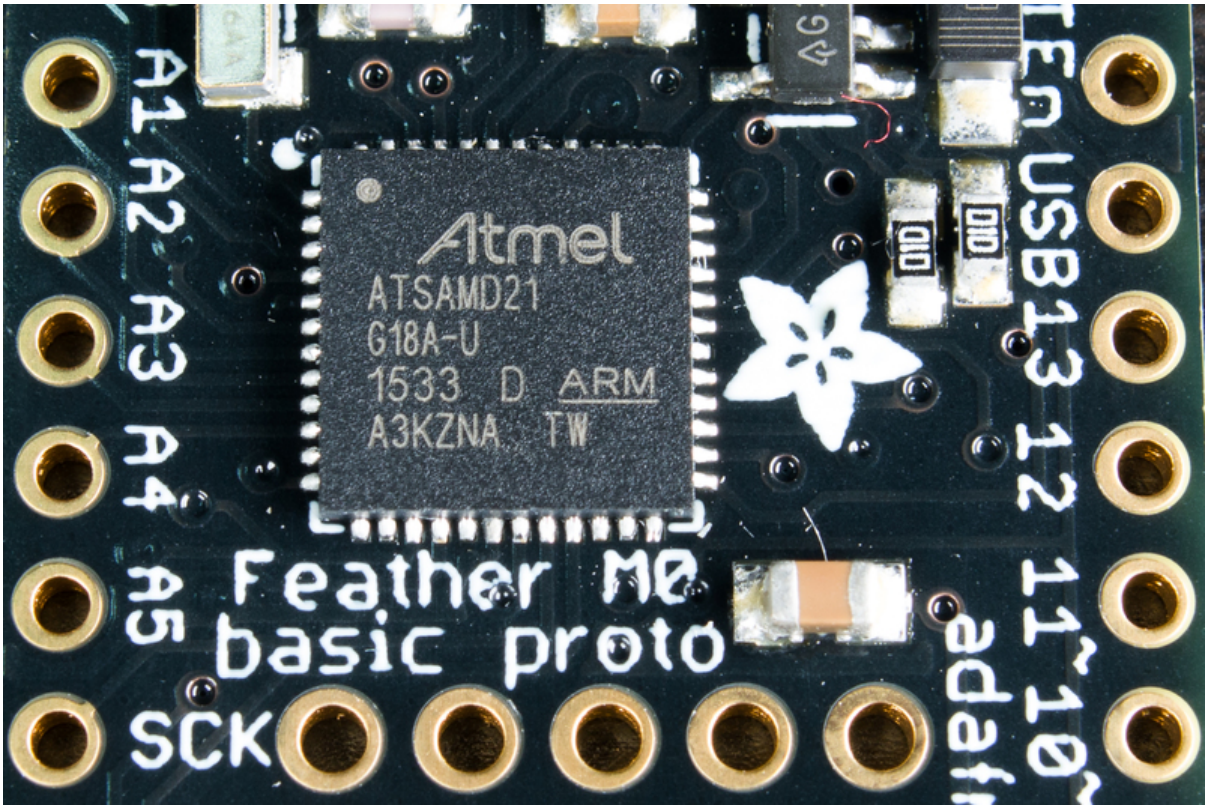
MicroPython for SAMD21

Created by Tony DiCola



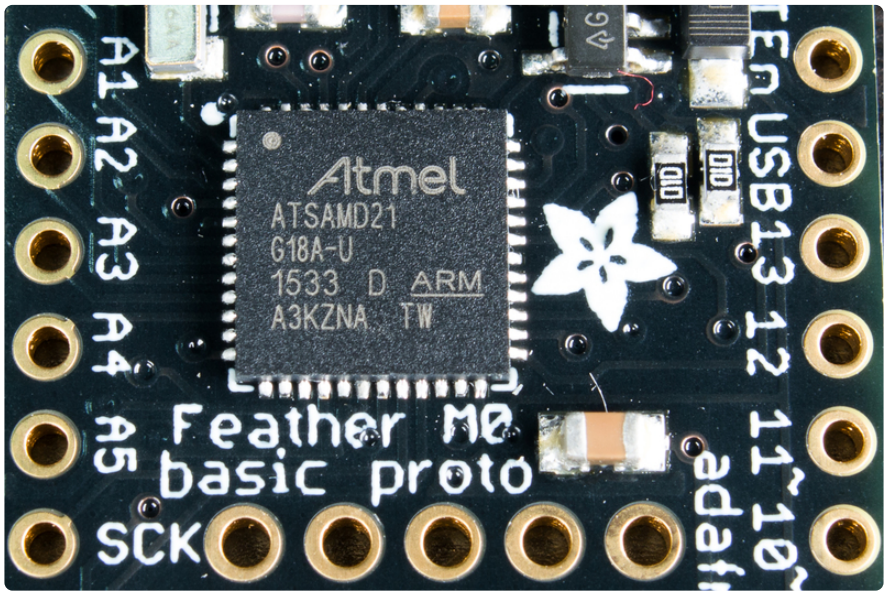
https://learn.adafruit.com/micropython-for-samd21

Last updated on 2024-03-08 02:29:50 PM EST

Table of Contents

Overview

This guide is no longer up to date - please check out https://learn.adafruit.com/welcome-to-circuitpython for the introduction guide to CircuitPython on ATSAMD21 boards (and more!)



MicroPython (https://adafru.it/pMa) is a fantastic environment for controlling hardware with the Python programming language (https://adafru.it/cFQ). Beginners can benefit from MicroPython's simple syntax and interactive REPL, while advanced users can explore mixing Python code with low-level hardware access. There are a few boards which support MicroPython today, such as the powerful pyboard (http://adafru.it/2390) or the WiFi-capable ESP8266 (http://adafru.it/2821). Now there's a new set of boards which can support MicroPython, Adafruit's Feather M0 and Arduino Zero boards which use the SAMD21 microprocessor!

Adafruit's SAMD21 MicroPython port (https://adafru.it/tB7) (now called CircuitPython) is a brand new port of MicroPython to the Atmel SAMD21 microprocessor. This processor is a 32-bit ARM Cortex-M0+ chip which means it's quite a bit more powerful than typical Arduino boards like the Uno or Leonardo. There's more RAM (32kb), flash memory (256kb), and processing power (48mhz) which enable these SAMD21 boards to run MicroPython. MicroPython on the SAMD21 has many of the same features as other boards, like an interactive USB serial REPL, USB mass storage for manipulating files, and all of the core MicroPython language support.

Be aware the SAMD21 MicroPython port is still in development! It is currently in an early 'preview' or beta state for advanced users to kick the tires and explore MicroPython on Feather M0 and similar boards. Not all MicroPython modules are

implemented yet, and there very likely will be bugs and other issues. Check out the quick start guide page (https://adafru.it/s1a) to see what modules are currently supported by the port.

