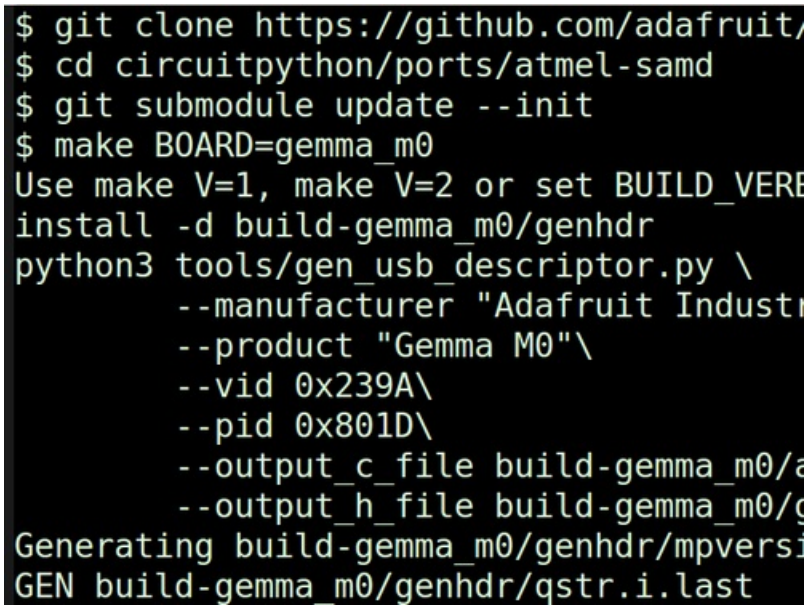
adafruit learning system

Building CircuitPython

Created by Dan Halbert



Last updated on 2021-06-10 05:04:48 PM EDT

Guide Contents

Guide Contents 2
Introduction 3
Linux Setup 4
- Install a Real or Virtual Linux Machine 4
  - Native Linux 4
  - Linux on a Virtual Machine 4
  - Raspberry Pi 4
- Install Build Tools on Ubuntu 4

MacOS Setup 6
Windows Subsystem for Linux Setup 7
- Install WSL 7
- Build CircuitPython 7
- Moving Files to Windows 7
- Mounting a CircuitPython Board in WSL 7

Manual Setup 9
Build CircuitPython 10
- Fetch the Code to Build 10
- Install Required Python Packages 10
- Install pre-commit 11
- Build mpy-cross 11
- Build CircuitPython 11
- Run Your Build! 12
- Use All Your CPUs When Building 12
- When to make clean 13
- Updating Your Repo 13

Adding Frozen Modules 14
Choosing a Different SPI Flash Chip 16
Customizing USB Devices 17
- Customizing USB Devices 17
- Customizing USB HID Devices 17
- Customizing USB Devices Before 6.2.0-beta.3 18
- Customizing HID Devices Before 6.2.0-beta.3 18

ESP32-S2 Build 20
How to Add a New Board to CircuitPython 21

Introduction

Adafruit's CircuitPython (https://adafru.it/AIP) is an open-source implementation of Python for microcontrollers. It's derived from (also known as, a "fork" of) MicroPython (https://adafru.it/f9W), a ground-breaking implementation of Python for microcontrollers and constrained environments.

CircuitPython ships on many Adafruit products. We regularly create new releases and make it easy to update your installation with new builds.

However, you might want to build your own version of CircuitPython. You might want to keep up with development versions between releases, adapt it to your own hardware, add or subtract features, or add "frozen" modules to save RAM space. This guide explains how to build CircuitPython yourself.

CircuitPython is meant to be built in a POSIX-style build environment. We'll talk about building it on Linux-style systems or on MacOS. It's possible, but tricky, to build in other environments such as CygWin or MinGW: we may cover how to use these in the future.

Linux Setup

Install a Real or Virtual Linux Machine

If you don't already have a Linux machine, you can set one up in several different ways. You can install a Linux distribution natively, either on its own machine or as a dual-boot system. You can install Linux on a virtual machine on, say, a Windows host machine. You can also use Windows Subsystem for Linux (https://adafru.it/C2z) (WSL), available on Microsoft Windows 10, which allows you to run a Linux distribution with an emulation layer substituting for the Linux kernel.

We recommend using the Ubuntu (https://adafru.it/C2A) distribution of Linux or one of its variants (Kubuntu, Mint, etc.). The instructions here assume you are using Ubuntu. The 20.04 LTS (Long Term Support) version is stable and reliable.

