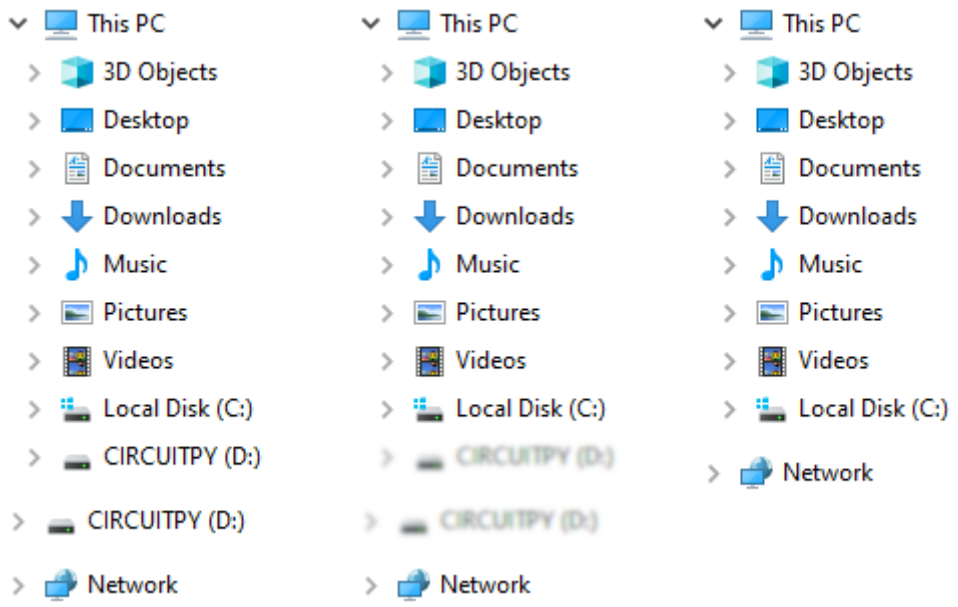
Customizing USB Devices in CircuitPython

Created by Dan Halbert



https://learn.adafruit.com/customizing-usb-devices-in-circuitpython

Last updated on 2024-11-22 03:06:34 PM EST

Table of Contents

Overview

Standard CircuitPython USB Devices

When you plug a CircuitPython board into a host computer, it shows up as several USB devices. Normally, you see:

- The **CIRCUITPY** drive, which is a USB "Mass Storage" (MSC) device.
- A serial connection to the REPL, which shows up as a **COM** port on Windows, a **/dev/tty** device on Linux, or a **/dev/cu** device on MacOS.
- MIDI in and out streams, which show up as a kind of audio device.
- A mouse, keyboard, etc., all of which are different kinds of "Human Interface Devices" (HID). The HID devices are lumped together in a single "composite" device.

Can I Hide Some Devices?

It's great that CircuitPython provides all these USB capabilities, **but sometimes you don't want to see all of them**. For instance, you might build a volume control, macro keypad, or your own fancy keyboard that you want to leave plugged in all the time. You wouldn't want its **CIRCUITPY** and serial connection to be visible. If you plugged in another CircuitPython board, you couldn't be sure which was your permanent accessory and which was the new board.

So CircuitPython allows you to choose which devices are visible at run time. You add code to the **boot.py** file which will specifies which USB devices you want to enable or disable. You can make CIRCUITPY disappear (https://adafru.it/SC9) or hide the standard REPL serial connection (https://adafru.it/SC9), for instance.

Add a Second Serial Port

In addition to just turning the standard devices on and off, you can also enable a second serial port (https://adafru.it/SC9). A second **COM** port or **/dev/tty** or **/dev/cu** device will appear. It is not connected to the REPL, so you can use it for unimpeded communication back and forth with the host computer. You can send and receive binary data, and not worry about having to escape ctrl-C characters or seeing print statements or errors in the data you read.

Define Custom HID Devices

Finally, you can also create new HID devices (https://adafru.it/VaJ), such as specialized game controllers, digitizers, custom mice, and so forth. You'll need to

understand how to make HID report descriptors, but you can often copy existing ones.

Change USB Identification

The overall USB device reports manufacturer and device names when it is plugged in. You can change the default values if you wish (https://adafru.it/18yF). A particular USB interface (MIDI, CDC, etc.) also reports names, but as of this writing, it is not possible to change these without building a custom version of CircuitPython.

Try it!

Read the rest of the guide to find out how to take control of CircuitPython USB!

CIRCUITPY, MIDI, and Serial

Here are the full details for how to enable and disable the **CIRCUITPY**, MIDI, and serial USB devices. HID devices are more complicated, and we'll cover them on their own page.