If you do run into bugs or issues with the port be sure to raise them on the GitHub repository (https://adafru.it/tB7) (please search first to see if the issue is already known!). Pull requests and changes are appreciated too, but be sure to read the contributing (https://adafru.it/tB8) and code of conduct (https://adafru.it/tB9) first.

To follow this guide you will need an Atmel SAMD21-based board like one of the following:

- Adafruit Feather M0 Basic Proto **(http://adafru.it/2772)**
- Arduino Zero **(http://adafru.it/2843)** (or the discontinued Arduino M0 Pro (http://adafru.it/2417))
- Adafruit Feather M0 Adalogger **(http://adafru.it/2796)**
- Adafruit Feather M0 Bluefruit LE **(http://adafru.it/2995)** (note the Bluetooth radio isn't yet supported by MicroPython)
- Adafruit Feather M0 WiFi **(http://adafru.it/3010)** (note the WiFi radio isn't yet supported by MicroPython)
- Adafruit Feather M0 RFM69HCW Packet Radio **(http://adafru.it/3176)** (note the radio isn't yet supported by MicroPython)
- Adafruit Feather M0 RFM95 LoRa Radio **(http://adafru.it/3178)** (note the radio isn't yet supported by MicroPython)

If you aren't sure what to try, the Feather M0 Basic Proto (http://adafru.it/2772) is a great starting option.

You will also want to be familiar with MicroPython by reading the following guides (and other MicroPython guides in the learning system (https://adafru.it/qzD)):

- MicroPython Basics: What is MicroPython? (https://adafru.it/pXa)
- MicroPython Basics: Blink a LED (https://adafru.it/s1e)
- MicroPython Basics: Load Files and Run Code (https://adafru.it/s1f)
- MicroPython Basics: Loading Modules (https://adafru.it/rcv)

Load Firmware

This guide is no longer up to date - please check out https://learn.adafruit.com/welcome-to-circuitpython/installing-circuitpython for the introduction guide to installing CircuitPython on ATSAMD21 boards (and more!)

Download Firmware

To get started you'll need to download a release of the SAMD21 MicroPython firmware from its home on GitHub (https://adafru.it/tB7). Alternatively you can build the firmware from scratch, but if you just want to get started quickly use the prebuilt firmware from a GitHub release.

Go to the SAMD21 MicroPython releases page (https://adafru.it/tBa) and download the latest .bin file for your board. Currently the Arduino Zero and Feather M0 boards are supported and have prebuilt firmware images.

As of the time of this guide's writing the current SAMD21 MicroPython port version is: **v1.8.4-20161014**

Never use bossac or its Windows equivalent (BOSSA) 1.9.0 or higher without specifying an offset: 0x2000 for SAMD21, 0x4000 for SAMD51. Omitting the offset will damage the fuse settings on the board and can lead to a broken bootloader. https://learn.adafruit.com/welcome-to-circuitpython/non-uf2-installation#download-bossa-2978786-12

Download BOSSA

Once you have a firmware image you'll need to download a special version of the BOSSA tool that can load firmware on SAMD21 boards. This tool is actually used internally by the Arduino IDE when it programs these boards, however you can use it yourself to load custom firmware like the SAMD21 MicroPython port.

Be aware you **must** use this special **1.7.0 or higher version** of BOSSA to program SAMD21 boards! If you find an older version of BOSSA it won't work because it doesn't support the SAMD21 chip.

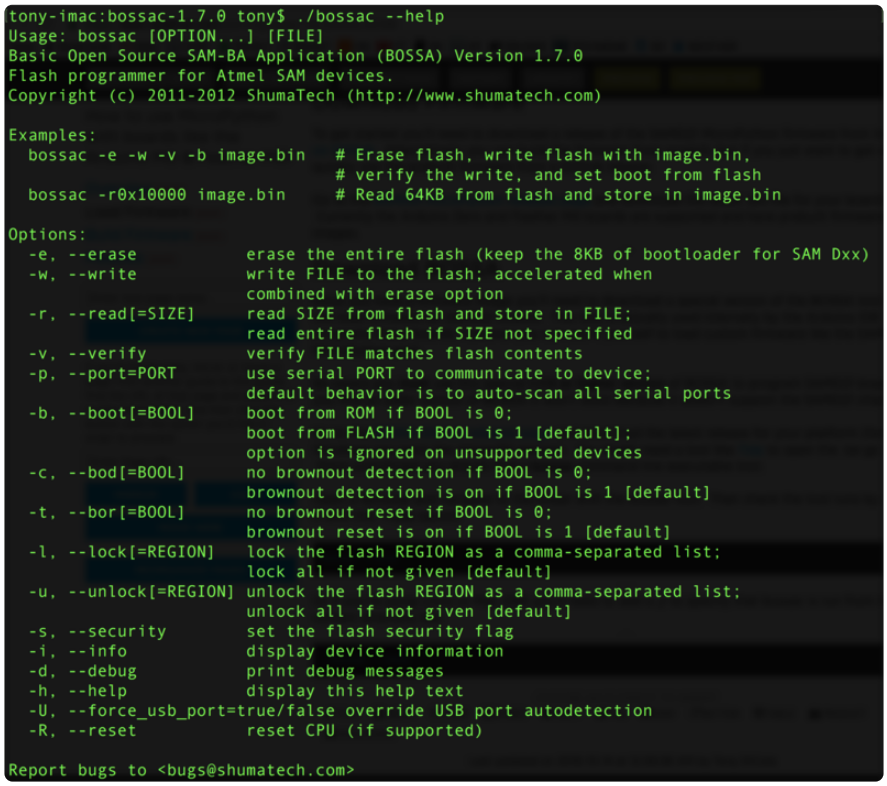
Go to the BOSSA GitHub releases page (https://adafru.it/s1B) and download the latest release for your platform (for Windows you want the mingw32 version). You might need a tool like 7zip (https://adafru.it/aZi) to open the .tar.gz archive. Once unarchived you should find a **bossac** command line executable tool.

Open a terminal and navigate to the folder with the bossac tool. Then check the tool runs by running it with the --help option:

```
bossac --help
```

Or if you're using Linux or Mac OSX you'll need to add a **./** to specify that bossac is run from the current directory like:

```
./bossac --help
```



Make sure you see BOSSA version 1.7.0 or higher! If you see a lower version then you accidentally downloaded an older version of the tool and it won't work to flash SAMD21 chips. Go back and grab the latest release from this BOSSA GitHub repository (https://adafru.it/s1B) as mentioned above.

Flash Board

Once you have the firmware and BOSSA tool you're ready to flash the firmware onto the board. Follow the steps below to flash the firmware for an Arduino Zero or Feather M0 board.