Native Linux

You can install Ubuntu on a bare machine easily. Follow the directions (https://adafru.it/ObC) (this link is for 20.04) on the Ubuntu website. You can also install Ubuntu on a disk shared with your Windows installation, or on a separate disk, and make a dual-boot installation.

Linux on a Virtual Machine

Linux can also be installed easily on a virtual machine. First you install the virtual machine software, and then create a new virtual machine, usually giving the VM software a .iso file of Ubuntu or another distribution. On Windows, VM Workstation Player (https://adafru.it/C2C) and VirtualBox (https://adafru.it/eiS) are both free and easily installed.

Raspberry Pi

It may be possible to build on Raspberry Pi OS, but it will be easier if you install Ubuntu on your Raspberry Pi. You will also need to download the aarch64 gcc toolchain. The RPi is not a fast machine: be prepared to wait for a build to complete.

Install Build Tools on Ubuntu

The Ubuntu 20.04 LTS Desktop distribution includes most of what you need to build CircuitPython. You'll need to install some additional packages, including **build-essential**, if it's not already installed, and also **gettext** and **uncrustify**. In a terminal window, do:

```
sudo apt update
# Try running `make`. If it's not installed, do:
# sudo apt install build-essential
sudo apt install git gettext uncrustify
```

Next you need to download and unpack the ARM gcc toolchain from its download page (https://adafru.it/HCO). ARM is no longer updating its Ubuntu "ppa" (private package archive), so you need to install the toolchain manually. (The links below point to the x64 versions of the toolchain. If you are building on some other architecture, download the appropriate toolchain.)

- For CircuitPython 6.1 and later, use the 10-2020-q4-major (https://adafru.it/Pid) version.

If you want to build an older version of CircuitPython:

- For CircuitPython 5 and 6.0, use the 9-2019-q4-major version (https://adafru.it/HCP).
- CircuitPython 4 was built with the 7-2018q2-update (https://adafru.it/HCQ) version.

```
# Put the unpacked toolchain into an appropriate directory.
# This is an example.
cd ~/bin
tar xvf <name of the .bz2 file you downloaded>
```

Next, add a line to your **.bash_profile** or other startup file to add the unpacked toolchain executables to your **PATH**. For example:

```
export PATH=/home/$USER/bin/gcc-arm-none-eabi-10-2020-q4-major/bin:$PATH
```

Open a new terminal window, and see if you now have the correct executables on your path:

```
which arm-none-eabi-gcc
/home/halbert/bin/gcc-arm-none-eabi-10-2020-q4-major/bin/arm-none-eabi-gcc
```

Now move to the Build CircuitPython (https://adafru.it/C2D) section of this guide. There we will get the CircuitPython source code, install a few more dependencies, and then build!

MacOS Setup

To build CircuitPython on MacOS, you need to install **git**, **python3**, and the ARM toolchain. The easiest way to do this is to first install Homebrew (https://adafru.it/wPC), a software package manager for MacOS. Follow the directions on its webpages.

Now install the software you need:

```
brew update
brew install git python3 gettext uncrustify
brew link gettext --force
```

And finally, install the ARM toolchain. Use the MacOS .pkg file from https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads (https://adafru.it/HCO). As of this writing (April 2021), use the **gcc-arm-none-eabi-10-2020-q4-major-mac.pkg** file.

Download the .pkg file and double-click it to install.

Once the dependencies are installed, we'll also need to make a disk image to clone CircuitPython into. By default the Mac OSX filesystem is case insensitive. In a few cases, this can confuse the build process. So, **we recommend** creating (https://adafru.it/CaT)a case sensitive disk image with Disk Utility (https://adafru.it/CaT) to store the source code. Make the image capable of storing at least 10GB, so you won't run out of space if you do extensive development. You can use a sparse bundle disk image which will grow as necessary, so you don't use up all the space at once. The disk image is a single file that can be mounted by double clicking it in the Finder. Once it's mounted, it works like a normal folder located under the **/Volumes** directory.

That's it! Now you can move on and actually Build CircuitPython (https://adafru.it/C2D). There we will get the CircuitPython source code, install a few more dependencies, and then build!

