



International Journal of Engineering Research and Science & Technology

www.ijerst.org

ISSN : 2319-5991

Vol. 14 No. 3 (2018)



ijerst.editor@gmail.com
editor@ijerst.com

Research Paper**STM32-Based Data Acquisition and Python GUI for a Low-Cost Four-LED Continuous-Wave OSL Reader**

A. A. Chaudhary

*Institute of Science, Nagpur, Maharashtra, India***Email:** saiifeabdul1973@gmail.com

Abstract. The optical and electronic design of a low-cost continuous-wave optically stimulated luminescence (CW-OSL) reader built around an STM32F103C8T6 (ARM Cortex-M3) microcontroller. Here we describe the software that turns that hardware into an automated instrument: a minimal serial firmware that times the LED-stimulation relay and streams three digitized analogue channels, and a Python/ Tkinter host application that configures a run, plots the decay curve in real time, and exports the data as a spreadsheet and an image. We give the serial protocol, the resolution and sampling-rate budget set by the 12-bit ADC and the USB-serial link, and the operator workflow, and identify weaknesses – an inconsistency between the acquisition delay implied by the GUI and the firmware, no timestamp resolution better than 1 ms, and no handshake or checksum on the link – together with the changes needed to remove them.

Keywords: OSL reader; STM32; embedded firmware; graphical user interface; Python; serial data acquisition

1 Introduction

An OSL reader must (i) switch the stimulation source on for a defined time, (ii) digitize the detector signal throughout that interval at a known rate, and (iii) present and store the resulting decay curve, $I(t)$, for later analysis. In the reader these three tasks are split between firmware running on an STM32F103C8T6 board and a Python application running on the host PC, connected over a USB virtual serial port. This division keeps the microcontroller code small (it only times the relay and forwards ADC readings) while placing the user interface, plotting, and file handling – all of which change more often during development – in the more easily edited host application. This paper documents both halves of the software, the protocol between them, and the resulting timing and resolution budget, and it is intended to be read together with the firmware and GUI [1].

2 System overview

Figure 1 extends the hardware block diagram with the software layers. The microcontroller firmware runs a single loop: it waits for one-character commands from the host ('r' = run, 's' = stop, 'g' = get-sample), drives GPIO pin PB9 (the relay/LED-enable line) accordingly, and on each 'g' reads three analogue channels with the on-chip 12-bit ADC and prints them as ASCII floating-point numbers. The host application (i) presents a configuration window, (ii) opens the serial port and sends the run command, (iii) polls the firmware for samples in a tight loop, timestamping each reply with the PC clock, (iv) updates a live plot every 100 samples, and (v) on completion or on a Stop click, writes the recorded points to an Excel workbook and a PNG image of the completed curve.[2]

3 Microcontroller firmware**3.1 Analogue front end**

Three analogue pins (PA0–PA2, labelled Signal 1–3 in the GUI) are configured as inputs to the STM32's two 12-bit successive-approximation ADCs. Only one channel is wired to the conditioned Pico ammeter output in the present build the remaining two are provided for future multi-channel use (e.g. a reference photodiode or a second PMT). Each reading is the mean of five consecutive conversions,

$$V = \frac{1}{5} \sum_{k=1}^5 V_k, \quad V_k = \frac{ADC_k}{4096} \times 3.300V \quad (1)$$

which reduces the contribution of uncorrelated ADC/amplifier noise to the reported value by a factor of $\sqrt{5} \approx 2.24$ relative to a single conversion, at the cost of a fivefold increase in the time needed for one reading. The quantization step of a single 12-bit conversion over the 3.3 V reference is

$$LSB = \frac{3.300V}{4096} = 0.806 \text{ mV}, \quad (2)$$

and with the picometers' calibrated transfer of 1 mV per pA this is a current resolution of about 0.81 pA per conversion, or ≈ 0.36 pA on the 5-sample average by Eq. (1) propagation. Table 1 collects the resulting figures.