Feather M0

For a Feather M0 board (including all variants like Feather M0 proto, AdaLogger, Bluefruit, RFM69, etc.) you need to put the board in a special bootloader mode before flashing firmware. Plug in the board and 'double click' the reset button. You should see the pin #13 LED start to fade on and off to signal that the bootloader is running. Once you see the bootloader running skip down to the BOSSA flash command below.

Also be sure you've followed the learn guide for your Feather M0 board to setup the Arduino IDE and install any necessary drivers. In particular Windows will need drivers installed (https://adafru.it/s1C) before you can flash the Feather M0. Linux and Mac OSX should be usable without any extra driver install. Be sure you can use and program your Feather M0 with Arduino before diving into loading MicroPython--this will help you make sure the board is working as expected before complicating things with custom firmware.

Arduino Zero

For the Arduino Zero you don't need to do anything special to enter the bootloader. However make sure you've plugged in the board using the **native USB port** and **not** the **programming port**.

One special note, if you're using the Arduino M0 Pro from Arduino.org you'll need to replace its bootloader with the Arduino Zero bootloader so it can work with BOSSA. To do this install the Arduino/Genuino Zero board in the Arduino IDE board manager and then follow these steps to burn the Arduino Zero bootloader (https://adafru.it/s1D) (using the **programming** port on the board). Once you've loaded the Arduino Zero bootloader you should be able to use the M0 Pro with BOSSAC as described below.

BOSSA Flash Command

With your board plugged in and running the bootloader (for Feather M0 boards as mentioned above) you're ready to flash the SAMD21 MicroPython firmware onto the

board. Copy the firmware .bin file to the same directory as the bossac tool, then in a terminal navigate to that location and run the following command:

Flashing new MicroPython firmware will delete all files on the board's filesystem! Backup files you wish to keep and restore them after the flash.

```
bossac -e -w -v -R -p PORT_NAME firmware.bin
```

Remember on Linux & Mac OSX you need to add the ./ to the front of the bossac command!

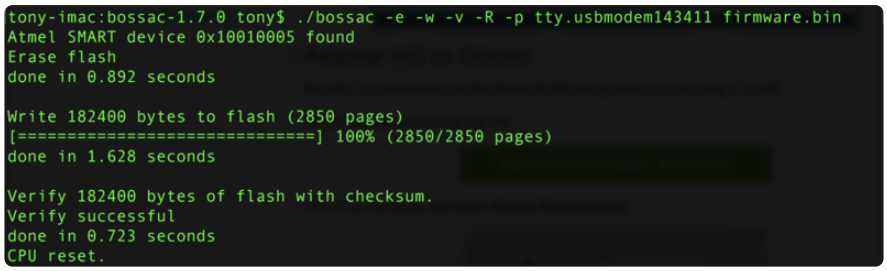
Change **PORT_NAME** to the name of the serial port for your board. On Windows this will be something like **COM4**, **COM5**, etc. (look in Device Manager for the list of all serial ports). On Mac OSX this will be something like **tty.usbmodem143411** (run **ls -l /dev/tty.usbmodem*** to list all the serial ports). On Linux this will be something like **ttyACM0**, **ttyACM1**, etc. (run **ls -l /dev/ttyACM*** to list all the ports). Note that on Mac OSX and Linux you don't specify the full /dev/tty... name of the device, instead you just specify the name after the /dev/ part of the path.

Also change **firmware.bin** to the name of the firmware .bin file you downloaded.

If you're using the Vagrant VM described later to build SAMD21 MicroPython firmware be sure the VM is halted and **not running** or else it might 'steal' the serial port from your host machine!

Finally note on Linux you might need to run as root with the sudo command if your user doesn't have access to the serial port. If you see an error like the device can't be found then try running with sudo in from of the ./bossac command.

After BOSSA loads the firmware you should see output similar to the following:



Access REPL

Once the firmware is loaded you're ready to access the serial REPL to run MicroPython code. Note that on Windows you might need to install this CDC serial driver from PJRC (https://adafru.it/s1E) to make the board accessible as a COM port. For Linux and Mac OSX no driver is necessary to install.

Using the serial REPL with SAMD21 MicroPython is just like using it with other MicroPython boards. See this MicroPython serial REPL guide (https://adafru.it/pMf) for details on using tools like screen and putty to access the REPL. SAMD21 MicroPython uses the same 115200 baud rate for the REPL as other MicroPython boards.

For example on Linux or Mac OSX you can use the screen command to open the REPL:

```
screen PORT_NAME 115200
```

Where **PORT_NAME** is the name of the serial port for the board (see the BOSSA flash command section above for how to list the serial ports).

Screen should open the port and if you press enter you should see the Python >>> REPL prompt. Try typing in a **print('Hello world!')** command and pressing enter to see MicroPython working on the SAMD21!

On Windows use the putty program as mentioned in the MicroPython serial REPL guide (https://adafru.it/pMf).

Woo hoo, that's all there is to loading SAMD21 MicroPython on a board! Once you see MicroPython running you can start using it just like other MicroPython boards. Tools like ampy can load files on the board's file system and even run scripts (https://adafru.it/s1f).

Be sure to read the quick start guide on supported modules (https://adafru.it/s1a) so you know what is currently supported by SAMD21 MicroPython!

Revert Back to Arduino Sketches

If you'd like to revert back to using your SAMD21 board with Arduino it's as easy as using the Arduino IDE to program the board again. For Feather M0 boards you might need to 'double click' the reset button to manually start the bootloader, then program the board from Arduino. For Arduino Zero boards the **programming port** and **Atmel EDBG** programmer type are guaranteed ways to always reprogram a board regardless of what it was previously running.

Should you ever want to go back to MicroPython just follow the steps on this page again!

Build Firmware

This guide is no longer up to date - please check out https://learn.adafruit.com/welcome-to-circuitpython for the introduction guide to CircuitPython on ATSAMD21 boards (and more!)

There's newer information about building CircuitPython in the Building CircuitPython Learn Guide: https://learn.adafruit.com/building-circuitpython

If you'd like to build the SAMD21 MicroPython firmware yourself you can use a special Vagrant-based virtual machine to simplify the build process. Note that you don't have to build the firmware yourself, you can download a prebuilt image and flash it as mentioned on the previous page. Building the firmware yourself is useful for modifying and working on the firmware code.

This isnt required if you're just using our release binaries! It's for people who want to build their very own releases.

For some background information skim the ESP8266 MicroPython firmware build guide (https://adafru.it/pMB) to see more details on Vagrant and VirtualBox usage. The SAMD21 MicroPython build VM is very similar to the ESP8266 build VM.

First install VirtualBox on your computer (https://adafru.it/cBK). If you already have VirtualBox installed it can't hurt to upgrade to the latest version. Virtualbox is an open source and no cost virtualization tool that can run a guest operating system 'inside' your host operating system.

