Python interfaces and tools for the Lynn lab electronic equipment

Kye W. Shi
Nick Koskelo
Lorenzo Calvano

Lynn Lab

Summer 2018

Abstract

There is a piece of software that is allegedly quite popular among scientists, called *LabVIEW*. Scientists use LabVIEW to do measurements with their sensors or something. However, there are some big problems with LabVIEW.

- The LabVIEW programming "language" (being graphical and all) is quite unlike mainstream languages such as Python, Java, C/C++, and so on. As such, LabVIEW is harder to work with. A typical Harvey Mudd student embarking on their career in the Lynn Lab almost certainly has worked with Python prior (having endured Core) but quite probably has not heard of LabVIEW.
- LabVIEW is a proprietary piece of software that costs a crap ton of money. As such, it is practically impossible for students to obtain LabVIEW on their own personal computers and consequently much harder for students, unfamiliar with LabVIEW, to attempt to learn LabVIEW on their own time, in their own rooms, on their own computers.
- LabVIEW has not done a spectacular job at keeping their versions cross-compatible (e.g. software designed in LabVIEW 2011 fails to start in LabVIEW 2009). This version incompatibility prevents us from collecting data and controlling motors from the same computer (the two lab computers run different LabVIEW versions) and consequently prevents us from fully automating the data collection process (which would require motor movements interlaced with coincidence sampling). Furthermore, proprietary-ness and software license red tape make this issue harder to fix than simply upgrading the LabVIEW version to match.

These barriers to working with LabVIEW render it a cumbersome and rather inconvenient tool. It is especially intimidating to newcomers to the lab, like me. As such, we've reimplemented some of the LabVIEW measurement tools in Python, in an effort to make the software easier to understand, alter, and maintain, and (on a larger scale) to facilitate in ultimately automating the data collection process.

Details on the usage and functioning of these tools are documented in this write-up, so the reader (i.e. *you*) won't have to spend hours scouring the internets for manuals and whatnot, as we did. Enjoy.

Contents

Prelude and some notes

Prior to our arrival in the Lynn lab, measurements were done using two computers.

A newer computer (named *lynnlab*, maybe), running Windows 7, was used to measure coincidences from an older time-to-amplitude converter (TAC) setup. It supposedly has better time resolution, but it is more complicated to configure and *much* bulkier. As such, we don't use it. The *lynnlab* computer is also used to control motorized optical table mounts (e.g. rotating plate holders and whatnot) from Thorlabs and Newport.

An older computer (named *MEHTA*), running Windows XP, was used to measure coincidences using the FPGA CCU. It contained (and ran) the LabVIEW VIs that interfaced with the CCU. It also lacked an internet connection, so we plugged in a cheap USB WiFi module from AliExpress as a temporary fix[1].

In order to integrate automatic motor movement and data collection, we would have to move both motor and coincidence-counter interfaces to one computer.

1 Coincidence-counting unit

Photon coincidence counting is done using an *Altera DE2*[2] field-programmable gate array (FPGA), designed by some folks at Whitman college or something. They have a website with some information on the FPGA coincidence counter here: `http://people.whitman.edu/~beckmk/QM/circuit/circuit.html`.

The CCU designers provide some reference LabVIEW "instruments" (*VI*s) to interface with their CCU. In addition, there was a VI already created on the Lynn lab computers for interfacing with the CCU before our arrival in the lab (`RCH_HMC_coincidence_time_rs232.vi`). We crawled through these sources code and gathered what we could from the attached user manual (written by the Whitman folks) to find what we could on connecting to and communicating with the CCU.

The CCU counts photon incidences over 8 channels, denoted `C0` through `C7`. Channels `C0` through `C3` correspond to single-incidence counts (i.e. photon incidences on a single detector), while channels `C4` through `C7` corresponds to coincidence counts (i.e. two photons striking two detectors "simultaneously"[3]). Single-incidence channels `C0`, `C1`, `C2`, and `C3` are labeled, perhaps for readability's sake, as `A`, `B`, `A'`, and `B'`, respectively. Coincidence channels are configured, using little toggle switches on the FPGA board, to track and count coincidences among various pairs of single-incidence channels, to different time resolutions. More details on these respective configurations and resolutions can be found in the Whitman CCU manual.

1.1 Serial connection parameters

The CCU communicates with the computer over the RS-232 serial protocol. Details on the protocol are not particularly important. Basically, under this protocol, the computer connects to the device over a particular *serial port* (loosely corresponds to the computer's hardware port attached to the device). Then, the data is transferred, one bit at a time, at a particular *baud rate* (i.e. the number of bits sent per second).

