# INSTRUCTIONS

Here are some basic “instructions” on what each script does, as well as a little bit of how to use them.

## A quick primer on the terminal

In case you’re unfamiliar with how terminals work, here is some very basic information on how to navigate yourself from the command-line.

The Windows default terminal (`cmd`) can be accessed from a launcher shortcut or by searching `cmd` in the Windows start menu. It’s not very good, but here’s the most basic information on how to navigate it:

- `cd folder` to “change directory” to a folder. (`cd ..` to navigate to the parent folder.)
- `dir` to list the files and folders.
- `python script.py` to run a Python script.

There is also another terminal installed, called Git Bash. It is, one might argue, much better than `cmd`. For the most part, if you are unfamiliar with terminals, the differences are not significant. If you are a regular terminal user, Git Bash will make you much happier. If you would like to use Git Bash but have not used Bash very much before, here’s the same basic information (it is very similar):

- `cd folder` to move into a folder.
- `ls` to List files.
- `python script.py` to run a Python script.

## The `thorlabs_apt` package

This is a Python package which we did not write, but which we use, that is used to connect to the Thorlabs motors and control them. Here’s how we use it:

- Open a Python interactive terminal (or, in a Python script…). Import the package with `import thorlabs_apt`. Upon importing the package, all motor connections are scanned and indexed.
- List available motor connections with `thorlabs_apt.list_available_devices()`. This function returns a list of `(hw_type, serial_number)` pairs. The hardware-type (`hw_type`) is not really important; all our motor controllers have type `31` (which, if you really care, corresponds to the TDC001 controller type). The eight-digit serial number (`serial_number`) identifies each individual motor. The serial number for each motor can be found on the back of the “T-Cube” motor controllers connected to the motor, labeled `S/N--------`.

- Open a motor connection by creating a `Motor` object, passing in the serial number: `motor = Motor(83811904)` (where `83811904`, for example, is the serial number).
  - Check the motor's position (in degrees) with `motor.position`.
  - Check whether the motor is moving with `motor.is_in_motion`.
  - Send a command to move the motor *to* a given position (in degrees) with `motor.move_to(position)`. An optional argument, `block`, can be passed to specify whether to wait for the motor to reach the target position and stop before executing the next command. For example, `motor.move_to(15, True)` moves the motor to 15 degrees and waits for it to reach 15 degrees before executing the next command. The default, if omitted, is `block=False`.
  - Move the motor *by* some angle using `motor.move_by(position)`. The `motor.move_by` method also takes an optional `block` argument.

## The `fpga_ccu` Python package

This is a Python package located in `fpga_ccu/`. The package provides some classes and functions for interfacing with the CCU and for displaying the CCU data to the screen. Located in the `fpga_ccu/fpga_ccu/` folder are the following Python modules:

**config.py** This module contains some fixed configurations for connecting to the CCU (i.e. *default* serial port and baud rates) and for the user interface. In general, this should not be modified. The default port/baud are fixed and correct for our CCU, and other things are irrelevant.

**controller.py** This module contains all the logic for interfacing with the Altera DE2 FPGA CCU. It provides a class, `FpgaCcuController`, that opens up the CCU serial port, reads serial data, and parses it to return data packets. More detailed documentation on the protocol and how to use the class in a script can be found in the source code.

This is probably the only class you might care about if you wanted to directly interface with the controller yourself in Python. For the most part, however, this is unnecessary and not preferred; we use `ccu_log.py` (see below) to write CCU data to a file, and all other scripts simply read data entries from that file. This way, we enable multiple programs to simultaneously access the data, with the actual CCU-facing script serving merely as a "backend" of sorts.

**renderer.py** This module contains some logic for actually displaying data to the screen (e.g. printing data entries to the terminal, or displaying data in line and bar plots via matplotlib). You *probably* don't need to care about this module.

**manager.py** This module contains some logic for connecting the CCU controller (`controller.py`) to a screen-side renderer (see the section on `renderer.py`). You probably don't ever have to care about this module.

## The `fpga-ccu` script

This is an executable script bundled with the `fpga_ccu` Python package. The package is installed with `pip install -e setup.py` from the `fpga_ccu/` folder, so that the `fpga-ccu` script is installed to the computer's `$PATH` and can be run directly from anywhere in the terminal.

You don't really have to know how to use all these command-line options; we've configured some Python scripts (in particular, `ccu_log.py`, `ccu_record.py`, and `ccu_monitor.py`) that effectively cover all the use-cases of the `fpga-ccu` script.