Next install Vagrant on your computer (https://adafru.it/epq). Again if you already have Vagrant installed be sure to upgrade to the latest version. In most cases Vagrant needs the latest version of VirtualBox installed so if you install one tool be sure to install or upgrade the other too. Vagrant is an open source and no cost tool that simplifies using VirtualBox with a simple command line interface.

Now you'll need git installed on your computer (https://adafru.it/fBD), then open a command terminal (on Windows run the Git Bash terminal to make sure you can run git commands) and clone the SAMD21 MicroPython Vagrant configuration repository by running:

```
git clone https://github.com/adafruit/atmel-samd-micropython-vagrant.git
```

Then change into the atmel-samd-micropython-vagrant directory by running:

```
cd atmel-samd-micropython-vagrant
```

Inside this directory is a text file called **Vagrantfile**. This file is the configuration for the virtual machine and also is the location where all vagrant commands should be run. In the following sections you'll see how to start up, stop, and enter the virtual machine. Make sure you're inside this atmel-samd-micropython-vagrant directory with the Vagrantfile before running vagrant commands.

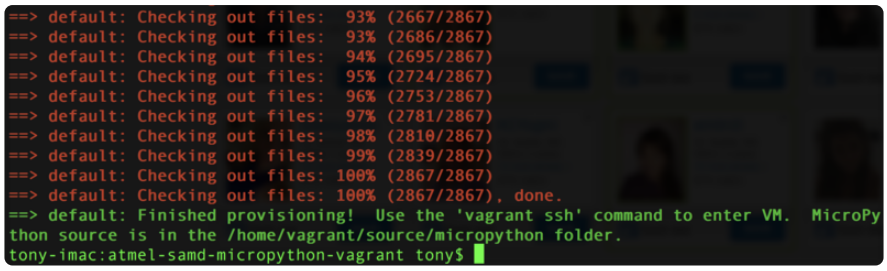
Start Virtual Machine

To start the virtual machine you'll use the vagrant up command:

```
vagrant up
```

The very first time this command runs it will provision the virtual machine by downloading a Linux operating system image and install tools & SAMD21 MicroPython firmware inside it. This process can take some time, perhaps 30 minutes or more, during the first run. Luckily the operating system image is saved on your computer so future starts of the VM will be very quick (a few seconds).

Once the virtual machine is running you should see output like the following that tells you to run **vagrant ssh** to enter the machine:

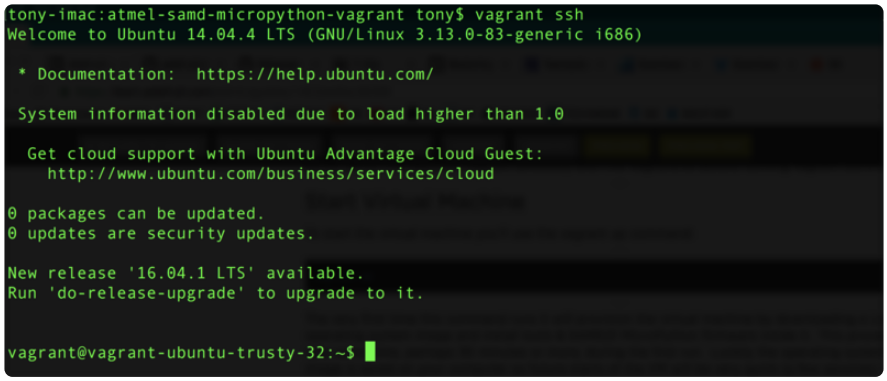


Enter Virtual Machine

With the virtual machine running use the vagrant ssh command to enter its command line terminal:

```
vagrant ssh
```

You should see an Ubuntu Linux prompt after a few moments:



Inside the VM you can build SAMD21 MicroPython firmware and even use BOSSA to flash it to a board. For the next sections you'll want to be inside the VM as shown above.

If you'd like to leave the VM run the **exit** command at the Linux terminal. This will exit from the VM's ssh session but note **the VM will continue to run in the background!** Skip down to the stop virtual machine section to see how to stop the VM from running.

If you need to enter the running VM again just use the **vagrant ssh** command again.

Build Firmware

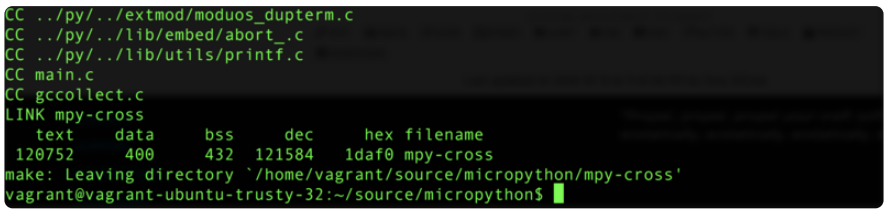
To build the SAMD21 MicroPython firmware you just need to go its source directory and use the make command. First change to the SAMD21 MicroPython source directory by running this command in the VM:

```
cd ~/source/circuitpython/
```

Next you'll need to build the **mpy-cross** tool by running this command:

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

You should see the **mpy-cross** tool successfully build and finish with output like:



You only need to build the mpy-cross tool once, then you can just focus on building the SAMD21 MicroPython firmware.

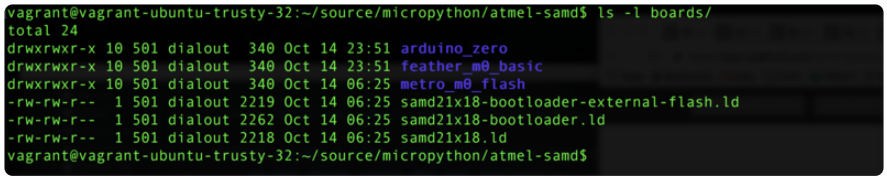
Now to build the SAMD21 MicroPython firmware first change to its directory (the **atmel-samd** directory) by running:

```
cd ~/source/circuitpython/atmel-samd
```

Before you can build the firmware you need to pick the right board that you're using. You can list the supported boards by listing all the subdirectories of the **boards** directory:

```
ls -l boards
```

Each of the subdirectory names (in blue) are supported boards, like **arduino_zero** or **feather_m0_basic**:



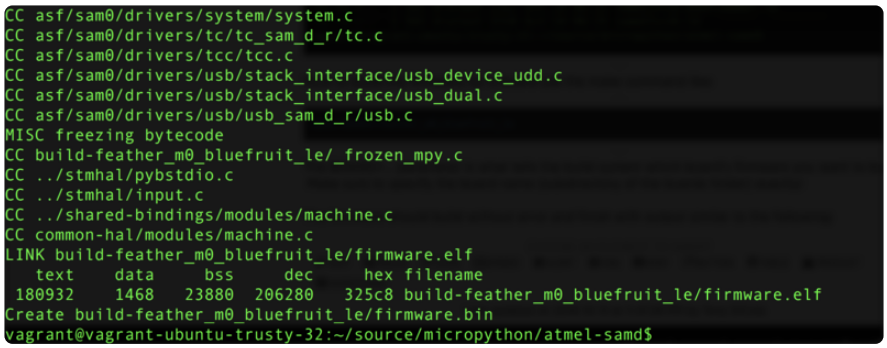
Note that the feather_m0_basic board can be used with any Feather M0 board, including the Feather M0 AdaLogger, Feather M0 Bluefruit, etc. Not all of the extra peripherals on these boards, like Bluetooth LE radios, etc., have MicroPython support however!