The serial port varies depending on what computer the device is connected to and what hardware slot the cable is plugged in to. In our case, we found the port to be `COM1` when the CCU was plugged into the native RS-232 port and `COM4` when the CCU was plugged into the USB port via an RS-232–to–USB converter cable. The *Windows Device Manager* should provide some useful clues regarding which serial port to use.

[1] By the end of the summer, I will have taken the WiFi stick back home with me, and so *MEHTA* will be once again without internet.

[2] The website also has a section on a CCU implemented with the *DE2-115* board. Ours is *not* that, I think.

[3] within a narrow time interval of each other. This time interval (termed the *time resolution*) is, for our CCU, about 7 ns, I think.

The baud rate is specific to the design and implementation of the serial device (i.e. it is chosen by the device, or the designers of the device). In our case, we found from the provided VI source (and confirmed by crude experimentation) the baud rate to be `19200` bits per second.

1.2 Serial data protocol and format

The fundamental "unit" of data is a *byte*, or a contiguous chunk of 8 bits. Each byte contains the binary representation of some value between 0 (`0b00000000`[4]) and $2^8 - 1 = 255$ (`0b11111111`). As such, data sent from any serial device is generally grouped into and interpreted as sequences of bytes[5].

Our CCU transmits photon-counting data in *packets* of 41 bytes. Each packet of 41 bytes is "partitioned" into 8 chunks—one for each counter channel—of 5 bytes each, with the last byte reserved for a termination byte (`0xFF`[6], or `0b11111111`) marking the end of the data packet.

Each of the 8 5-byte chunks encode some sort of a multi-byte unsigned integer in little-endian form (i.e. least-significant byte first). To avoid possible clashes with the termination byte, only the first 7 bits of each byte are used, and the 8th bit in a data byte is always kept a `0`[7]. Thus, in total, there are up to 35 bits for storing each counter value[8]. For example, the counter value 4321, with binary representation `0b1000011100001`, would be sent, in order, as the byte sequence

`01100001 00100001 00000000 00000000 00000000`.

The first byte contains the first (i.e. least significant) 7 bits, padded on the left by a prefix `0` bit. The next byte contains the next 7 bits, and so on. Thus the byte-wise encoding of each counter value forms a base-128 representation[9], such that, if the 5 bytes were denoted b_0, b_1, b_2, b_3, b_4, the counter value would be found as

$$b_0 \cdot 1 + b_1 \cdot 128 + b_2 \cdot 128^2 + b_3 \cdot 128^3 + b_4 \cdot 128^4.$$

For example, a data packet containing the counter values

`C0`	2718
`C1`	281828
`C2`	4
`C3`	59045
`C4`	235
`C5`	360
`C6`	2874
`C7`	71352

would be transmitted as the following sequence of bytes (with a header row added for clarity—columns containing "delimiter"/prefix `0` bits are labeled `0`, and relevant data bit columns are labeled `X`):

[4] The `0b` notation denotes that a particular sequence of digits encodes the *binary* (i.e. base-2) representation of an integer. For example, binary `0b1101` corresponds to the integer $1101_2 = 2^3 + 2^2 + 2 = 13$.

[5] Data is, on a *physical* level, sent one *bit* at a time, but it is *read*, or *interpreted*, one *byte* at a time.

[6] The `0x` prefix denotes a *hexadecimal* (i.e. base-16) number representation, with letters `A` through `F` denoting digits for 10 through 15. For example, hexadecimal `0xC0FFEE` corresponds to $\texttt{C0FFEE}_{16} = 12{\cdot}16^5{+}0{\cdot}16^4{+}15{\cdot}16^3{+}15{\cdot}16^2{+}14{\cdot}16{+}14 = 12648430$.

[7] This observation has also been confirmed by crude experimentation: in a relatively large sample of counter values over a large-ish (i.e. effectively random) range of numbers, the 8th bit consistently remained `0`.

[8] The Whitman manual claims each counter to be encoded as a 32-bit number; that sounds like the correct order of magnitude.

[9] This claim is backed by evidence in the Whitman LabVIEW code, where byte sequences indeed appear to be decoded in base-128.

0XXXXXXX	0XXXXXXX	0XXXXXXX	0XXXXXXX	0XXXXXXX	
00011110	00010101	00000000	00000000	00000000	
01100100	00011001	00010001	00000000	00000000	
00000100	00000000	00000000	00000000	00000000	
00100101	01001101	00000011	00000000	00000000	
01101011	00000001	00000000	00000000	00000000	
01101000	00000010	00000000	00000000	00000000	
00111010	00010110	00000000	00000000	00000000	
00111000	00101101	00000100	00000000	00000000	11111111

1.3 Python tool set

1.3.1 CCU controller library: `fpga_ccu`

1.3.2 Command-line monitoring utility: `ccu.py`

2 Motorized plate mounts