Windows Subsystem for Linux Setup

Windows Subsystem for Linux (WSL) is a feature of Windows 10 that lets you run Ubuntu and other versions of Linux right in Windows. It's real Ubuntu, without the Linux kernel, but with all the software packages that don't need a graphical interface. You can build CircuitPython in WSL easily. It's easier to install than a Linux virtual machine.

Install WSL

The installation procedures for WSL continue to evolve. Rather than provide information here which quickly becomes outdated, we ask that you refer to Microsoft's official instructions: Windows Subsystem for Linux Installation Guide for Windows 10 (https://adafru.it/ReS). Enable the WSL 2, which is better for our purposes than WSL 1 in several ways.

One WSL 2 is set up, you need to choose a Linux distribution to install, as described in the document above. Choose Ubuntu 20.04.

Build CircuitPython

From this point on, you can build CircuitPython just as it's built in regular Ubuntu, described in the Build CircuitPython (https://adafru.it/C2D) section of this guide.

Moving Files to Windows

You can copy files to and from Windows through the /mnt/c. For instance, if you want to copy a CircuitPython build to the desktop, do:

```
cp build-circuitplayground_express/firmware.uf2 /mnt/c/Users/YourName/Desktop
```

Warning: Don't build in a shared folder (in /mnt/c). You'll probably have filename and line-ending problems.

You might be tempted to clone and build CircuitPython in a folder shared with the Windows filesystem (in **/mnt/c** somewhere). That can cause problems, especially if you use **git** commands on the Windows side in that folder. The CircuitPython build assumes case-sensitive filenames, but Windows usually ignores filename case differences. You may also have line-ending problems (CRLF vs. LF). It's better to clone and build inside your home directory in WSL, and copy files over to a shared folder as needed.

Mounting a CircuitPython Board in WSL

You can mount your **...BOOT** or **CIRCUITPY** drive in WSL. Create a mount point and then mount it. Note that you'll have to remount each time the drive goes away, such as when you restart the board or switch between the **BOOT** drive and **CIRCUITPY**. So it's probably more convenient to copy files to the board from Windows instead of WSL.

```
# You only need to do this once.

# Choose the appropriate drive letter.
sudo mkdir /mnt/d

# Now mount the drive.
sudo mount -t drvfs D: /mnt/d

# Now you can look at the contents, copy things, etc.
ls /mnt/d
cp firmware.bin /mnt/d
# etc.
```

Manual Setup

If the setup instructions above don't work for your particular OS setup, for whatever reason, you can get the ball rolling by installing these tools in whatever way you can and then getting them to work with the **Makefile** in **circuitpython/ports/atmel-samd**. (**main** branch):

- **git**
- **make**
- **python3**
- **gettext**
- All Python packages listed in **requirements-dev.txt**. You will need to clone the repository to see that file.
- ARM gcc toolchain: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads (https://adafru.it/C2E)
  See information about which gcc version to pick on the Linux Setup page (https://adafru.it/HCR) of this guide.

Build CircuitPython

Fetch the Code to Build

Once your build tools are installed, fetch the CircuitPython source code from its GitHub repository ("repo") and also fetch the git "submodules" it needs. The submodules are extra code that you need that's stored in other repos.

In the commands below, you're cloning from Adafruit's CircuitPython repo. But if you want to make changes, you might want to "fork" that repo on GitHub to make a copy for yourself, and clone from there.

```
git clone https://github.com/adafruit/circuitpython.git
cd circuitpython
git submodule sync --quiet --recursive
git submodule update --init
```

We are not using `git submodule update --init --recursive` because it brings in a large number of unneeded submodules under tinyusb.

If you want to build a version other the latest, checkout the branch or tag you want to build. For example:

```
# Build using the latest code on the 6.3.x branch.
git checkout 6.3.x

# Build the 6.2.0 version exactly.
git checkout 6.2.0
```

Note the build process has evolved, and earlier versions will need to be built somewhat differently than how the instructions in this guide specify. If you have trouble, ask on Discord () or the forums (https://adafru.it/jlf).

Install Required Python Packages

After you have cloned the repo, you'll need to install some Python packages that are needed for the build process. You only need to do this the first time, though you may want to run this again from time to time to make sure the packages are up to date.

```
# Install pip if it is not already installed (Linux only)
sudo apt install python3-pip

# Install needed Python packages from pypi.org.
pip3 install -r requirements-dev.txt

# Force a particular version of click. It does not get updated to the
# proper version sometimes
pip3 install --upgrade click==7.1.2
```

You may see errors if you have too-old or too-new versions of some of these packages already installed, and have to fix an installation by installing a package with a version specifier. For example, if the `whatever` package is too new, you may need to do `pip3 install --upgrade 'whatever<2.1.0'`.