Now to build the firmware for a board use the make command like:

```
make BOARD=feather_m0_basic
```

The BOARD=... parameter is what tells the build system which board's firmware you want to build. Make sure to specify the board name (subdirectory of the boards folder) exactly!

The firmware should build without error and finish with output similar to the following:



If you do see the build fail with an error you might check the GitHub issues for the code (https://adafru.it/tBb) to see if there's a known issue with the build.

Once the firmware is built it will be placed in a **build-BOARD_NAME** directory, where **BOARD_NAME** Is the name of the board that was built. For example the **feather_m0_basic** board will have its firmware in the **build-feather_m0_basic** directory. Inside the build directory will be temporary files used during the compilation, and most importantly a **firmware.bin** file which contains the compiled firmware.

To load this firmware.bin on a board you can copy it out of the Vagrant VM and use BOSSA as described on the previous page (https://adafru.it/s2a). To copy out the file from the VM simply copy it to the **/vagrant** folder. This will place the file on your host PC (i.e. the machine running the VM) so that you can access it from BOSSA and other tools. For example run the following command to copy out the built Feather M0 firmware:

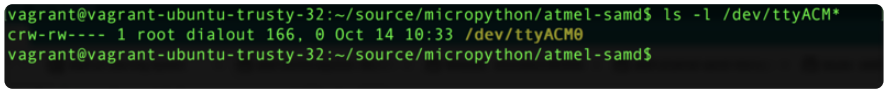
```
cp build-feather_m0_basic/firmware.bin /vagrant
```

Now exit the VM (using the **exit** command) and you'll find the **firmware.bin** file in the **vagrant** subdirectory (under the directory with the Vagrantfile--note this is slightly different than the default Vagrant behavior to copy files into the same directory as the Vagrantfile).

If you're using a Feather M0 board you can actually flash the firmware using BOSSA directly inside the VM. Back inside the VM and with the board connected to your host computer 'double click' the reset button to force the bootloader to run (pin #13 red LED starts fading up and down). Once a Feather M0 is running its bootloader the

Vagrant VM should 'capture' it and make it available to the VM instead of the host computer. You should see a **/dev/ttyACM0** device if you run the following command:

```
ls -l /dev/ttyACM*
```

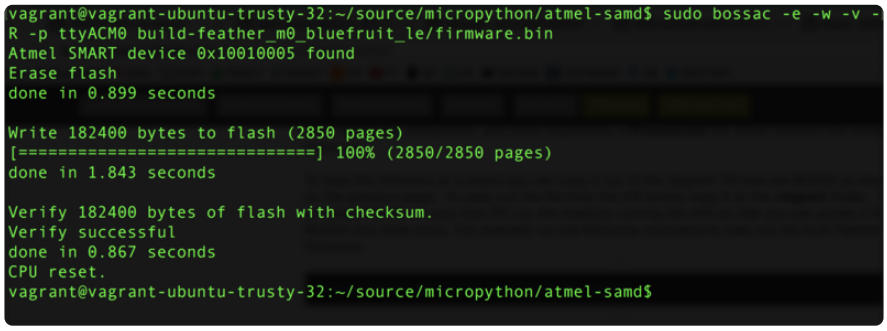


BOSSA is already built and installed in the path of the VM so it can be run from anywhere. To flash the newly built firmware simply run:

```
sudo bossac -e -w -v -R -p ttyACM0 build-feather_m0_basic/firmware.bin
```

Change the build-BOARD_NAME directory to the right name for the firmware you built.

You should see BOSSA flash the firmware successfully:



However if you run into trouble try copying the firmware out of the VM and using BOSSA from your host machine.

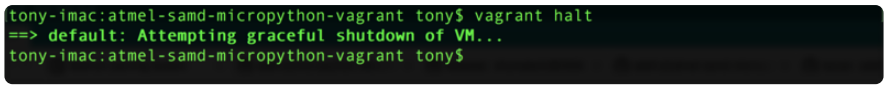
Remember when the VM is running it will 'capture' any Feather M0 it sees running the bootloader! This means your host PC won't see Feather M0 boards and you can't run BOSSA against them. Simply stop the VM (see section below) and try connecting the board again to prevent this capture.

Stop Virtual Machine

To shut down the virtual machine first run the exit command to leave it if you're still inside. Then from the same directory as the Vagrantfile run the following command:

```
vagrant halt
```

After a moment you should see the VM is shutdown:



To start the VM again once it's stopped just use the **vagrant up** command again.

Sync & Edit Source

If you'd like to edit the source code inside the VM you might want to enable a syncronization of source files between the VM and host machine. By syncing the source files you can use any text editor on your host machine to edit them instead of using a basic text editor inside the VM. This sync feature is disabled by default because it depends on what operating system you're using on the host machine (the default sync system Vagrant uses tends to cause slowdowns with many files like in the MicroPython source so a more optimized but platform-specific sync is necessary).

If you're using Linux or Mac OSX you can use NFS to sync source files between the machines. First read the Vagrant NFS documentation (https://adafru.it/s2b) and make sure you system supports NFS (you might need to install some depenedencies depending on your distribution). Exit and stop the VM, then edit the Vagrantfile and find this commented line:

```
# Optionally share the /home/vagrant/source directory with NFS (only for Mac
# and Linux host machines, for Windows try SMB below).
#config.vm.synced_folder "./source", "/home/vagrant/source", type: "nfs", create:
true
```

Uncomment the last line which defines a synced folder using NFS. The lines should look like:

```
# Optionally share the /home/vagrant/source directory with NFS (only for Mac
# and Linux host machines, for Windows try SMB below).
config.vm.synced_folder "./source", "/home/vagrant/source", type: "nfs", create:
true
```

Save the file and skip past the Windows instructions below to continue.

