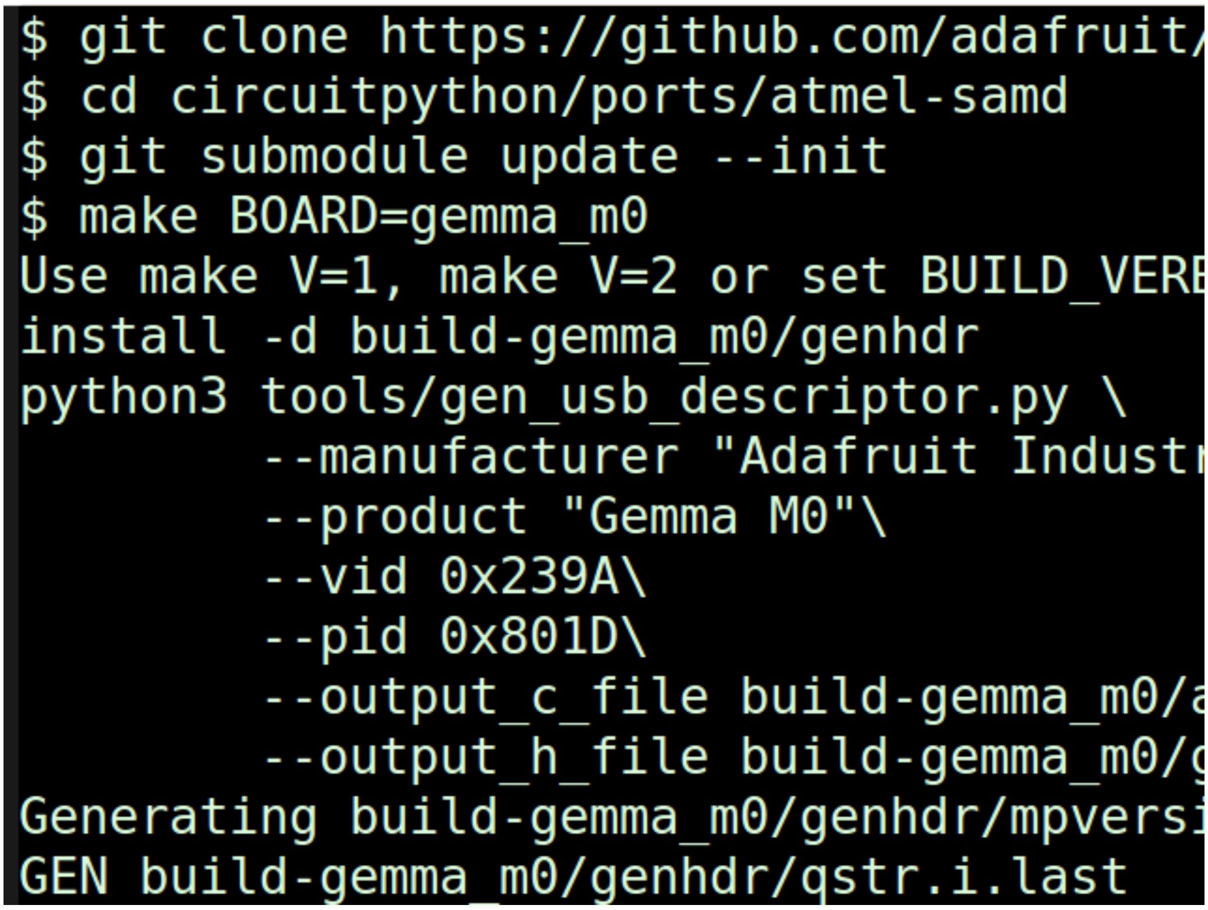
Building CircuitPython

Created by Dan Halbert



```
$ git clone https://github.com/adafruit/
$ cd circuitpython/ports/atmel-samd
$ git submodule update --init
$ make BOARD=gemma_m0
Use make V=1, make V=2 or set BUILD_VERE
install -d build-gemma_m0/genhdr
python3 tools/gen_usb_descriptor.py \
        --manufacturer "Adafruit Industr
        --product "Gemma M0"\
        --vid 0x239A\
        --pid 0x801D\
        --output_c_file build-gemma_m0/a
        --output_h_file build-gemma_m0/g
Generating build-gemma_m0/genhdr/mpversi
GEN build-gemma_m0/genhdr/qstr.i.last
```

https://learn.adafruit.com/building-circuitpython

Last updated on 2025-05-30 02:59:28 PM EDT

Table of Contents

Introduction

Adafruit CircuitPython (https://adafru.it/AlP) is an open-source implementation of Python for microcontrollers. It's derived from 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