Install **pre-commit**

We are using the **pre-commit** system to check submitted code for problems before it gets to GitHub. For more information, see this Learn Guide page (https://adafru.it/check-your-code). To add **pre-commit** to your repository clone, do:

```
cd <your repository clone directory>
# You only need to do this once in each clone.
pre-commit install
```

Build **mpy-cross**

Build the **mpy-cross** compiler first, which compiles Circuitpython **.py** files into **.mpy** files. It's needed to include library code in certain boards.

(If you get a **make: msgfmt: Command not found** error, you have not installed **gettext**. Go back to the Setup page for your operating system.)

Normally you do not need to rebuild **mpy-cross** on every pull or merge from the `circuitpython` repository or for your own changes. The **.mpy** format does not change very often. But occasionally when we merge from MicroPython, the format changes. You will find that your old **.mpy** files or frozen libraries give an error, and you will need to rebuild **mpy-cross**.

```
make -C mpy-cross
```

Build CircuitPython

Now you're all set to build CircuitPython. If you're in the master branch of the repo, you'll be building the latest version. Choose which board you want to build for. The boards available are all the subdirectories in **ports/atmel-samd/boards/**.

```
cd ports/atmel-samd
make BOARD=circuitplayground_express
```

By default the en_US version will be built. To build for a different language supply a **TRANSLATION** argument.

```
cd ports/atmel-samd
make BOARD=circuitplayground_express TRANSLATION=es
```

Run Your Build!

When you've successfully built, you'll see output like:

```
Create build-circuitplayground_express/firmware.bin
Create build-circuitplayground_express/firmware.uf2
python2 ../../tools/uf2/utils/uf2conv.py -b 0x2000 -c -o build-
circuitplayground_express/firmware.uf2 build-circuitplayground_express/firmware.bin
Converting to uf2, output size: 485888, start address: 0x2000
Wrote 485888 bytes to build-circuitplayground_express/firmware.uf2.
```

Copy **firmware.uf2** to your board the same way you'd update CircuitPython: Double-click to get the **BOOT** drive, and then just copy the **.uf2** file:

```
# Double-click the reset button, then:
cp build-circuitplayground_express/firmware.uf2 /media/yourname/CPLAYBOOT
```

The board will restart, and your build will start running.

If you're using a board without a UF2 bootloader, you'll need to use **bossac** and the **firmware.bin** file, not the **.uf2** file. Detailed instructions are here (https://adafru.it/Bid).

Use All Your CPUs When Building

Most modern computers have CPU chips with multiple cores. For instance, you may have a 2-core, 4-core, or 6-core or more CPU. Your CPU may also allow 2 "threads" per core, so that it appears to have even more cores. You can run much of the build in parallel by using the `make -j` flag. This is will speed up the build noticeably.

If you don't know how many cores or threads your CPU has, on Linux you can use this command:

```
getconf _NPROCESSORS_ONLN
12
# This CPU has 6 cores and 12 threads.
```

Then, when you run make, add the -j<n> option to use as many cores or threads as possible. For example:

```
make -j12 BOARD=trinket_m0
```

When to `make clean`

After you make changes to code, normally just doing `make BOARD=...` will be sufficient. The changed files will be recompiled and CircuitPython will be rebuilt.

However, there are some circumstance where you must do:

```
make clean BOARD=...
```

If you have changed the `#include` file structure in certain ways, or if you have defined QSTR's (a way of defining constants strings in the CircuitPython source), then you must `make clean` before rebuilding. If you're not sure, it's always safe to `make clean` and then `make`. It might take a little longer to build, but you'll be sure it was rebuilt properly.

Updating Your Repo

When there are changes in the GitHub repo, you might want to fetch those and then rebuild. Just "pull" the new code (assuming you haven't made changes yourself), update the submodules if necessary, and rebuild:

```
git pull
git submodule sync
git submodule update --init
# Then make again.
```

Those are the basics. There's a lot more to know about how to keep your forked repo up to date, merge "upstream" (Adafruit's) changes into your code, etc. We cover this in the Contribute to CircuitPython with Git and GitHub (https://adafru.it/Dkh) guide

Adding Frozen Modules

Normally, all imported Python modules in CircuitPython are loaded into RAM in compiled form, whether they start as `.mpy` or `.py` files. Especially on M0 boards, a user program can run out of RAM if too much code needs to be loaded.