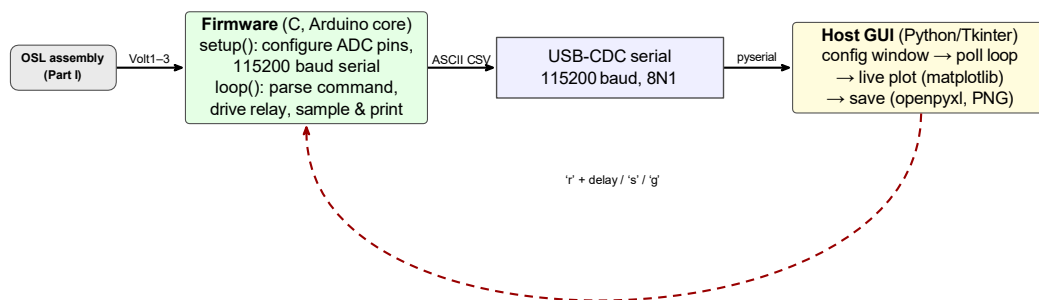


Figure 1. Software layers between the OSL assembly and the operator, extending the hardware block diagram. The dashed control path carries the single-character command protocol of Table 2.

Table 1. ADC resolution budget (STM32F103C8T6, $V_{\text{ref}} = 3.3 \text{ V}$, 12-bit).

Quantity	Single conversion	5-sample average
Voltage LSB	0.806 mV	0.360 mV (propagated)
Equivalent current LSB (1 mV/pA, Part III)	0.81 pA	0.36 pA
Conversion time (12-bit, STM32F1 ADC)	$\approx 1 \mu\text{s}$	$\approx 5 \mu\text{s}$ (5×)

3.2 Serial protocol

Table 2 lists the single-character commands recognized by `loop()`. On 'r', the firmware reads a floating-point delay value (in milliseconds, sent immediately after the 'r'), arms a timer against `millis()`, and two seconds later asserts PB9 (relay ON); it asserts PB9 on 's'. Independent of the run state, each 'g' triggers one 5-sample-averaged conversion on all three channels and a single line of space-separated ASCII values.

3.3 An inconsistency between the GUI and the firmware, and its consequence

The host computes a per-sample delay from the requested sampling rate f (samples per second) as

$$t_{\text{delay}} [\text{ms}] = \frac{10000}{f} \cdot 24, \quad (3)$$

and sends it to the firmware with the 'r' command, expecting it to be applied in milliseconds. The firmware, however, passes the same numeric value directly to `delayMicroseconds()`, i.e. it is applied in *microseconds*. For the GUI default of $f = 16000 \text{ samples s}^{-1}$, Eq. (3) gives $t_{\text{delay}} = 0.026$, so the unit mismatch has little practical effect at that setting, but at lower requested rates (e.g. $f = 4000$, $t_{\text{delay}} = 0.0104$) the firmware inserts a delay about 1000 shorter than the host intends. In practice the achieved sampling rate is set by the USB-serial round trip and the five-fold-averaged conversion (Table 1) rather than by `delayMicroseconds()`, which explains why the instrument still functions; but the *requested* sampling rate entered in the GUI is not the one actually obtained, and the two should be reconciled – either by dropping the 1000 conversion on the host or by changing the firmware call to `delay()` – before the sampling-rate field is relied on quantitatively.

3.4 Timestamping

Sample times are not generated by the firmware; the host stamps each reply with `datetime.now()` at the moment the line is received, formatted to millisecond resolution. Because this timestamp reflects host-side scheduling and USB latency rather than the instant of conversion, successive intervals are not perfectly uniform; this is consistent with the visible jitter in the recorded background traces. A firmware-side counter (e.g. from `millis()`), echoed alongside the three voltages) would remove this source of timing uncertainty and is a straightforward firmware change.

4 Host graphical user interface

The GUI (Fig. 2) is a Tkinter application, using `matplotlib` (via `dravnov`) for plotting, `pyserial` for the link and `openpyxl` for the spreadsheet export. Table 3 lists the configuration fields exposed to the operator.

4.1 Data export

On Stop (or normal completion), the host decimates the buffered lists to the requested sample count, writes a four-column workbook (three signal channels and elapsed time) with `openpyxl`, and calls `plt.savefig` to store a PNG of the finished curve, both under `. /Plot—Data/ProjectInrie / [3]`. A confirmation dialog reports the total number of points recorded before the files are written.

Table 2. Firmware serial protocol (115200 baud, 8N1). All replies are terminated by Serial.print, i.e. no explicit newline is framed by the firmware; the host relies on Serial.readline() timing out or on the reply length.

