Geiger Counter Logger

Areg Hovhannisyan

March 7, 2022

Contents

This file documents the Arduino-based data capture system for the Geiger Counter project. Using a common Arduino board, we capture the pulses emitted by the Geiger counter and transfer the timestamps to a PC with the Arduino's serial port for further processing.

1 Requirements

- 1x Arduino board (Nano, Micro, Uno, Pro Mini, etc) with a matching USB cable
- 2x Jumper wires
- 1x Geiger counter
- The DAQ software from `https://github.com/ustajan/GeigerDAQ`

2 Getting Started

2.1 Programming the Arduino

First, we need to upload the pulse detector code to the Arduino. Download and install the Arduino IDE, and open the `GeigerCounter.ino` sketch. Then, connect the Arduino to the computer using a USB cable. A new serial port should appear in the `Tools > Port` submenu in the Arduino IDE - select it and note it down. Next, select your Arduino board type (Nano, Uno, etc) from the `Tools > Board` menu. If your Arduino model does not appear in the menu, use the Boards Manager (first item in the same submenu) and install the matching Arduino core.

You should now be able to upload the sketch to the Arduino board - just press the upload button. If the upload fails, you might need to try playing with some of the options in `Tools > Processor`.

2.2 Wiring

Now that we have uploaded the code to the Arduino, we need to physically connect it to the Geiger counter to detect and log the pulses. Make sure both the Arduino and Geiger counter are powered off for this step. Use the jumper wires to connect any ground pin (marked GND) of your Arduino

board to the GND pin of the Geiger counter, and pin `D0` of the Arduino to the `PS` pin of the Geiger counter[1].

2.3 Logging the data

We are now ready to run the data acquisition software. You need to have Python 3 installed for this step, so download it from https://python.org or your distribution's package manager. Additionally, the `pyserial` library is needed for our logger program to be able to talk to the Arduino. You can install it from PyPI by running `pip install pyserial` in a terminal.
Connect the Arduino to your PC once again and open a terminal. We are going to use `logger.py` to connect to the Arduino and log the signals: run `./logger.py <port>` where `<port>` is the serial port you noted down earlier from within Arduino IDE[2]. Power on the Geiger Counter, and if everything went right, you should see a number get printed every time a detection happens. These numbers are the timestamps[3] of detections.

Once you've verified that the detections and the communication work reliably, you can use `logger.py` to log the detections to a file using the `-o` option (e.g. `./logger.py -o data.txt /dev/ttyUSB0`).

3 More options

`logger.py` has a few more options that may be useful if you wish to further tweak the setup:

3.1 `-p / --data-points`

This option sets the number of data points to be captured before quitting. For example, if you wish to log exactly 50000 detections overnight, you can use the following:

```
./logger.py -p 50000 -o data.txt /dev/ttyUSB
```

[1] While the PS pin is the "safest" source of the signal, one should also try to read out the GM pin or even the INV (inverter pin), as those will produce faster signals.

[2] On Windows systems, `<port>` should look something like `COMn` where `n` is a number. Linux and MacOS systems usually have `/dev/ttyUSBn` and `/dev/tty.usbmodemn`, respectively (`n` is once again a number). For example, if you uploaded the code through `/dev/ttyUSB0`, run `./logger.py /dev/ttyUSB0`.

[3] The timestamp is the number of microseconds since startup, measured by the Arduino's clock.

3.2 `-b / --baudrate`

If you wish to adjust the baudrate used for the Arduino <—> PC communication, you can use this option, but make sure to update the baudrate in the Arduino code too (in the `Serial.begin()` call, in `setup()`). However, do note that using a low baudrate can cause detection misses, especially when hits are coming in at a high frequency.
As an example, if we wish to use the (relatively slow) baudrate of 9600, we can update the Arduino code to

```
// ...
void setup() {
    // ...
    Serial.begin(9600);
    // ...
}
// ...
```

and use run `logger.py` like so:

```
./logger.py -b 9600 /dev/ttyUSB0
```

4 Internal workings

This section describes the technical realization of the pulse detector system.

Conceptually, the system is very simple: the Arduino continuously measures the voltage at one of its pins (which is connected to the GC output) and reports its current system time to the PC whenever it detects a rising edge. However, the implementation has a few noteworthy points.

4.1 The Arduino Code

Let's start with `setup()`, the code that runs only once when the Arduino is first powered up:

```
void setup() {
    pinMode(interruptPin, INPUT);
    Serial.begin(115200);
    attachInterrupt(digitalPinToInterrupt(interruptPin), detect, RISING);
}
```

On the first line, we configure the pin used for detecting the pulses as an input. `interruptPin` is a global constant, defined so:

```
const byte interruptPin = 0;
```

On the common ATMega328p-based Arduinos (Nano, Uno, etc), this makes pin D0 the input pin.