To ameliorate this problem, a CircuitPython image can include compiled Python code that is stored in the image, in flash memory, and executed directly from there. These are "internal frozen modules". The `circuitplayground_express` builds use this technique, for example.

If you would like to build a custom image that includes some frozen modules, you can imitate how it's done in the `circuitplayground_express` build. Look at `boards/circuit_playground_express/mpconfigboard.mk`:

```
USB_VID = 0x239A
USB_PID = 0x8019
USB_PRODUCT = "CircuitPlayground Express"
USB_MANUFACTURER = "Adafruit Industries LLC"

CHIP_VARIANT = SAMD21G18A
CHIP_FAMILY = samd21

SPI_FLASH_FILESYSTEM = 1
EXTERNAL_FLASH_DEVICES = "S25FL216K, GD25Q16C"
LONGINT_IMPL = MPZ

# Turn off displayio to make room for frozen libs.
CIRCUITPY_DISPLAYIO = 0

# Now we actually have a lot of room. Put back some useful modules.
CIRCUITPY_BITBANGIO = 1
CIRCUITPY_COUNTIO = 1
CIRCUITPY_BUSDEVICE = 1

# Include these Python libraries in firmware.
FROZEN_MPY_DIRS += $(TOP)/frozen/Adafruit_CircuitPython_CircuitPlayground
FROZEN_MPY_DIRS += $(TOP)/frozen/Adafruit_CircuitPython_HID
FROZEN_MPY_DIRS += $(TOP)/frozen/Adafruit_CircuitPython_LIS3DH
FROZEN_MPY_DIRS += $(TOP)/frozen/Adafruit_CircuitPython_NeoPixel
FROZEN_MPY_DIRS += $(TOP)/frozen/Adafruit_CircuitPython_Thermistor
```

Notice the `FROZEN_MPY_DIRS` lines in the file. Pick the **mpconfigboard.mk** file for the board you are using, and add one or more similar lines. You will need to do add directories for the libraries you want to include. If these are existing libraries in GitHub, you can add them as submodules. For instance, suppose you want to add the **Adafruit_CircuitPython_HID** library to the **feather_m0_express** build. Add this line to **boards/feather_m0_express/mpconfigboard.mk**:

```
FROZEN_MPY_DIRS += $(TOP)/frozen/Adafruit_CircuitPython_HID
```

Then add the library as a submodule:

```
cd circuitpython/frozen
git submodule add https://github.com/adafruit/Adafruit_CircuitPython_HID
```

Set the submodule to a commit that is a release tag. If you try to freeze a module that is at untagged commit, you'll get a git error when building.

When you add the submodule it will be cloned into the **frozen/** directory.

Note that there is limited unused space available in the images, especially in the non-Express M0 builds, and you may not be able to fit all the libraries you want to freeze. You can of course try to simplify the library code if necessary to make it fit.

Choosing a Different SPI Flash Chip

CircuitPython supports external SPI/QSPI flash chips for the CIRCUITPY filesystem. Each board build is setup to support only one or a few flash chips. The chips are not identical in how they are accessed, so you can't just substitute without rebuilding. The chips that CircuitPython currently supports are listed in a GitHub repository of chip data, https://github.com/adafruit/nvm.toml (https://adafru.it/RAQ), along with the settings needed for each.

The chip(s) that are supported in a particular board are specified in the **mpconfigboard.mk** file for that board, in the line that defines `EXTERNAL_FLASH_DEVICES`. So change or add to `EXTERNAL_FLASH_DEVICES` if you want to use a different supported chip. We don't support a entire list of chips for each build because the table of data for all possible chips would take significant space in the CircuitPython build.

Here's an example:

```
EXTERNAL_FLASH_DEVICES = "S25FL116K, S25FL216K, GD25Q16C"
```

Customizing USB Devices

You can customize which USB devices to include in your CircuitPython build by adding some settings to the **circuitpython/ports/*port-choice*/boards/*board-choice*/mpconfigboard.mk** file.

In CircuitPython 7.0.0 and beyond, most devices are on by default. You can turn them and off at runtime. See the guide Customizing USB Devices in CircuitPython (https://adafru.it/Sxf).

