> **The Particle Web IDE, Particle Dev, and particle-cli are not the only ways to program Particle devices - by Nathan Robinson**

What is po-util?

Po-util, short for Particle Offline Utility, is a tool I maintain for facilitating the ultimate local Particle development experience. Po-util takes the hassle out of developing Particle projects locally by eliminating the need to use the Particle cloud to build your firmware.

Po-util is my personal solution to local Particle development that uses the GCC for ARM Cortex processors toolchain to build your projects using the Particle Firmware Source directly on your computer.

Po-util is a bash shell script that is available on GitHub for Linux and macOS.

Features of po-util

There are many more features to po-util than aliasing a few `make` commands. Po-util is unique workflow I meticulously developed so that others can experience local Particle development the way it was meant to be.

Po-util offers many features, including:

- Full installation of required dependencies.
- Support for building firmware for the Particle Photon, Electron, P1, Core, Raspberry Pi, and Redbear Duo.
- "Instant" firmware uploading over USB using DFU Utilities.
- Sequential Over The Air firmware uploading to multiple devices in a "product".
- A unified project structure that is cross-compatible between the Linux and macOS editions of po-util.
- An efficient library manager that works with libraries from Particle Libraries 2.0 or GitHub.
- A range of keyboard shortcuts to expedite development within Atom.

Getting started with po-util

Installing and using po-util is a rewarding experience. (You must be an administrator on your computer however.)

For Linux Distributions based on Debian, Red Hat, or Arch, you can run the following in your Terminal to install po-util:

```
$ bash <( curl -sL https://git.io/vX4cl ) install
```

On macOS, you can install po-util using Homebrew:

```
$ brew tap nrobinson2000/po
$ brew install po
$ po install
```

Creating a project with po-util

To use po-util, you must work in an initialized project directory. (More on project structure below.)

To create a project simply run the following in your Terminal:

```
$ po init photon newProject
```

This command creates the `newProject` folder and creates the appropriate folders and files inside. The Atom shortcuts file is set to build firmware for the Particle Photon. (You can substitute `photon` with `electron`, `P1`, `core`, `pi`, or `duo` to initialize the project for other devices.)

To make use of the shortcuts you will need the Atom Build package. You can easily the package and a couple other recommended ones with:

```
$ po setup-atom
```

Po-util project structure

Po-util operates by keeping your code separated into projects that follow an organized structure:

```
firmware/
└ main.cpp
```

```
bin/
└ firmware.bin

devices.txt
libs.txt
.atom-build.yml
README.md
```

All source code goes in `firmware/`, and the compiled binary is saved as `bin/firmware.bin`. You can specify devices to flash sequentially to in `devices.txt`, and `libs.txt` is how po-util keeps track of what libraries to use in the project.

Po-util supports build shortcuts for Atom, and these are set in `.atom-build.yml` should you need to modify it on a per-project basis.

Using po-util with Atom

Now that you have created a project and installed the recommended packages, open the project folder in Atom.

On the left, you should find your `firmware/main.cpp` file. This is where you should keep most of your code. (You can use separate files and libraries however, but the majority of your code should be kept here.)

When your project is initialized, you will find the following dummy code in your `firmware/main.cpp`:

```
#include "Particle.h"

void setup() { // Put setup code here to run once

}
```

```
void loop() { // Put code here to loop forever

}
```

Here is where you can observe one of the most useful features of po-util, the keyboard shortcuts for Atom. If you press `Ctrl + Alt + 1`, Atom will build the project by running:

```
$ po photon build
```

Pressing `Ctrl + Alt + 2` builds the firmware and flashes it to your device over USB using DFU Utilities by running the following, which can also be done from your Terminal:

```
$ po photon flash
```

> **If you prototype rapidly like I do, flashing will become a frequently used shortcut.**

Pressing `Ctrl + Alt + 3` cleans the firmware and removes the `bin/` directory by running:

```
$ po photon clean
```

Pressing `Ctrl + Alt + 4` uploads firmware to your device without rebuilding by running:

```
$ po photon dfu
```

Pressing `Ctrl + Alt + 5` uploads your firmware Over The Air to any devices listed in your `devices.txt` by running:

```
$ po photon ota --multi
```

Configuring your devices to work with po-util

All devices that po-util will use DFU Utilities to upload firmware to must have their firmware patched/upgraded in order to allow po-util to automatically put them into DFU mode. To do so, manually put your device into DFU mode and run the following to update the system firmware on your device:

```
$ po photon update
```

> By default, all Particle devices will go into DFU mode when the computer they are connected to sets the serial baud rate to `14400`. However, on Linux, baud rates can not be set to an arbitrary value like `14400`. Po-util solves this caveat by using `19200` as the baud rate to put the device into DFU mode. If you wish, you can use `14400` on macOS by running `$ po config` and choosing `default` when prompted for a baud rate.

Using Particle libraries with po-util

One of the greatest features of po-util is its flexible library manager. The library manager not only allows using libraries from Particle's list of libraries, but it also supports using any library that is available from GitHub.

A library can be downloaded and added to a project by using the `$ po lib` `get` and `$ po lib add` commands. For example, you could add the `neopixel` library to a project with the following commands:

```
$ po lib get neopixel
$ po lib add neopixel
```

You could go even further and load an example from the `neopixel` library.

First, check what examples are available with:

```
$ po lib ex ls neopixel
```

The command above returns the following, indicating there are examples available:

```
Found the following neopixel examples:

a-rainbow, extra-examples, rgbw-strandtest
```

To load the `extra-examples` example into your project run:

```
$ po lib ex load neopixel extra-examples
```

Sharing a po-util project

The most practical way to share a po-util project is to upload it to GitHub, as your project is initialized as a git repository and set up to use Travis CI for testing.

Po-util provides another method if you want a "quick and dirty" way to share your code so that it can be built without using po-util. Using `$ po lib pack`, you can create a copy of your `firmware/` directory with all symbolic links and libraries packaged inside. A tar.gz archive is created as well.

When using publicly accessible libraries it is advised that you clean the binaries and libraries from your project. This removes the symbolic links from your `firmware` directory, but leaves the libraries your project depends on in your `libs.txt`.

To clean your project run the following:

```
$ po lib clean
```

At any time, you can re-download and add the libraries using:

```
$ po lib setup
```

> **If you are sharing using GitHub and Travis CI, cleaning is not necessary, as Travis CI will do it for you.**

More po-util information

For more information about po-util, read the man page with `$ man po`, check out the website, or consult the repository on GitHub.

Stars, Feedback, Contributions and Donations are greatly appreciated. I hope you enjoy using po-util as much as I have enjoyed creating it.

Linux Edition Repo - macOS Edition Repo

paypal donate bitcoin donate