Command	Sent by	Argument	Action
r	host	float, ms	Sets a start delay; after 2000 ms elapsed (fixed in firmware) drives PB9 HIGH
s	host	—	Drives PB9 LOW three times in succession (stop)
g	host	—	Performs one averaged conversion on PA0–PA2 and prints v1 v2 v3

Algorithm 1 Firmware main loop (condensed from the listing in [3], Appendix A)

```

1: setup: configure PB9 as output; PA0–PA2 as analogue input; open serial at 115200 baud
2: loop
3:   if serial byte available and not running then
4:      $c \leftarrow$  read one byte
5:     if  $c = 'r'$  then
6:        $t_{\text{delay}} \leftarrow \text{parseFloat}() \times 1000$ ; running  $\leftarrow$  true;  $t_0 \leftarrow \text{millis}()$ 
7:     else if  $c = 's'$  then
8:       PB9  $\leftarrow$  LOW (repeated 3  $\times$ ); delay(1000)
9:     end if
10:  end if
11:  if running and  $\text{millis}() - t_0 \geq 2000$  then
12:    running  $\leftarrow$  false; PB9  $\leftarrow$  HIGH ▷ stimulation relay energised
13:  end if
14:   $v_1, v_2, v_3 \leftarrow \text{getvoltage}(\text{PA0}), \text{getvoltage}(\text{PA1}), \text{getvoltage}(\text{PA2})$  ▷ Eq. (1)
15:  print  $v_1, v_2, v_3$  (4 decimal places, space-separated)
16:  delayMicroseconds( $t_{\text{delay}}$ )
17: end loop
18: function GETVOLT(pin)
19:    $s \leftarrow 0$ 
20:   for  $i \leftarrow 1$  to 5 do
21:      $s \leftarrow s + \text{analogRead}(\text{pin})$ 
22:   end for
23:   return ( $s/5$ )  $\times 3.300/4096$ 
24: end function

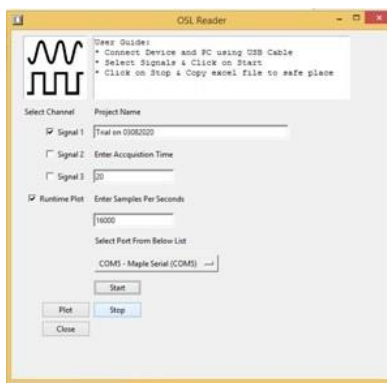
```

5 Discussion: robustness and suggested improvements

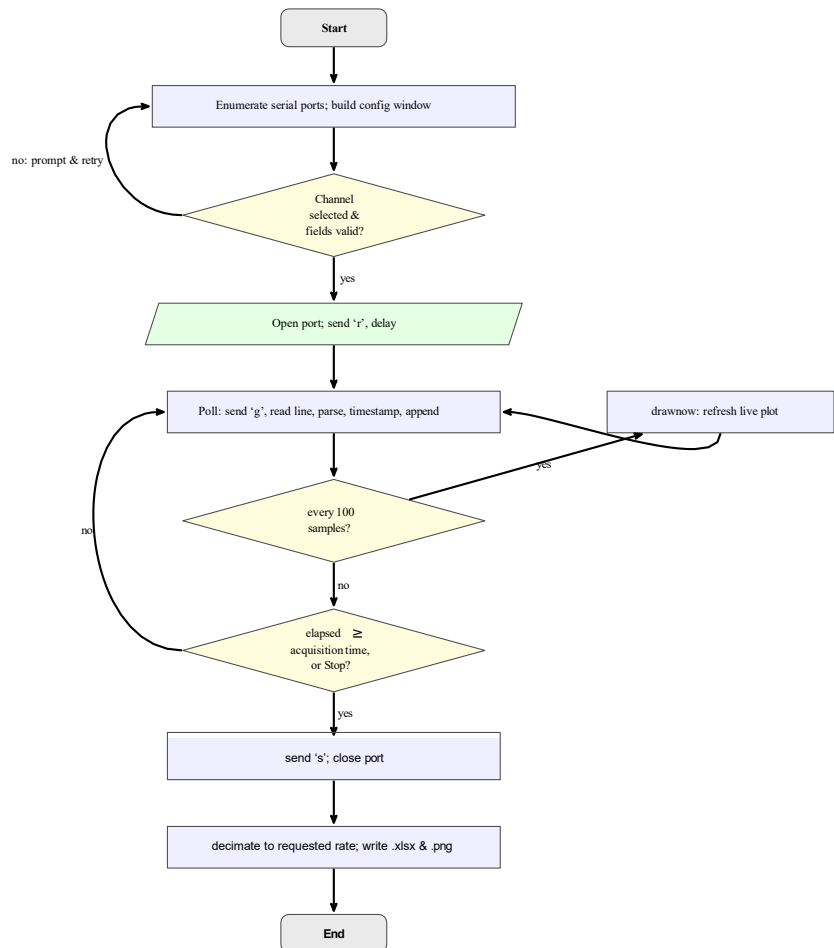
The present software is adequate to reproduce the measurements of Part III but has several weaknesses typical of a first laboratory build, which we list so that they can be addressed in a revision.