Set Up 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 and 11. WSL 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 24.04 LTS (Long Term Support) version is stable and reliable, but needs a Python **venv** setup, detailed on the Build CircuitPython (https://adafru.it/C2D) page in this guide. Ubuntu 22.04 can also be used, but some of the tools are out of date. Use 24.04 if you can.

Native Linux

You can install Ubuntu on a bare machine easily. Follow the directions (https://adafru.it/10Sc) 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. Make your virtual machine filesystem at least 20GB-40GB so you won't run out of space.

Raspberry Pi

Raspberry Pi OS is a Debian-based Linux distribution, like Ubuntu. Use the latest version of Raspberry Pi OS so that the tools are as up to date as possible. You can also install Ubuntu on your Raspberry Pi. You will need to download the **aarch64** executable for the **arm-none-eabi-gcc** toolchain. The RPis, particularly Pi 3 and earlier, are not fast machines: be prepared to wait a while for a build to complete.

Install Build Tools on Ubuntu

The Ubuntu 24.04 and 22.04 LTS Desktop distributions include 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 **git**, **git-lfs**, **gettext,** and **cmake**.

You also need the **uncrustify** tool if using pre-commit, but that will be installed later, when you install the Python dependencies, even though uncrustify is not a Python tool per se.

The version of **git** (2.30) installed with Ubuntu 22.04 will work, but git versions that support partial submodule cloning (2.36 or later) will work better with submodules. The standard version of git in Ubuntu 24.04 works fine.

To install these tools, in a terminal window, do:

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

# If you don't have add-apt-repository, do:
# sudo apt install software-properties-common

# Recommended on Ubuntu 22.04: use the latest stable version of git:
# Optional and not necessary on Ubuntu 24.04.
sudo add-apt-repository ppa:git-core/ppa

sudo apt install git git-lfs gettext cmake
```

Cortex-M Builds

Most CircuitPython boards use an ARM Cortex-M processor. You need to download and unpack the appropriate ARM Cortex-M toolchain. Do not use the obsolete ARM Ubuntu "ppa" (private package archive). The links below point to the general download pages for the toolchains. Choose the **arm-none-eabi** version of the compiler, and select the correct architecture for your development computer, such as **x64 Linu**x for a typical Linux PC. The package library for your particular Linux distribution will not necessary install the correct version.

- For CIrcuitPython 9.x, use the 13.2.Rel1 version for (https://adafru.it/19c0)AArch32 bare-metal target (arm-none-eabi) (https://adafru.it/19c0) on the Arm GNU Toolchain Downloads page. Scroll down to find the correct version.
- For CircuitPython 10.x use the 14.2.Rel1 version for (https://adafru.it/19c0)AArch32 bare-metal target (arm-none-eabi) (https://adafru.it/19c0) on the Arm GNU Toolchain Downloads page. Scroll down to find the correct version. GCC 14 is required as of PR #10386 (https://adafru.it/1ajH), which is after 10.0.0-alpha.6.

If you want to build an older version of CircuitPython:

- For CircuitPython 6.1, 7.x, and 8.x, use the 10-2020-q4-major version on this page (https://adafru.it/19bZ).
- For CircuitPython 5 and 6.0, use the 9-2019-q4-major version on this page (https://adafru.it/19bZ).
- CircuitPython 4 was built with the 7-2018q2-update version on this page (https://adafru.it/19bZ).

Cortex-A Builds (Only for **broadcom** Port)

The **broadcom** port (Raspberry Pi Linux boards, not Pi RP2040 or RP2350) needs a different toolchain. Use the Cortex-A toolchain:

- For CircuitPython 7.x, 8.x, and 9.x, use the 10.3-2021.07 version (https://adafru.it/1ajw).
- For CircuitPython 10.x, use the 13.3-Rel1 AArch64 bare-metal target (aarch64-none-elf) (https://adafru.it/19c0).

You will also need the **mtools** package for Cortex-A:

```
sudo apt install mtools
```

And finally, you need a version 4.2 or later of **mkfs.fat,** which is part of the **dosfstools** package. Ubuntu 20.04 has version 4.1. If you are still running 20.04. You can download and build dosfstools. After you do so, copy mkfs.fat to some place that is or will be on your **PATH**.

(This is not necessary on Ubuntu 22.04 or later.)

```
wget https://github.com/dosfstools/dosfstools/releases/download/v4.2/
dosfstools-4.2.tar.gz
tar xvf dosfstools-4.2.tar.gz
cd dosfstools-4.2
./configure
make
```

Installing the Toolchain

Unpack the Cortex-M or Cortex-A toolchain in a convenient directory.

```
# This is an example.
cd ~/bin
tar xvf &lt;name of the .bz2 or .xz file you downloaded&gt;
```

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/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi/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/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi/bin/arm-none-
eabi-gcc
```

Other Builds

- For Espressif (ESP32-S2, -S3, -C3), see the Espressif build page (https://adafru.it/Ykd).
- For Spresense (cxd56), see the cxd56 README (https://adafru.it/Yke).
- For Litex (FOMU), see the litex README (https://adafru.it/Ykf).

Setting up a Python Virtual Environment (Ubuntu 24.04 and similar only)

On Ubuntu 24.04 and other recent Debian-derived Linux distribution releases, you can no longer install Python modules with **pip** without first setting up a Python virtual environment. You can override this restriction, but it has been set up as a safeguard.

The guide Python Virtual Environment Usage on Raspberry Pi (https://adafru.it/19a5) explains this in great detail, from the point of view of using Raspberry Pi OS.

For building CircuitPython, here is a short recipe. First, make sure that **venv**, the virtual environment module, is installed. Then create the virtual environment. Here we name it **.py**, but you can pick a different name.

```
# Install if not already installed.
sudo apt install python3-venv

# Put the venv (virtual environment) in your home directory.
cd

# Create the venv and name it `.py` (or whatever you'd like).
python3 -mvenv .py
```

Once you create a virtual environment, you "activate" it by sourcing the **activate** script provided by the virtual environment. Note that you cannot just run the script. You have to source it into the current shell using the `source` or the `.` command:

```
# Enable the virtual environment.
source .py/bin/activate
```

Your shell will probably show an indication that you are now using the Python virtual environment named **.py**:

```
(.py) yourusername@yourmachine:~$
```

Now you are ready to start building. You'll install the Python modules you need while you are inside this virtual environment.

Note that if you're doing an Espressif build, it creates its own virtual environment when you run `esp-idf/export.sh`. So don't use the virtual environment you created above for those builds.

To leave the virtual environment, you deactivate it:

```
deactivate
```

Using the Virtual Environment Automatically

If it's convenient for you, you can activate the virtual environment in any new terminal by putting a line like this in your **.bashrc**, **.bash_aliases**, or other shell startup file:

```
source ~/.py/bin/activate
```

Next Steps

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 compiler 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 git-lfs python3 gettext uncrustify cmake
brew link gettext --force
```

Then install the compiler toolchain. You may be able to use the **gcc-arm-embedded** homebrew formula (https://adafru.it/19c2) to install the correct toolchain, but make sure you have **homebrew** install the correct version, as described on the Linux Setup (https://adafru.it/HCR) page in this guide. A version that is too old or too new may cause compilation errors.

As an alternative to **homebrew**, on the Linux Setup (https://adafru.it/HCR) page, you can find pointers to the download pages for different kinds of ARM processors, and also special instructions for other builds (not all are available on macOS). Download and install the appropiate **.pkg** file for your Mac.

Once the dependencies are installed, we'll also need to make a disk image to clone CircuitPython into. By default the macOS 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, preferably 20GB, 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.

Install `gmake` if Necessary

Depending on your version of macOS, whether or not you have installed Xcode, and other factors, the version of make on your Mac may not be recent enough to run builds. Check the version:

```
make --version
# and
gmake --version
```

If neither program exists, or both are older than version 4.3, install a newer version of `gmake` with `brew`:

```
brew install make
```

If you install the `brew` version of make, it will be known as `gmake`.

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 (WSL) Setup

Windows Subsystem for Linux (WSL) is a feature of Windows 10 and 11 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: How to install Linux on Windows with WSL (https://adafru.it/1aby). Enable WSL 2, which is better for our purposes than WSL 1 in several ways.

Note that your PC must have hardware virtualization support. Virtualization must also be enabled in the BIOS or UEFI on your motherboard.

One WSL 2 is set up, you need to choose a Linux distribution to install, as described in the document above. Choose Ubuntu 24.04 if you can.

Finish Linux Install

Once WSL is set up, just proceed with the regular Linux Setup (https://adafru.it/HCR). You will install the Linux-based tools, not Windows-based tools

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. It is possible to drag and drop files between WSL and Windows.

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.
```

Editing Files in WSL

You can use the usual Linux editors in WSL, such as **vim** and **emacs**. You may need to install them explicitly. Visual Studio Code (not Visual Studio) is also available: do a websearch to find the latest instructions. Visual Studio Code in Windows has an extension that supports WSL.

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**
- **git-lfs**
- **make**
- **python3**
- **gettext**
- **cmake**
- All Python packages listed in **requirements-dev.txt**. You will need to clone the repository to see that file.
- ARM gcc toolchain for Cortex-M boards: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads (https://adafru.it/C2E)
- ARM gcc toolchain for Cortex-A boards (only needed for `broadcom` port): https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads (https://adafru.it/Ykb)
  See information about which gcc version to pick on the Linux Setup page (https://adafru.it/HCR) of this guide.

GitHub Codespaces

GitHub Codespaces provides a cloud-based development environment which you can use instead of creating and maintaining a local development environment. There is a free tier that should be sufficient for casual CircuitPython development.

GitHub user @bablokb (https://adafru.it/19If) has set up and plans to maintain GitHub Codespaces "devcontainers" for the different CIrcuitPython port builds. For more information, see Using Github Codespaces for CircuitPython Development (https://adafru.it/19IA) in the Adafruit Playground (https://adafru.it/18fM). Thank you, @bablokb!

Recent Changes

The build setup and steps in this guide change frequently, and are updated as needed. If you are are having trouble, check this page, which highlights recent changes, to see if you need to update your build environment. The changes mentioned here are also documented in the other pages in this guide.

Install **git-lfs**

git-lfs is now a required prerequisite, as of CircuitPython 8.1.0. The submodules in the `silabs` port require it.

Fetching Submodules

Submodules should be fetched using the Makefile targets `fetch-all-submodules`, which is in the top-level **Makefile**, or `fetch-port-submodules`, which is used from port-specific Makefiles. If you use standard git commands for fetching submodules, you will end up fetching many gigabytes of unnecessary versions.

See this page section (https://adafru.it/C2D) for more details.

These make targets do either a partial "blobless" clone of all submodules, if your version of **git** supports it, or else does a shallow depth 1 clone.

`make remove-all-submodules` removes all submodules and cleans up, so you can do a fresh `make fetch-all-submodules` or `fetch-port-submodules`. Use `make remove-all-submodules` if you are getting errors when fetching submodules.

These Makefile targets were added June 6, 2023, changing a previous target, added in late March, 2023 that only fetched all modules, `make fetch-submodules`.

Build CircuitPython

If you are building for an Espressif board, read this section and also read the Espressif Builds page in this guide for additional instructions.

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.

Fork the `adafruit` repo if you plan to submit pull requests back to `circuitpython`

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. For more information about using GitHub and forking a repo, see the Contribute to CircuitPython with Git and GitHub (https://adafru.it/Fj0) guide.

If you are on macOS and you had to install the **brew** version of **make**, use **gmake** in all examples below instead of **make**.

```
git clone https://github.com/adafruit/circuitpython.git
cd circuitpython
```

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). Try running pip first.
sudo apt install python3-pip

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

The `--upgrade` flag will force installation of the latest packages, except where a version is explicitly specified in the **requirements-*.txt** files.

