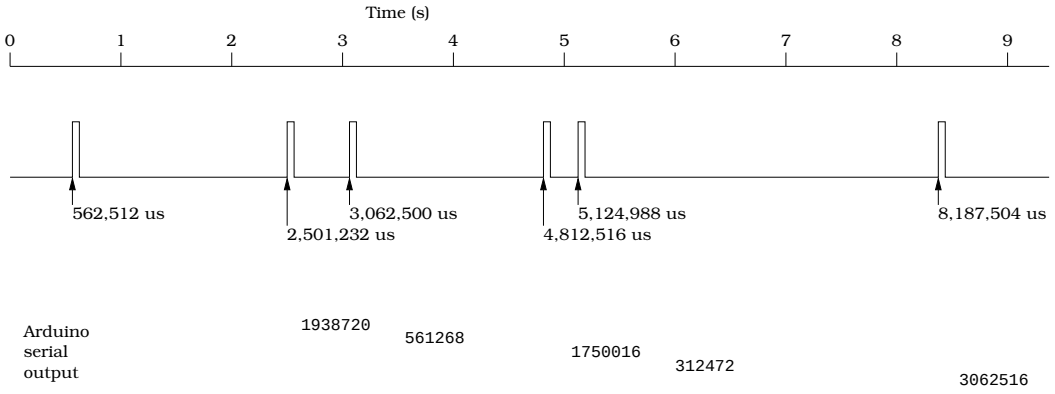

Figure 2: Overview of process. Incoming pulses are timestamped, delta-time between current timestamp and previous timestamp are calculated and sent out over serial interface to computer.

An examination of the output of the Geiger counter reveals that the positive going pulse is high for O(200 μs). This implies the Geiger counter can not produce more that 5,000 pulses per second. The true number is probably lower.

The procesed information needs to come out via the serial port. For simplicity, let us assume we will, rather than using binary formats or other compression, just use `Serial.println()` to output the number in it's ASCII representation — this is grossly inefficient, since we will only use 12 characters[1] from an alphabet of 256 characters — but we can read the data right off the screen, which is not without benefit!

to calculate the theoretical maximum throughput, we need to digress into how serial communication operates. The serial protocol sends individual bits at a rate called the *baudrate*. Obviously we want a high baudrate, but on the downside, the higher the baudrate the more likely errors become. Historically the higher baudrate supported by the Arduino environment

[1]Namely, these characters: 0 1 2 3 4 5 6 7 8 9 CR and LF

was 115,200 *baud*, or bits per second. In serial protocol, a character requires between 9[2] and 12 bits[3]. In the case of the Arduino the format is 8N1, requiring 10 bits, so at 115,200 baud we can output 11.5 kB/s.

A line of output, as produced by `Serial.println()`, will consist of the ASCII representation of the number followed by a carriage-return, CR, and line-feed, LF. Recalling that the shortest interval possible from the Geiger counter is 200 μs, the shortest interval will be 3 digits plus CR/LF overhead characters, for 5 total. Thus, maximum throughput is 2300 events/s.

Here we have an interesting problem, it is possible pulses can come faster than we can describe them. To make matters worse, as you will have noticed, the output of the Geiger counter may frequently involve relatively long durations with no events, followed by a rapid fire events. If the long duration is of order seconds, it will require 800 μs to describe. A näive design will have a nasty bias — the minimum duration measurable will be a function for the duration of the preceeding interval. Short intervals which follow long intervals will be suppressed.

On the other hand, long intervals might be very useful, since during a single long event we could potentially output multiple measurements.

The solution to this problem is a first-in-first-out queue or FIFO structure. In this case a circular queue. With the marvelous operating system installed on the Arduino, we have one process putting measurements on to the tail the queue and a second process taking measurement from the head of the queue and copying them to the serial output.

The process copying to the serial port is the `loop()` function. The measurement process is what is called an interrupt handler. When a pulse arrives, the processor saves a few bytes of context and jumps from *whatever* it was doing and jumps into the `click()` function. When the `click()` function is finished, the processor returns to the point where it was when the interrupt arrived and restores the saved context and resumes execution as if nothing had happened. The `click` function is called an *Interrupt Service Routine*, or ISR.