USB Setup and Configuration is Done in **boot.py**

All **the code examples on this page must be put in boot.py**. If you try to use them in **code.py,** you'll get an error, because by the time **code.py** runs, the USB devices are already set up (https://adafru.it/SIF).

Hard Reset Required After Changing **boot.py**

boot.py runs only after a hard reset. So if you change **boot.py**, you'll need to reset the board to have it re-run. Just editing **boot.py** or typing ctrl-D in the REPL will not cause it to re-run again. **Make sure your changes are** completely written out **(https://adafru.it/BIN) before you reset, to avoid confusion and filesystem corruption.**

> All the code examples in the page go in **boot.py**. **boot.py** only runs after a hard reset. So reset the board or power-cycle the board after you've changed **boot.py** and waited for your changes to be written to the board.

CIRCUITPY Mass Storage Device

The **CIRCUITPY** drive is normally visible on the host computer. To disable it showing up as a USB device, you can disable the drive in **boot.py**, as shown below. But note that if you add only the lines below, you'll lock yourself out (see below (https://

adafru.it/SC9)) from editing files! So don't put just this in your **boot.py** without realizing the consequences.

```
# Don't do just this! You'll lock yourself out from editing files.

import storage

storage.disable_usb_drive()
```

Note that disabling the USB device does not make the drive not work. It's still available for use by your program. In CircuitPython 9.0.0 and later, **CIRCUITPY** also becomes read-write to your program if it is not visible over USB. In earlier versions of CIrcuitPython, if you need to write to **CIRCUITPY** in your program, you need to use `storage.remount("/", readonly=False)` to remount it as read/write. See this guide page (https://adafru.it/DIE) for more details.

There is also a `storage.enable_usb_drive()` function, but you normally don't need to use it, unless your build has **CIRCUITPY** disabled by default or you want to re-enable it after disabling it in **boot.py**.

```
import storage

storage.disable_usb_drive()
storage.enable_usb_drive()   # Changed my mind :)
```

MIDI

The USB MIDI device is enabled by default on most boards. To disable MIDI, do this in boot.py:

```
import usb_midi

usb_midi.disable()
```

> USB MIDI is disabled by default on some boards, but can be enabled if another USB device is disabled.

Some microcontrollers, such as the STM32F4, ESP32-S2, and ESP32-S3 do not provide enough USB endpoints (details here (https://adafru.it/Sma)) to allow MIDI to be enabled all the time. On those boards, MIDI is disabled by default, but is available. If you want to enable it, you'll need to disable some other USB device to free up a pair of endpoints to accommodate MIDI. For example, you could do this:

```
import usb_hid, usb_midi

# On some boards, we need to give up HID to accomodate MIDI.
usb_hid.disable()
usb_midi.enable()
```

USB Serial: Console (REPL) and Data

CircuitPython normally provides a USB serial device which lets you talk to the CircuitPython console (https://adafru.it/Bec), where you can use the Python REPL (https://adafru.it/Awz). On Windows, this device shows up as a numbered **COM** port, such as **COM5**. On Linux, it shows up as **/dev/tty** device, often **/dev/ttyACM0**. On MacOS, it shows up with a name starting with **/dev/cu**, such as **/dev/cu.usbmodem14301**.

The serial device is called a CDC device, which stands for "Communications Device Class". The CircuitPython module that controls serial devices is called `usb_cdc`.

CircuitPython can also optionally provide a second serial device, which is not connected to the console. It's called the `data` serial device. You can send and receive arbitrary binary data on this device, so it's very useful if you want to exchange data or commands without worrying about the REPL interfering or a ctrl-C character stopping your program. The second serial channel will appear as a second **COM** port or **/dev/tty** or **/dev/cu** device, with a different name.

For details about how to use the second serial device, see the `usb_cdc` documentation (https://adafru.it/Smb).

Note that some microcontrollers have hardware limitations which can make using the second serial device awkward or impossible. See this page (https://adafru.it/18za) for more information.

You can enable and disable the `console` device and the `data` device independently. The `console` device is enabled by default, but the `data` device is not.

```
import usb_cdc

usb_cdc.disable()   # Disable both serial devices.

usb_cdc.enable(console=True, data=False)   # Enable just console
                                           # (the default setting)

usb_cdc.enable(console=True, data=True)    # Enable console and data

usb_cdc.enable(console=False, data=False)  # Disable both
                                           # Same as usb_cdc.disable()
```

Which Serial Port on the Host?