If you get an error indicating that `rust` is needed to install `minify-html`, install `rust`:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# You can verify the installation by running:

source $HOME/.cargo/env #Import the environment config for rust
rustc --version
```

Checking out a Specific Branch or Version

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 main branch.
git checkout main

# Build the latest code on the 9.0.x branch
git checkout 9.0.x

# Build the 9.2.1 version exactly.
git checkout 9.2.1
```

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/jIf).

Fetch Submodules

We are not using `git submodule update --init --recursive` or `git submodule update --init`. Instead you run the special **Makefile** target `make fetch-all-submodules` at the top level, or `make fetch-port-submodules` when you in a particular **ports/port-name** directory. The target fetches only as many commits from each submodule as is necessary, using either a blobless partial clone (if available) or a shallow clone. This avoids downloading the complete trees of some large submodules, saving time and space.

`make fetch-all-submodules` fetches all the submodules in the entire repository, and can take several minutes and use up a lot of storage. If you are planning to build only in one or a few port directories, you'll save time by using `make fetch-port-submodules`, which fetches only the submodules need for that particular port.

If you are having trouble with your submodules, you can clean out all the fetched modules by doing `make remove-all-submodules` at the top level. Then run `make fetch-all-submodules` or `make fetch-port-submodules` again to fetch a fresh copy of all the submodules. (There is no `make remove-port-submodules`.)

Using a Version of git that can do Partial Clones

As mentioned above, the submodule fetching targets try to do a blobless partial clone if possible. This is better than a shallow clone: it is slightly faster, and the partial clone acts like a full clone if you want to maneuver around in the submodule by checking out other commits, examining the history, etc.

Git version 2.36 or later supports blobless partial clones. Ubuntu 24.04 includes such a version. To find out which version of git you are using, do `git --version`. Consider installing a newer version of git if it is available. On earlier versions of Ubuntu, you can install the git PPA (https://adafru.it/19c3) and get the latest stable version of git. On macOS, **homebrew** may provide a later version for you.

```
cd circuitpython   # go to the top level
make fetch-all-submodules

# OR, for example, to fetch only the submodules needed for RP2040 builds:

cd circuitpython/ports/raspberrypi
make fetch-port-submodules
```

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 &lt;your repository clone directory&gt;
# You only need to do this once in each clone.
pre-commit install
```

Are you seeing errors when pre-commit runs? See this hint.

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 builds that use frozen modules (https://adafru.it/1abz).

(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 **main** 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, if you have defined QSTR's (a way of defining constants strings in the CircuitPython source), or if you have added new error messages, 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
 # only if necessary, from the top level directory
make fetch-submodules
# 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`:

```
Temporarily unable to load content:
```

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
```

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

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. You can update all the frozen modules to the latest release tags by doing this, at the top level of your repository clone:

```
make update-frozen-libraries
```

Alternatively, simply check out a tagged version of the submodule after you add the submodule and before you commit it:

```
cd circuitpython/frozen/The_Module_to_Freeze   # example submodule name
git checkout 2.11.0   # example version number
```

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 Included Modules

You may want to include a particular module that is not included by default for the board for your board. You can customize which modules to include in your CircuitPython build by adding or changing settings in the **mpconfigboard.mk** file for your board.

For example, here is the standard **circuitpython/ports/atmel-samd/boards/trinket_m0/mpconfigboard.mk file:**

```
USB_VID = 0x239A
USB_PID = 0x801F
USB_PRODUCT = "Trinket M0"
USB_MANUFACTURER = "Adafruit Industries LLC"

CHIP_VARIANT = SAMD21E18A
CHIP_FAMILY = samd21

INTERNAL_FLASH_FILESYSTEM = 1
```

```
LONGINT_IMPL = NONE
CIRCUITPY_FULL_BUILD = 0
```

Suppose you wanted to turn on `pulseio`, but you did not need MIDI support. There is not enough room for `pulseio` in this build, but if you turn other modules, you can make room. Sometimes this will work only for certain languages due to the size of the translation. In this case, for the `en_US` translation, turning off `usb_midi` frees up enough space to include `pulseio`. Add these lines to the file above:

```
CIRCUITPY_PULSEIO = 1
CIRCUITPY_USB_MIDI = 0
```

You can figure out which modules are on or off by looking at the module support matrix in the CIrcuitPython documentation (https://adafru.it/N2a). It also helps to study the files **circuitpython/py/circuitpy_mpconfig.mk** and **circuitpympconfig.h, and circuitpython/ports/<your-board's-port>/mpconfigport.mk** and **mpconfigport.h**.