Customizing USB Devices

You can enable or disable the different available USB devices by using these flags and setting them to 1 or 0. You do not need to specify all these settings, only the ones that are different from the defaults for your build.

```
# Enable second CDC (serial) channel (usually off; was on in 6.2.0-beta.3)
CIRCUITPY_USB_CDC = 1

# Disable composite HID device (usually on)
CIRCUITPY_USB_HID = 0

# Disable MIDI (sometimes on)
CIRCUITPY_USB_MIDI = 0

# Disable mass storage (on by default)
CIRCUITPY_USB_MSC = 0

# Enable WEBUSB (off by default)
CIRCUITPY_USB_VENDOR = 1
```

Customizing USB HID Devices

Similarly, these flags control whether various USB HID devices are enabled or disabled. You only need to specify the ones that are different from the defaults for your build.

These settings do not exist in CircuitPython 7.0.0. See Customizing USB Devices in CircuitPython (https://adafru.it/Sxf).

```
# Disable Consumer Control (volume, etc.) (on by default)
CIRCUITPY_USB_HID_CONSUMER = 0

# Enable digitizer tablet (off by default)
CIRCUITPY_USB_HID_DIGITIZER = 1

# Disable HID gamepad (usually on)
CIRCUITPY_USB_HID_GAMEPAD = 0

# Disable keyboard (always on)
CIRCUITPY_USB_HID_KEYBOARD = 0

# Disable mouse (always on)
CIRCUITPY_USB_HID_MOUSE = 0

# Enable SysControl (power, etc.) (off by default)
CIRCUITPY_USB_HID_SYS_CONTROL = 1

# Enable Microsoft XAC gamepad (off by default)
CIRCUITPY_USB_HID_XAC_COMPATIBLE_GAMEPAD = 1
```

Side note: Don't put Makefile comments on the same line as Makefile assignments. I causes the variable value to include the trailing spaces before comment character (https://adafru.it/QEu).

Customizing USB Devices Before 6.2.0-beta.3

If you are changing a build before 6.2.0-beta.3 (actually before PR 4215 (https://adafru.it/QEp)), use this older way of specifying which USB devices to include. Add a line like one of these:

```
# This is the default if USB_DEVICES is not specified.
# AUDIO refers to MIDI capability.
USB_DEVICES = "CDC,MSC,AUDIO,HID"

# Here's an example of providing only HID devices
USB_DEVICES = HID

# Provide only CDC and HID
USB_DEVICES = "CDC,HID"
```

Of course, if you disable MSC, you will not be able to load and edit programs via the CIRCUITPY drive. So get your program running with a regular CircuitPython build, and when you are satisfied, replace it with your custom build. If you need to edit the program again, reload the regular build.

Customizing HID Devices Before 6.2.0-beta.3

Similarly, you can specify which HID devices are available:

```
# This is the default if USB_HID_DEVICES is not specified
USB_HID_DEVICES = "KEYBOARD,MOUSE,CONSUMER,GAMEPAD"

# Provide only KEYBOARD and CONSUMER HID devices, for use as, say, a volume control.
USB_HID_DEVICES = "KEYBOARD,CONSUMER"
```

ESP32-S2 Build

The **ports/esp32s2** build setup is a rather involved process. Please read the **README.md** in that directory. It has instructions that also refer to further instructions in the ESP-IDF documentation.

On MacOS, you will need to install **cmake:**

```
brew install cmake
```

On Linux you probably need to install **ninja-build** and **cmake**.

```
sudo apt install ninja-build cmake
```

The ESP-IDF expects there to be a **python** command which runs **python3**. On Ubuntu, there is no plain **python** by default, so install this simple package which links **python** to **python3**.

```
sudo apt install python-is-python3
```

Once you have the prerequisites installed, change to the **ports/esp32s2** directory, and run the **install.sh** script. You only need to do this once.

```
cd circuitpython/ports/esp32s2
esp-idf/install.sh
```

After this, in each fresh terminal window in which you are doing builds, you need to use **esp-idf/export.sh** in order to set up the correct **PATH** and other environment variables.

```
# Do this in each new terminal.
cd circuitpython/ports/esp32s2
source esp-idf/export.sh
```

Now you can build, for example:

```
make BOARD=adafruit_magtag_2.9_grayscale
```

How to Add a New Board to CircuitPython

How to Add a New Board to CircuitPython (https://adafru.it/PBG)