To run this script, simply run `fpga-ccu` from any terminal window. It takes a variety of command-line options for customizing the serial connection or the display. Documentation on those options can be found by running `fpga-ccu -h`.

## The `ccu_log.py` script

This script connects itself to the FPGA CCU and logs received data continuously to the screen, as well as to a log file at "tmp/ccu-log.csv". By logging data to a log file, we make it possible for multiple different scripts/programs to access the same data at the same time (since only one program can be connected to the CCU at a time). This script is, for now, used in combination with various scripted measurement routines (e.g. full tomography scan, single-motor calibration scan/sweep).

We are (as of summer 2018) in the process of replacing some of the Python backend with a better, more modularized set of tools. In particular, we have a web-based tool (see the web interface section) that conveniently combines data logging, plotting, and motor movement into a single interface. For compatibility with other scripts that use the log file at "tmp/ccu-log.csv", the webserver also writes its log files to the same location. Consequently, the webserver should not be run concurrently with "ccu_log.py", or else both programs will attempt to seize the CCU serial connection, and one of the two will fail. For now, the webserver doesn't quite do everything we want; in particular, the web server grabs all motor connections upon startup, preventing any other programs from accessing the motors as long as the web server is running. Consequently, none of our automated routines, which require control over the motors, work while the web server is running. For now, we have not yet implemented a fix, and so we still require "ccu_log.py" for most of our routines.

To run this script, simply run `python ccu_log.py` from the terminal, or double-click the script from a file browser.

## The `ccu_record.py` script

This script records and saves some number of data entries to a file. You can select the number of entries to save, or save entries indefinitely. This script amounts to a "save data" or "collect data" kind of tool, which listens for data over an amount of time, saves them to a file, and summarizes them (mean and SEM). The output file is in CSV (comma-separated values) format.

This script collects data from the `tmp/ccu-log.csv` log file, so in order for `ccu_record.py` to work, *either* `ccu_log.py` *or* the web server must be running.

To run this script, simply run `python ccu_record.py`, or double-click the script from a file browser.

## The `ccu_monitor.py` script

This script records data and displays it in some bar and line plots via matplotlib. Counts on channels 0 and 1 (respectively, A and B singles channels) over time are displayed in a line plot on the top left quadrant. Counts on channel 4 (AB coincidences) over time are displayed in a line plot on the top right quadrant. The two bar plots on the bottom display "instantaneous" one-second counts across all eight channels.

The web interface also provides plotting. In fact, the web interface's plotting features are so much more flexible and customizable that there really isn't much reason to still use `ccu_monitor.py` if you are able to use the web interface instead. Nevertheless, `ccu_monitor.py` is still somewhat useful for monitoring data in situations when we cannot keep the server running.

Like `ccu_record.py`, the `ccu_monitor.py` script also runs off the `tmp/ccu-log.csv` log, and so *either* `ccu_log.py` or the web server must be running for `ccu_monitor.py` to work.

## The web interface

The web interface (written in Go, a language much better suited for web applications than Python) combines all of data collection, monitoring, logging, and plotting into a single tool.

The web server does three things: - It connects to the CCU and reads/parses data from the CCU. This data then gets saved to two locations: - A timestamped log file at `web/data/`. This way, all data that ever gets logged while the server is running gets saved, regardless of whether we choose to save it at the time of recording (this would be helpful, maybe, if at some point we realize that we had produced some useful data without remembering to actually record it; we could go back into the log file and grab any data that we collected, provided we know, roughly, the time when we collected the data of interest or have some other means of identifying which entries are actually of interest). - The temporary log file at `tmp/ccu-log.csv`. This was done to provide reverse-compatibility with other scripts that relied on this log file. However, now that the webserver seizes control of the motors (see below) and prevents most of our other automation scripts from working, reverse-compatibility is kind of out the window… for now. We have plans to fix things and make them play nicer with each other, but they are not yet implemented. - It connects to the Thorlabs motors and monitors the motor status by checking the motor's `position` and `is_in_motion` properties periodically. It receives commands from the client (see below) and relays these commands to the motors, allowing the motors to be controlled from the client's web interface. - It runs a web server, allowing a browser ("client") to access the interface and actually manage the controls. The client receives CCU data entries as they are collected and displays the data in various tables, plots, and bars. The client also offers some (basic) motor controls, allowing the motors to be moved to any position. Live monitoring of motor position and in-motion status is also provided on the client interface.

To run the web interface, first launch the web server by running the executable located at `web/server/server.exe` (there is also a shortcut to it at `webserver.lnk`). Then, to access the interface, open up a browser and connect to http://localhost:5000.