Since turning on the `data` device presents another serial port to the host, you want to be able to determine which serial ports correspond to which CircuitPython devices. You can use the Adafruit_Board_Toolkit (https://adafru.it/UBU) Python library, which is available for installation via `pip3 install adafruit-board-toolkit`.

You can list the available `usb_cdc.data` serial ports using `adafruit_board_toolkit.circuitpython_serial.data_comports()`. Similarly, to get a list of the `usb_cdc.console` (REPL) serial ports, use `adafruit_board_toolkit.circuitpython.repl_comports()`. Full documentation is here (https://adafru.it/UC2).

Don't Lock Yourself Out!

If you turn off both **CIRCUITPY** and the REPL in **boot.py,** and don't provide a way to turn them back on, you will lock yourself out of your board: you won't have any way to edit files anymore. You can recover by forcing the board to start in safe mode, which skips running **boot.py**. But it's easier and better to provide yourself an out: for instance, allow button push to skip disabling the devices.

This kind of code will lock you out:

```
import storage, usb_cdc

# DON'T DO THIS!
storage.disable_usb_drive()
usb_cdc.disable()
```

The code below is safer. It skips turning off both devices if a button is pushed. The first vesion of the code is for buttons that are grounded when pressed.

```
# This example is for the MacroPad,
# or any board with buttons that are connected to ground when pressed.

import storage
import board, digitalio

# On the Macropad, pressing a key grounds it. You need to set a pull-up.
# If not pressed, the key will be at +V (due to the pull-up).
button = digitalio.DigitalInOut(board.KEY12)
button.pull = digitalio.Pull.UP

# Disable devices only if button is not pressed.
if button.value:
   storage.disable_usb_drive()
```

The second version is for buttons that are connected to +V when pressed.

```
# This example is for a Circuit Playground,
# or any board with buttons that are connected to +V when pressed.
```

```
import storage, usb_cdc
import board, digitalio

# On the Circuit Playground, pressing an on-board button
# connects the button to +V.
button = digitalio.DigitalInOut(board.BUTTON_A)
button.pull = digitalio.Pull.DOWN

# Disable devices only if button is not pressed.
if not button.value:
    storage.disable_usb_drive()
    usb_cdc.disable()
```

HID Devices

HID stands for "Human Interface Device". Keyboards, mice, digitizer tablets, joysticks, and game controllers are all examples of HID devices.

Standard HID Devices

CircuitPython provides three HID devices by default. They are defined in `usb_hid.Devices` (https://adafru.it/SmB):

- `KEYBOARD` - A standard keyboard, including five (virtual) LED indicators.
- `MOUSE` - A standard mouse supporting five buttons and a mouse wheel.
- `CONSUMER_CONTROL` - A USB Consumer Control device: multimedia controls, browser shortcut keys, etc.

Using one of these devices does not preclude using another. Your program can use any and all at once.

Consumer Control

You may not have heard of Consumer Control devices, but you probably have one on your desk. Your keyboard may have volume control, play, and pause keys, and perhaps also keys to do browser operations, open a calculator, etc. Those keys are not actually regular keyboard device keys. Instead, key presses for those keys are sent to the host computer via a Consumer Control device, not the keyboard device.

For instance, on the keyboard below, the marked functions are sent via Consumer Control keys. Those physical keys send both regular keyboard presses and Consumer Control presses. For example, you can send the regular keyboard keycode **F4**, or the Consumer Control code **MUTE** depending on whether you press the **Fn** key or not.

Consumer Control codes are defined in this USB standards document (page 125) (https://adafru.it/1a3d).

Choosing HID Devices

You can choose which HID devices CircuitPython provides using code like this in **boot.py**. Normally you don't need to disable a device if you aren't using it, but you can do so if its presence is a problem. For instance, you might have your laptop set up to disable its touchpad if a mouse is plugged in. In that case, if you plugged in a CircuitPython board, the touchpad would stop working. But if you disable `usb_hid.Device.MOUSE`, assuming you don't need it, then that wouldn't happen.

```
import usb_hid

# These are the default devices, so you don't need to write
# this explicitly if the default is what you want.
usb_hid.enable(
    (usb_hid.Device.KEYBOARD,
     usb_hid.Device.MOUSE,
     USB_hid.Device.CONSUMER_CONTROL)
)

usb_hid.enable((usb_hid.Device.KEYBOARD,))   # Enable just KEYBOARD.

usb_hid.disable()       # Disable all HID devices.

usb_hid.enable(())      # Enabling no devices is another way to disable all the
devices.
```

Note that `usb_hid.enable()` always takes a tuple of devices, even if there is just one device, or zero devices.