There are issues which arise when we are sharing variables between the two processes. Even something as simple as an increment operator can be interrupted mid-execution with unfortunate results. Assume `loop()` is incrementing a variable, x. This really means route the bits at the location in memory assigned to the variable to the processor, activate the incrementing microcode, then route the resulting bits back to the location in memory assigned to the variable. If the interrupt occured during this operation, and `click()` modified the same variable, the result `click()` stored, would be overwritten by the result `loop()` calculated. The solution to this problem are the functions, `cli()`, clear-interrupts, and `sei()`, set-interrupts. These functions disable and enable interrupts to avoid the described senario, when working outside the interrupt handler on variables which may be accessed by the interrupt handler. These are called critical sections. What happens now is that when a pulse arrives, if interrupts are disabled, an interrupt pending flag is set in the processor. When the interrupts are enabled after the critical section, the processor then jumps to `click()`. The downside is that if interrupts are disabled when a pulse arrives, it takes longer before the timestamp is recorded — time waits for no interrupt! This obviously decreases the precision, but not by much, if we keep the critical sections short.

The FIFO queue is implemented using an array of 256 unsigned long (32 bits) values, `buffer`, which has two unsigned char (8 bits) indexes, `head` and `tail`. The range of an unsigned char is 0 – 255, which corresponds to the index space of `buffer`. When `head` has the value, 255,

[2]Example, 7N1: 1 start bit, 7 data bits, no parity bit, 1 stop bit.
[3]Example, 8E2: 1 start bit, 8 data bits, parity bit and 2 stop bits

(11111111 in binary) and is incremented, the resulting value, 256 (100000000 in binary), which requires 9 bits, the leading 1 bit falls off, and the value of `head` returns to 0. This is why the queue is called circular. Again, there is a potential problem: if pulses are coming in faster than numbers are going out, `head` may get to be 255 steps ahead of `tail`. This is a problem, because if `head` is incremented again, it will coincide with `tail`, which is identical to the state of an empty queue. If this happens there is inevitably data loss, but worse, we will be unaware there was data loss. To catch such errors, There is a variable, `count`, which keeps track of how many intervals are stored in the queue. If the queue is overfilled, the Arduino prints out the string "`Overrun`" and resets to an empty queue.

It is possible to monitor when the Arduino is running in the interrupt handler code. Pin 4 is brought high upon entry and brought low just before exit. My observations are that the latency before the interrupt handler gains control of the processor is 3.7 μs and the duration of the handler is 11.4 μs.

The value of `count` can be monitored on the oscilloscope. As you will learn in the calibration experiment, there are several outputs on the Arduino which support variable duty. or *Pulse Width Modulatated* pulse streams. Looking at pin 5 on an oscilloscope, the duration of the high section represents occupied space in `buffer`, the low in free space. A time scale of 0.1 ms/div, triggering on rising edge at around 2.5 V works well.

In not terribly rigourous testing, the shortest interval I have been able to measure is 48 μs. This is a quarter of the shortest possible interval coming from the Geiger counter, so the software is not the limiting factor.

4 Design of `geiger.py`

The challenge for the Python code is to capture all the data emitted by the Arduino. A secondary goal is to display the data. There are ways of doing this using multiple threads, but this might not be fantastically appropriate as the first physical measurement in the lab. I have kept the code as simple as possible, with the caveat that using graphics *may* cause you grief recording large data sets. The reason is that the data is coming in at a rate beyond your control, updating the graphics requires making a call to a function which will return at some point in the future, but if the input buffer has overflowed you will have lost data. It is highly probable that the time taken by the graphics scales with the size of the data set, so I recommend disabling graphics if you are collecting data at a high count rate.

The first 70 lines of `geiger.py` are an implementation of a driver for the Arduino. When creating an object for communicating with the Arduino, you have an option of setting the debugging verbosity. If you use

```
arduino = Arduino()
```

or

```
arduino = Arduino(0)
```

or

```
arduino = Arduino(debug=0)
```

There will be no debugging statements printed. To debug the connection, use

```
arduino = Arduino(1)
```

or

```
arduino = Arduino(debug=1)
```

To see every line read from the Arduino, use the value 2. To see both of these at the same time, use 3 — the sum, or more accurately, the *logical or* of the individual values.

Once created, the `arduino` object has two methods:

- `get_interval()` — returns the next interval in the series. If the interval is not yet ready, it will block until the pulse arrives.
- `backlog()` — returns the number of bytes currently waiting in the input buffer.

There is also a function, `plot_histogram()` which accepts a vector of intervals, which will be binned into a histogram using `numpy`'s default `histogram()` algorithm. A rather brute force approach to generating a histogram graph using a line plot is then used — this was an arbitrary decision, based upon my cynical belief that it would be far easier than wasting my time trying to figure out how to do a real time plot using matplotlib.pyplot.hist() — if it has been implemented.

The last 20 lines show a quick demonstration of the above mentioned components. Graphics are used, or not used, based upon the value of the `graphics` variable.

When the programs runs successfully to completion, the variable `intervals` is a vector of intervals between successive pulses in units of seconds.