The second line in `setup()` initializes the Arduino's serial interface, which is used for communicating to the PC. We configure it to run at 115200 bauds/second, which allows for relatively high-speed communication.

Finally, we attach an interrupt routine[4] to our digital input pin. This essentially configures the Arduino to call the `detect` function every time the voltage on `interruptPin` (connected to the PS output of the GC) rises above a certain threshold.

Speaking of `detect`, this is what it looks like:

```
void detect() {
    stack[stack_top] = micros();
    ++stack_top;
}
```

Let's walk through it step by step.
First off, as one may already guess from the syntax, `stack` is an array and `stack_top` is an integer. They are both global variables, defined like this:

```
volatile uint32_t stack[256];
volatile uint8_t stack_top = 0;
```

Ignore the `volatile` keywords for now - we will get to them in a bit. What's important here is that `stack` is an array of 256 32-bit integers, and `stack_top` is an 8-bit number, which obviously ranges from 0 to 255.

Every time `detect` is called, we set `stack`'s `stack_top`-th element to the return value of `micros()`, which is essentially the Arduino's internal clock. We then increment `stack_top`.

Effectively, what we have here is a stack (as the variable names suggest) and we push the current time onto it every time a detection happens. `stack_top` is a variable which keeps track of how many elements we have in the stack: it starts at 0 and we increment it every time a detection happens.

In `loop()`, we simply "dump" whatever we have on the stack to the PC:

```
void loop() {
    while (stack_top > 0) {
```

[4] An interrupt routine is a function that gets called by hardware whenever some event happens, and we use one to efficiently detect the pulses from the GC.

```
        Serial.write((uint8_t*) &stack[stack_top - 1], 4);
        --stack_top;
    }
}
```

This function might look a little more complicated, but conceptually it is once again very simple. It basically states the following: "As long as our stack is not empty, pop the top element and send it to the PC over the Serial interface". Since this code runs all the time while the Arduino is powered on, we are basically reporting all our detections to the PC as soon as we get the chance. In practice, our stack will almost never fill up beyond one or maybe two elements, because the detections usually come in a lot slower than `loop()` can empty them. Do note that because we are using a stack and not a queue[5], when two detections happen extremely close together, they will be reported in reverse - the stack is a last-in-first-out data structure after all.

So to reiterate: whenever a detection happens, `detect()` gets called, where we log the current system time to the list of detection timestamps to be reported. Then, when we have "free time" (in `loop()`), we report any unreported hits from that list.

Also, it is important to mention that we send raw byte values to the serial port, instead of printing a human-readable representation:

```
Serial.write((uint8_t*) &stack[stack_top - 1], 4);
```

`Serial.write` is a function that sends `n` raw bytes over the Serial port. In this case, our bytes are the top element in `stack`, and `n` is 4, because our stack holds 32-bit integers, each of which takes up 4 bytes.

This is done to offload the Arduino: generating a human-readable representation is slow and might cause us to miss hits, so we send the raw bytes to the PC and have the python code generate the human-readable representation. This is also the reason why we cannot read the timestamps using the serial monitor in Arduino IDE.

As for the `volatile` keywords in (in the global variable definitions), they tell the compiler that the value of these variable can change outside normal control paths. In our case, this "unexpected" change comes from `detect`, which is an interrupt - the compiler has no way of predicting when `detect` will be called. Because we change `stack` and `stack_top` from inside `detect`, we need to warn the compiler about this, otherwise it might make some optimizations that break our code.

[5] A stack was chosen instead of a queue for practical reasons: queues are usually slower and more complex to implement.

4.2 Logger.py

`logger.py` is much simpler than the Arduino code. It has a few command-line options, but those have already been documented and their implementation is not very interesting.

The most noteworthy lines from `logger.py` are the following:

```
with serial.Serial(args.port, args.baudrate) as ser:
    while True:
        s = ser.read(4)
        micros = int.from_bytes(s, "little")

        outfile.write(str(micros) + "\n")
```

We open a serial port and enter an infinite loop. On each iteration of the loop, we read 4 bytes from the port, and using the `int.from_bytes` function, we "reconstruct" the integer timestamp sent by the Arduino. The second argument of `int.from_bytes` tells it that our four bytes are in little-endian order. What this means is that the least significant byte (a.k.a the lowest one) comes first. Essentially, if the Arduino sends the bytes `12 34 56 78` (hex format), it's really the integer `0x78563412` - this is because the AVR CPU on the Arduino stores integers that way in RAM.

Once we have the integer timestamp sent from the Arduino, we convert it to a string, append a newline, and write it to the output stream (which could be `stdout` or a file).