Advanced Topics

Composite HID Devices

Multiple HID devices can be grouped together into a single composite HID device, which contains all the devices at once. They share a single endpoint pair (see here (https://adafru.it/Sma) for details), and each device uses a distinct report ID to distinguish it from the other devices in the composite device. In the code above, all the devices in the tuple are in a single composite device. If there is only one device listed, it does not need a separate report ID, and will not use one.

Right now CircuitPython only allows a single HID composite device, but this may change in the future.

Custom HID Devices

Besides the devices listed above, you can also define custom HID devices. You need to supply an HID report descriptor, which is a binary string of bytes that defines the kind of device and the details of the reports that it sends and receives. For instance, a mouse report will contain data describing which buttons that are currently pushed, how far the mouse has moved in X and Y directions, and how much the scroll wheel has been turned. A keyboard report will report which regular keys are pressed and which modifier keys (Shift, Ctrl, etc.) are pressed.

Writing your own HID report descriptors from scratch requires a lot of detailed knowledge, and is beyond the scope of this guide. However, you can often get the report descriptor for an existing HID device you want to emulate, and just use it as is, or modify it slightly for your purposes. There are many tutorials about HID report descriptors available, such as this one (https://adafru.it/SmE). And here's an online tool (https://adafru.it/SmF) that can decipher existing report descriptors.

You will also need to write a CircuitPython driver to handle your new device. There are examples in the adafruit_hid (https://adafru.it/xid) library.

As an example of what is possible, below is some code that defines a particular gamepad controller HID device and adds to the standard set of HID devices. **The sample report descriptor given here may not work with your operating system. You must understand how report descriptors are defined to make sure it suits your needs.**

```
import usb_hid

# This is only one example of a gamepad report descriptor,
# and may not suit your needs.
GAMEPAD_REPORT_DESCRIPTOR = bytes((
    0x05, 0x01,  # Usage Page (Generic Desktop Ctrls)
    0x09, 0x05,  # Usage (Game Pad)
    0xA1, 0x01,  # Collection (Application)
    0x85, 0x04,  #   Report ID (4)
    0x05, 0x09,  #   Usage Page (Button)
    0x19, 0x01,  #   Usage Minimum (Button 1)
    0x29, 0x10,  #   Usage Maximum (Button 16)
    0x15, 0x00,  #   Logical Minimum (0)
    0x25, 0x01,  #   Logical Maximum (1)
    0x75, 0x01,  #   Report Size (1)
    0x95, 0x10,  #   Report Count (16)
    0x81, 0x02,  #   Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null
Position)
    0x05, 0x01,  #   Usage Page (Generic Desktop Ctrls)
    0x15, 0x81,  #   Logical Minimum (-127)
    0x25, 0x7F,  #   Logical Maximum (127)
    0x09, 0x30,  #   Usage (X)
    0x09, 0x31,  #   Usage (Y)
    0x09, 0x32,  #   Usage (Z)
```

```
    0x09, 0x35,  #   Usage (Rz)
    0x75, 0x08,  #   Report Size (8)
    0x95, 0x04,  #   Report Count (4)
    0x81, 0x02,  #   Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null
Position)
    0xC0,        # End Collection
))

gamepad = usb_hid.Device(
    report_descriptor=GAMEPAD_REPORT_DESCRIPTOR,
    usage_page=0x01,           # Generic Desktop Control
    usage=0x05,                # Gamepad
    report_ids=(4,),           # Descriptor uses report ID 4.
    in_report_lengths=(6,),    # This gamepad sends 6 bytes in its report.
    out_report_lengths=(0,),   # It does not receive any reports.
)

usb_hid.enable(
    (usb_hid.Device.KEYBOARD,
     usb_hid.Device.MOUSE,
     usb_hid.Device.CONSUMER_CONTROL,
     gamepad)
)
```

Boot Keyboard and Mouse

The keyboard and mouse provided by CircuitPython can be marked as "boot" devices. This is a special feature of USB HID devices, used when you need to talk a computer when it's booting or to its BIOS. The report descriptor for a boot keyboard or mouse is standard. Any descriptor you supply is ignored if the device is used in boot mode.

See the `usb_hid` documentation (https://adafru.it/SmB) for details on boot devices. There are issues (https://adafru.it/YsD) with boot devices not working properly on some host computers, particularly Macs, and they are still under investigation.