Warning: 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).

Espressif Builds

The **ports/espressif** build setup is a rather involved process. Start with the setup described on the Build CircuitPython (https://adafru.it/C2D) page. Don't forget to `make fetch-port-submodules` in **ports/espressif** or `make-fetch-all-submodules` at the top level.

Linux

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
```

macOS

On macOS, you will need to install **cmake** and **ninja:**

```
brew install cmake
brew install ninja
```

Run the ESP-IDF Installation Script Once

Once you have the prerequisites installed, change to the **ports/espressif** directory, and run the **esp-idf/install.sh** script. You only need to do this once: **install.sh** downloads the toolchains and other files it needs, and copies a number of files into **~/.espressif**. (If the ESP-IDF version used by CircuitPython changes, you will need to run **install.sh** again.)

install.sh also sets up a Python venv (virtual environment) in **~/.espressif/python-dev**. If you are already using a venv, maybe because you're using a recent version of Linux, you'll need to `deactivate` that venv before running **install.sh**. Otherwise you'll get an error that you can't create a venv inside another venv.

```
cd circuitpython/ports/espressif

# If you are already inside a Python venv, deactivate it.
deactivate

esp-idf/install.sh
```

Install Python Packages in the ESP-IDF venv

After running **export.sh**, you need to repeat the `pip3 install` commands described here (https://adafru.it/C2D) so that they take effect inside the Python environment set up by `source esp-idf/export.sh`. But you will need to do this only once.

```
# Install needed Python packages from pypi.org after running install.sh or if the
needed packages change.
pip3 install --upgrade -r requirements-dev.txt
pip3 install --upgrade -r requirements-doc.txt
```

Setting Up the ESP-IDF Environment Before Building

You only need to run **esp-idf/install.sh** once. But later, in each fresh terminal window in which you are doing builds, you need to run **esp-idf/export.sh** in order to set up the correct **PATH** and other environment variables.

As with **install.sh**, if you're inside a Python venv, you need to deactivate it, so that **export.sh** can activate its own venv . Otherwise you will get an error that you can't create venv inside another venv.

```
# Do this in each new terminal.
cd circuitpython/ports/espressif

# If you are inside a Python venv, deactivate it.
deactivate
```

```
source esp-idf/export.sh
```

export.sh will suggest that you use **idf.py** to do a build after you run it. But we don't use **idf.py**. Instead, use `make`. For example:

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

Files Generated by a Build

`make BOARD=...` generates several files:

- **circuitpython-firmware.bin**
- **firmware.uf2** (not generated on boards that don't support the UF2 bootloader)
- **firmware.bin**

The **circuitpython-firmware.bin** is an intermediate file that does not include the bootloader and partition table. Ignore it: do not flash it to the chip. Flash **firmware.bin** instead.

Decoding a Crash Backtrace

If your Espressif build of CircuitPython crashes with a line that looks something like this, you can process this backtrace and get a symbolic backtrace.

```
Backtrace: 0x400e33fd:0x3ffb1090 0x40086755:0x3ffb10d0 0x40084025:0x3ffb1100
0x4008e75d:0x3ffb3b70 0x4008e7a5:0x3ffb3ba0 ...
```

Run this python script, substituting the `BOARD` name for `<board>` below. Then paste the `Backtrace: ...` line into the `?` prompt. The script will look in the `build-<board>` directory, and use the **.elf** file there to decode the hex numbers in the backtrace.

```
$ cd ports/espressif
$ python3 tools/decode_backtrace.py &lt;board&gt;
```

How to Add a New Board to CircuitPython

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