- **No framing or checksum on the serial link.** A line is accepted if it splits into exactly three numeric tokens; a corrupted or partial USB packet is silently dropped rather than flagged, which can bias the reported sampling rate on a busy USB bus.
- **Blocking, single-threaded polling.** The acquisition loop and the live-plot redraw run in the same thread as the Tkinter event loop, so the GUI is unresponsive while a run is in progress except through the polled Stop check; a background thread or asyncio-based serial reader would remove this.
- **No firmware acknowledgement of 'r'/'s'.** The host cannot distinguish a genuinely started run from a device that failed to open, beyond the absence of subsequent data.
- **Units mismatch (Sect. 3.3),** which should be fixed before the "samples per second" field is used to set an absolute time base.
- **No persistent configuration.** Every run re-enters the acquisition time and sampling rate; a saved-defaults file would speed up routine use.[4]

None of these affect the validity of the decay curves reported which depend on the ratio of successive samples rather than on the absolute sample timing, but they should be corrected before the instrument is used for quantitative kinetics work (e.g. Eqs. 1 and the LM-OSL expressions) where the time axis itself carries information.



(a)



(b)

Figure 2. Graphical user interface. (a) Configuration window, reproduced from the running application (channel selection, project name, acquisition time, sampling rate, COM port and a live-plot check box). (b) Operator/software flow redrawn from the programming flow chart of [3].

Table 3. Configuration fields presented in the GUI window.

Field	Description
Signal 1/2/3 check boxes	Select which of the three ADC channels are recorded and plotted (only Signal 1 is wired to the picoammeter output in the present build)
Project Name	File name stem used for the saved .xlsx and .png outputs
Acquisition Time	Run duration in seconds (default 100 s in the traces of Part III)
Samples Per Second	Requested sampling rate, 0–16000 (see Eq. (3) and the caveat above)
Select Port	COM port drop-down, populated from serial.tools.list ports
Runtime Plot	If ticked, the decay curve is redrawn every 100 samples during acquisition
	Start / Stop / Plot / Close
	Begin acquisition; stop early; redraw/finalize the plot; exit

6 Conclusions

The reader is operated by a two-layer software stack: a minimal, single-character-command STM32 firmware that times the stimulation relay and streams averaged ADC samples, and a Python/Tkinter host that configures, plots and exports a run. We have set out the protocol, the resulting resolution budget (about 0.8 mV, or 0.8 pA, per 12-bit conversion, improved by 5 through on-board averaging), and the operator workflow, and identified a delay-unit mismatch between host and firmware, host-side timestamping jitter, and the absence of link-level error checking as the priority items for a future revision.

References

- [1] T. Lapp, M. Jain, C. Ankjærgaard and L. Pirtzel, “Development of pulsed stimulation and Photon Timer attachments to the Riso TL/OSL reader,” *Radiation Measurements*, vol. 44, nos. 5–6, pp. 571–575, 2009, doi: 10.1016/j.radmeas.2009.01.012.
- [2] J. D. Scudder, J. A. D. M. et al., “ACQ4: An open-source software platform for data acquisition and analysis in neurophysiology research,” *Frontiers in Neuroinformatics*, vol. 8, 2014, Article 3. doi: 10.3389/fninf.2014.00003
- [3] E. Gazoni and C. Clark, *openpyxl: A Python library to read/write Excel 2010 xlsx/xlsm/xltx/xltm files*, open-source software/documentation, 2014.
- [4] pySerial Team, *pySerial-asyncio: Async I/O Extension for pySerial*, 2016.