If you're using Windows you'll instead need to use SMB shares to sync source between the machines. First read the Vagrant SMB documentation (https://adafru.it/s2c) and note that you must run vagrant commands from inside a terminal that has administrator privileges. Exit and stop the VM, then edit the Vagrantfile and find this commented line:

```
# Optionally share the /home/vagrant/source directory with SMB (only for Windows
# host machines, for Mac OSX or Linux try NFS above).
```

```
#config.vm.synced_folder "./source", "/home/vagrant/source", type: "smb", create:
true
```

Uncomment the last line which defines a synced folder using SMB. The lines should look like:

```
# Optionally share the /home/vagrant/source directory with SMB (only for Windows
# host machines, for Mac OSX or Linux try NFS above).
config.vm.synced_folder "./source", "/home/vagrant/source", type: "smb", create:
true
```

Save the file to continue setting up the synchronization.

After editing the Vagrantfile start the VM again (**vagrant up**). During the startup you might need to enter your administrator password so Vagrant can configure the NFS shares on Mac OSX or Linux. On Windows remember you'll need to be running in an administrator terminal to use SMB shares! If you run into issues check the Vagrant NFS share (https://adafru.it/s2b) or Vagrant SMB share documentation (https://adafru.it/s2c).

Once the VM is running it will syncronize the SAMD21 MicroPython source code between the **/home/vagrant/source/** folder inside the VM and the **source** subdirectory under the Vagrantfile location on the host machine. You can use a text editor to modify the files in the source directory and they'll be changed inside the VM. Build the firmware in the VM and load it on your board to have a quick and easy development process!

Quick Start Guide

This guide is no longer up to date - please check out https://learn.adafruit.com/welcome-to-circuitpython for the introduction guide to CircuitPython on ATSAMD21 boards (and more!)

Be aware as of 11/2016 the master branch of Adafruit's MicroPython GitHub repository has a new API which is not directly compatible with other MicroPython libraries. The following info is just for reference of the older 1.8.5 release of Adafruit SAMD21 MicroPython firmware. See the new API docs here: https://circuitpython.readthedocs.io/en/latest/

This quick start guide acts as both examples and a reference for what modules are currently implemented in the SAMD21 MicroPython port. You can also find in-progress reference documentation for the SAMD21 MicroPython port here (https://adafru.it/tBc).

As of the time of this guide writing the current version of the SAMD21 MicroPython port is: **v1.8.4-20161014**

Contents:

Machine Module

The SAMD21 MicroPython port currently supports the following machine module features described below.

Note there are some small differences in how the machine module support is implemented compared to other boards. As the SAMD21 MicroPython firmware stabilizes and the machine module specification evolves these differences will hopefully grow smaller, but be aware you might need to slightly modify MicroPython code from other boards to run on the SAMD21 MicroPython port.

Digital I/O

You can use the machine.Pin class to perform digital I/O like reading buttons and blinking LEDs. See the MicroPython Digital I/O guide (https://adafru.it/qhF) for more details on usage.

For the SAMD21 MicroPython port pins are identified by a string, like 'D6' for digital pin 6 on the board.

For a **Feather M0** the full list of pin names are:

- **A0** - analog input 0
- **A1** - analog input 1
- **A2** - analog input 2
- **A3** - analog input 3
- **A4** - analog input 4
- **A5** - analog input 5
- **SCK** - SPI serial clock
- **MOSI** - SPI serial main output, secondary input
- **MISO** - SPI serial main input, secondary output
- **0RX** - digital pin 0 / UART receive **(note UART isn't supported yet)**
- **1TX** - digital pin 1 / UART transmit **(note UART isn't supported yet)**
- **SDA** - I2C data line
- **SCL** - I2C clock line
- **D5** - digital pin 5
- **D6** - digital pin 6
- **D9** - digital pin 9
- **D10** - digital pin 10
- **D11** - digital pin 11
- **D12** - digital pin 12
- **D13** - digital pin 13

For an **Arduino Zero** the full list of pin names are:

- **A0** - analog input 0
- **A1** - analog input 1
- **A2** - analog input 2
- **A3** - analog input 3
- **A4** - analog input 4
- **A5** - analog input 5
- **0RX** - digital pin 0 / UART receive **(note UART isn't supported yet)**
- **1TX** - digital pin 1 / UART transmit **(note UART isn't supported yet)**
- **D2** - digital pin 2 **(note on the Arduino M0 Pro this is swapped with digital pin 4!)**
- **D3** - digital pin 3
- **D4** - digital pin 4 **(note on the Arduino M0 Pro this is swapped with digital pin 2!)**
- **D5** - digital pin 5
- **D6** - digital pin 6

- **D7** - digital pin 7
- **D8** - digital pin 8
- **D9** - digital pin 9
- **D10** - digital pin 10
- **D11** - digital pin 11
- **D12** - digital pin 12
- **D13** - digital pin 13
- **SDA** - I2C data line
- **SCL** - I2C clock line

For example to blink a LED on pin #13 (the built-in LED for most boards) you could run the following code:

```
import machine
import time

led = machine.Pin('D13', machine.Pin.OUT)
while True:
    led.high()
    time.sleep(0.5)
    led.low()
    time.sleep(0.5)
```

Or to read a momentary push button connected to pin #6 and ground (using an internal pull-up resistor) you could run the following code:

```
import machine

button = machine.Pin('D6', machine.Pin.IN, machine.Pin.PULL_UP)
while True:
    if not button:
        print('Button pressed!')
```

Analog I/O

You can use analog inputs, PWM, and even a DAC output just like with using analog I/O on other MicroPython boards (https://adafru.it/s2d)! Remember the pin names are strings for the SAMD21 MicroPython port--see the digital I/O section above for all the pin names.

Analog to Digital Converter (ADC)

The SAMD21 includes a 12-bit ADC which can read up to 6 channels (labeled as A0-A5 on boards). For example to read analog input 0 you could run the following code:

```
import machine

a0 = machine.ADC(machine.Pin('A0'))
value = a0.read()
print('Analog input 0 reads a value of: {0}'.format(value))
```

Note that the ADC class needs to take in a machine.Pin instance and not just a string pin name!

Since the ADC has a 12-bit resolution you'll see values between 0 and 4095. A value of 0 means there's 0 volts on the analog input, and a value of 4095 means there's 3.3v or more on the input (be careful not to exceed 3.3v!). An inbetween value like 2047 means there's about half of 3.3v, or 1.6v, on the analog input.

Pulse-Width Modulation (PWM)

To use a PWM output to fade up pin #13 (most board's built in LED) you could run the following code:

```
import machine
import time

led_pwm = machine.PWM(machine.Pin('D13'))
while True:
    for i in range(256):
        led_pwm.duty(i)
        time.sleep(0.01)
```

Note that PWM outputs currently have a fixed frequency of around 730 hz. In the future configurable frequencies will be allowed (check this bug (https://adafru.it/s2e)) but for now the frequency is fixed.

Duty cycle can be configured to any 8-bit value from 0 (completely off) to 255 (completely on). A value of 127 for example would be a 50% duty cycle, or half lit LED if controlling the brightness of a LED.