Cleaning up Windows HID Devices

If you are developing a new HID device on Windows, and change the report descriptor in the process of development, you may find that the device does not seem to work properly. Windows remembers the old HID information, and can have trouble with a changed report descriptor. To fix this, you can remove the previously installed device, with the board not plugged in. You can do this from the Windows Device Manager, but it's much easier to use Uwe Sieber's Device Cleanup Tool (https://adafru.it/RWd). For more details see this section on the Troubleshooting page (https://adafru.it/Den) in the Welcome to CircuitPython guide.

USB Setup Timing

We mentioned previously that you have to set up USB devices in **boot.py,** not **code.py**.

boot.py runs before CircuitPython connects to the host computer via USB. When **code.py** runs, it's too late to change the USB devices. Here is what happens, step by step:

(1)
The board does a hard restart. This happens when you plug the board in, press the reset button, or do `microcontroller.reset()` in your program.

(2)
boot.py runs, if it exists. The board does not yet present any USB devices to the host computer. You set up USB devices in **boot.py**. Anything you print or any errors reported will go to **boot_out.txt**, since there is no console connection yet.

(3)
After **boot.py** has run, CircuitPython creates the data needed to tell the host about all the USB devices. This binary data is packaged into a number of USB descriptors.

(4)
CircuitPython tells the host USB is ready.

(5)
The host enumerates all the USB devices by asking for and receiving the descriptors, and setting up connections to the devices. At the same time, **code.py** starts to run. If there is no c**ode.py**, CircuitPython just starts the REPL.

How Errors Are Reported

Various things can go wrong in the steps above. For instance, if you make a programming error like a typo in **boot.py**, the error will show up in the **boot_out.txt** file in **CIRCUITPY**. They will not be printed on the console, since there is no USB connection yet.

But your code could also try to create too many USB devices (https://adafru.it/Sma). That error is not detected until step 3. Instead, CircuitPython will go into safe mode, and will not run **code.py**. If you go into the REPL, you will see CircuitPython reporting the reason it is in safe mode, such as "**USB devices need more endpoints than are available".**

How Many USB Devices Can I Have?

The number of USB devices you can have active at once is limited. Here we'll explain the limitations.

CircuitPython Limitations

You can specify no more than eight HID devices in total. In addition, you can only define a single composite HID device (https://adafru.it/Snd), though you may be able to define more in the future.

Hardware Limitations

Microcontrollers provide a limited number of USB endpoints. This is a hardware limitation. An endpoint is a single low-level USB communications channel. Typically endpoints are paired. Each endpoint pair has an **IN** endpoint and **OUT** endpoint. The naming is from the point of view of the host: **IN** means sending data from device to the host; **OUT** means sending data from the host to the device.

Endpoint pairs are numbered starting at 0. Endpoint pair 0 is always reserved for USB setup and control, so we can't use it for regular devices.

Each separate USB device needs to use one or more endpoint pairs. Here are the endpoint requirements for the devices we provide:

- **CIRCUITPY (MSC):** 1 IN/OUT endpoint pair.
- **MIDI:** 1 IN/OUT pair.
- **CDC:** 1 IN endpoint for control, and 1 IN/OUT pair for data, for a total of 2 pairs, for each CDC device. If you enable both `console` and `data` `usb_cdc` devices in CircuitPython, 2+2=4 pairs are needed in total.
- HID: 1 IN/OUT pair for each composite device. All the devices in the composite share the single endpoint pair.

So if all these devices are enabled, including both `usb_cdc` devices, you'll need 1+1+2+2+1=7 endpoint pairs.

The SAMD21, SAMD51, nRF52840, and RP2040 microcontrollers all provide 8 endpoint pairs each. since pair 0 is reserved, 7 pairs are available, and so enabling all the devices above just fits.

However, other microcontrollers provide fewer than 8 pairs:

- STM32F4 chips typically provide only 3 pairs, not counting pair 0. That means only **CIRCUITPY** and one CDC device will fit. If you want HID or MIDI on an STM32F4, you'll need to turn off **CIRCUITPY** or CDC.
- ESP32-S2 and ESP32-S3 effectively provide only 4 pairs, not counting pair 0. (There are 6 pairs, but the hardware allows only 4 IN endpoints active at a time, not including pair 0.) So if you wanted both CDC `console` and `data`, you would have to turn everything else off, including **CIRCUITPY**.

- Spresense provides 6 pairs but assigns its endpoints at build-time, so you can't turn on MIDI or an extra CDC device.

Using Too Many Devices

If your **boot.py** settings turn on too many devices, CircuitPython will go into safe mode after running **boot.py,** and will not run **code.py**. If you go into the REPL, you will see CircuitPython reporting the reason it is in safe mode, such as "**USB devices need more endpoints than are available".**