Digital to Analog Converter

The SAMD21 includes a single channel 10-bit DAC (digital to analog converter) that can be used to produce a voltage between 0-3.3v (depending on the board's analog reference, most are 3.3V). An example of the DAC usage to alternate between 0v, 1.6v (half of 3.3v), and 3.3v is:

```
import machine
import time
```

```
dac = machine.DAC()
while True:
    dac.write(0)    # 0 volts output on A0
    time.sleep(1.0)
    dac.write(512)  # 1.6 volts output on A0
    time.sleep(1.0)
    dac.write(1023) # 3.3 volts output on A0
    time.sleep(1.0)
```

For the SAMD21's DAC it can **only** output voltage on the **A0** pin. No other pin can be used to receive the DAC output.

The DAC write function takes a 10-bit value, i.e. one between 0 and 1023. A value of 0 means 0v is output on the A0 pin, and 1023 means the maximum analog reference voltage (3.3v) is output on A0. A value inbetween produce a proportional voltage, like 512 will output about half of 3.3v (1.6v).

There's also a handy **write_mv** function on the DAC object which allows you to specify a voltage in millivolts between 0 and 3300 (3.3v). For example to set the DAC output to 2.5 volts you could run:

```
import machine

dac = machine.DAC()
dac.write_mv(2500)
```

I2C Devices

Communication with I2C devices is supported using a similar interface as I2C on other MicroPython boards. (https://adafru.it/s2f) However one important difference is that you must either call an **init** function before using I2C and a **deinit** function after using it, or use a context manager.

First an example of reading the MAX9808 I2C temperature sensor with init & deinit:

```
import machine

# Function to convert temperature register to degrees celsius.
def temp_c(data):
    value = data[0] &lt;&lt; 8 | data[1]
    temp = (value &amp; 0xFFF) / 16.0
    if value &amp; 0x1000:
        temp -= 256.0
    return temp

i2c = machine.I2C(machine.Pin('SCL'), machine.Pin('SDA'))
i2c.init()  # Initialize I2C interface.

print('Found I2C devices: {0}'.format(i2c.scan()))

address = 24
temp_reg = 5
```

```
data = i2c.readfrom_mem(address, temp_reg, 2)
print('Temperature: {0}C'.format(temp_c(data)))

i2c.deinit()  # Be sure to call deinit when done.
```

And the same example using the with statement and a context manager which guarantees the init & deinit functions are called appropriately:

```
import machine

# Function to convert temperature register to degress celsius.
def temp_c(data):
    value = data[0] &lt;&lt; 8 | data[1]
    temp = (value &amp; 0xFFF) / 16.0
    if value &amp; 0x1000:
        temp -= 256.0
    return temp

# With statement uses I2C interface as context manager for automatic init/deinit.
with machine.I2C(machine.Pin('SCL'), machine.Pin('SDA')) as i2c:
    print('Found I2C devices: {0}'.format(i2c.scan()))

    address = 24
    temp_reg = 5
    data = i2c.readfrom_mem(address, temp_reg, 2)
    print('Temperature: {0}C'.format(temp_c(data)))
```

The choice of init/deinit vs. context manager comes down to how you'd like to structure your code. For simplicity init & deinit are a good first choice, but if you need to guarantee the I2C interface is cleanly deinitialized (like on board soft reset) use a context manager.

Also note **only** the SCL and SDA pins as shown above can be used for the clock and data lines respectively. The SAMD21 MicroPython port uses a hardware I2C interface which is fixed to those pins. Also be aware you cannot take low level control of the I2C bus and send explicit start/stop bits or create custom transactions. This means some libraries like the SSD1306 OLED display won't work until they are updated to work with hardware I2C interfaces.

Other I2C interface functions are supported too as shown in the MicroPython I2C guide (https://adafru.it/s2f):

- **readfrom_mem(**address, register, number of bytes**)**
- **readfrom_mem_into(**address, register, buffer**)**
- **writeto_mem(**address, register, byte string**)**
- **readfrom(**address, number of bytes**)**
- **readfrom_into(**address, buffer**)**
- **writeto(**address, byte string**)**

SPI Devices

SPI support is currently in-progress. Check back with future releases to see when SPI is available!

NeoPixel Module

Basic RGB NeoPixel support is available in the **neopixel** module, just like with the ESP8266 MicroPython port. For example to use a string of 4 NeoPixels connected to pin D6 of a Feather M0:

```
import machine
import neopixel

# Create NeoPixel signal pin (connected to D6).
pixel_pin = machine.Pin('D6', machine.Pin.OUT)

# Create a strip of 4 pixels.
pixels = neopixel.NeoPixel(pixel_pin, 4)

# Clear all the pixels to black/off.
pixels.fill((0,0,0))

# Use write to light up the pixels with the previously set colors.
pixels.write()

# Set the first four pixels to red, green, blue, white:
pixels[0] = (255, 0, 0)
pixels[1] = (0, 255, 0)
pixels[2] = (0, 0, 255)
pixels[3] = (255, 255, 255)

# Again call write to light up the pixels.
pixels.write()
```

OS Module

The OS module provides core 'operating system' style functions for MicroPython, particularly to access the file system on the board. The SAMD21 MicroPython port currently implements the following OS module functions:

- **uname**
- **chdir**
- **getcwd**
- **listdir**
- **mkdir**
- **remove**
- **rename**

- **rmdir**
- **stat**
- **statvfs**
- **unlink**
- **sync**
- **sep**
- **mount**
- **umount**
- **mkfs**

See the pyboard OS module documentation (https://adafru.it/s2A) for details on the OS module function usage as the functions are the same across boards.

Note that the random number generation function urandom is **not** currently supported.

Time Module

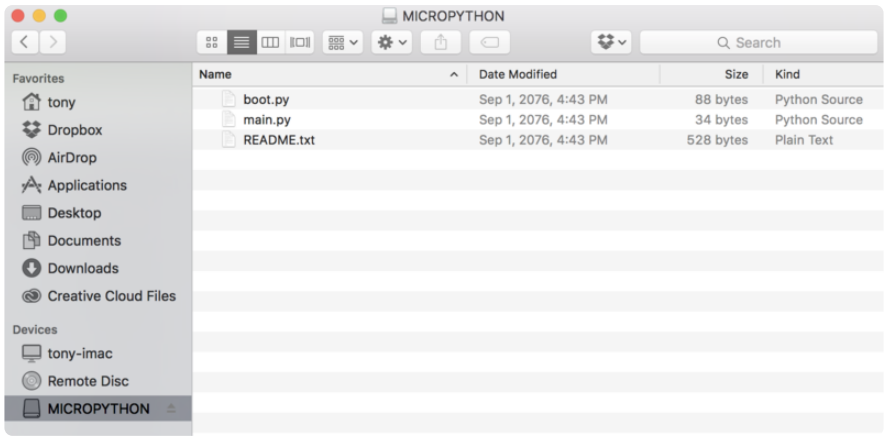
The SAMD21 MicroPython port time module currently implements the following basic sleep/delay and millisecond tick functions:

- **sleep**
- **sleep_ms**
- **sleep_us**
- **ticks_ms**
- **ticks_diff**

See the pyboard time module documentation (https://adafru.it/s2B) for details on the time module function usage.

USB Mass Storage

This guide is no longer up to date - please check out https://learn.adafruit.com/welcome-to-circuitpython/installing-circuitpython for the introduction guide to installing CircuitPython on ATSAMD21 boards (and more!)



One great feature of the SAMD21 MicroPython port is that it exposes its internal filesystem as a USB mass storage device. This means when you plug the board into a computer you should see a **MICROPYTHON** drive created, and inside you'll find all the files on the board's file system. You can edit boot.py, main.py, and even read or write other files.

For reference the SAMD21 MicroPython port currently reserves **64 kilobytes** of space total for the internal filesystem. This is great for storing source code, frozen modules, and other small assets

Don't write files to the internal filesystem with Python's open & write functions while you are also accessing files with USB mass storage! There is currently the potential to corrupt the entire filesystem.

Mac OSX File Copy Issues

On Mac OSX you might run into issues copying files with simple drag and drop or even terminal **cp** commands. The issue is that Mac OSX creates hidden files to store extended attributes, recycle bin files, etc. and those files can take up a significant amount of space--even filling the entire board's filesystem! Luckily you can work around this issue by following these steps

First install the latest firmware on your board and ensure its filesystem is completely erased and in a basic starting point.

Next find the volume name for your board. With the board plugged in run this command in a terminal to list all the volumes:

```
ls -l /Volumes
```

Look for a volume with a name like **MICROPYTHON** (the default for SAMD21 MicroPython). The full path to the volume is the **/Volumes/MICROPYTHON** path.

Now follow the steps from this question (https://adafru.it/u1c) to run these terminal commands that stop hidden files from being created on the board:

```
mdutil -i off /Volumes/MICROPYTHON
cd /Volumes/MICROPYTHON
rm -rf .{,_.}{fseventsd,Spotlight-V*,Trashes}
mkdir .fseventsd
touch .fseventsd/no_log .metadata_never_index .Trashes
cd -
```

Replace **/Volumes/MICROPYTHON** in the commands above with the full path to your board's volume if it's different.

Finally when you copy files or directories to the board you **must** use a special terminal command to do so. Unfortunately you **cannot** use drag and drop copy in Finder because it will still create hidden extended attribute files in some cases (for files downloaded from the internet, like Adafruit's modules).

To copy a file or folder use the **-X** option for the **cp** command. For example to copy a foo.mpy file to the board use a command like:

```
cp -X foo.mpy /Volumes/MICROPYTHON
```

Or to copy a folder and all of its child files/folders use a command like:

```
cp -rX folder_to_copy /Volumes/MICROPYTHON
```

Frozen Modules

The SAMD21 MicroPython port supports two ways of freezing modules to reduce their memory usage (https://adafru.it/ryA).

Internal Frozen Modules

The first way is by building a custom SAMD21 MicroPython firmware build which packs in .py files as frozen modules. The **modules** folder inside the **atmel-samd** folder of the source code is where you should place modules that will be frozen into the firmware. This process is just like freezing modules on the ESP8266 port (https://adafru.it/ryA) (however note there is no scripts folder for 'less frozen' modules, only the modules folder).

External Frozen Modules

The second way to use frozen modules is by converting a Python source file into a .mpy frozen module file. You can copy .mpy files onto the board's file system and import them just like importing Python source files and modules (https://adafru.it/s2C). However .mpy files will consume less RAM compared to raw Python .py source files.

To generate a .mpy file it's easiest to use the Vagrant build VM described earlier in this guide (https://adafru.it/s2D). Start the VM and enter its SSH terminal. Then run the following command to change to the .mpy compiler directory:

```
cd ~/source/circuitpython/mpy-cross
```

Make sure you've built the mpy-cross tool already as the previous build firmware instructions mentioned. If you skipped that step then first run the **make** command in the directory to build the tool.

Once the tool is built converting a .py file to a .mpy file is as easy as running it with the tool. First copy your Python source .py file into the VM (remember any file in the **vagrant** subdirectory on the host machine will be copied into the **/vagrant** folder inside the VM).

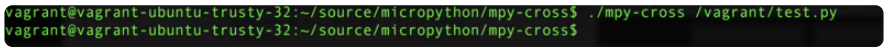
For example if you have **test.py** file with a few functions:

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

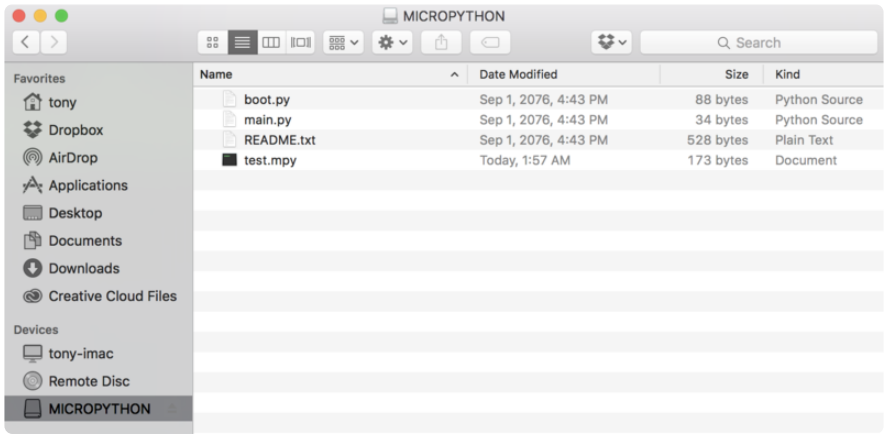
Copy it into the **vagrant** folder on the host PC, then in the mpy-cross directory in the VM run the following command to covert **test.py** into **test.mpy**:

```
./mpy-cross /vagrant/test.py
```



After the tool runs you should see a **test.mpy** file in the **vagrant** folder on the host PC, right next to the **test.py** file.

Now copy **test.mpy** onto the SAMD21 MicroPython board's filesystem. Remember you can just drag and drop with USB mass storage!



Note: To be sure the file is accessible to the MicroPython board, after copying it over unmount the USB drive and reset the board by pressing the reset button.

Open a serial REPL on the board and run the following code to confirm you can import the test.mpy file and call its functions:

```
import test
test.add(1, 1)
test.subtract(1, 1)
```

That's all there is to using .mpy files as external frozen modules!