NeXT Computer Keyboard to USB HID with CircuitPython

Created by Jeff Epler

https://learn.adafruit.com/next-computer-keyboard-to-usb-hid-with-circuitpython

Last updated on 2024-06-03 03:44:06 PM EDT

Table of Contents

Overview 3

- Jeff's Verdict
- Parts

Research 5

Research II: Mouse 7

Install CircuitPython 8

- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

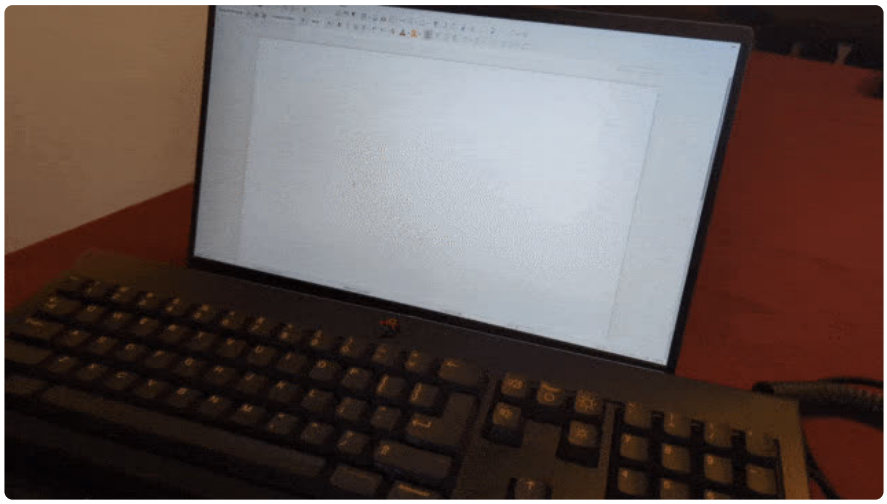
Coding the Keyboard 12

- Text Editor
- Download the Project Bundle
- How it works

Wiring the adapter & printing the enclosure 21

- Wiring the adapter
- Printing the enclosure

Overview



About ten years before this guide was written, Ladyada wrote a fairly early guide for the Adafruit Learning System: "USB NeXT Keyboard with an Arduino Micro. (https://adafru.it/18he)" The NeXT Computer and its keyboard remain iconic, and on the tenth anniversary of that older guide—more than 2000 guides later—I couldn't resist the challenge of updating the code from Arduino to CircuitPython.

The software in this guide is only for the original NeXT keyboard with 5-pin connector; the mouse is not supported. Later NeXT keyboards that use a 4-pin connector use a different protocol.

The techniques in this guide may also be helpful in converting other classic keyboards that use a clocked serial protocol. The RP2040's "pio" peripheral is perfect for this kind of low level I/O task, making it a breeze to meet the microsecond timing requirements while still writing the bulk of the code in high-level Python.

Since there are just a few pins used, the Adafruit QT Py RP2040 makes a solid choice of board for the project. However, there's no reason you couldn't adapt the code to the KB2040, Raspberry Pi Pico, or other board based on the same microcontroller.

Jeff's Verdict

The good about the NeXT Keyboard:

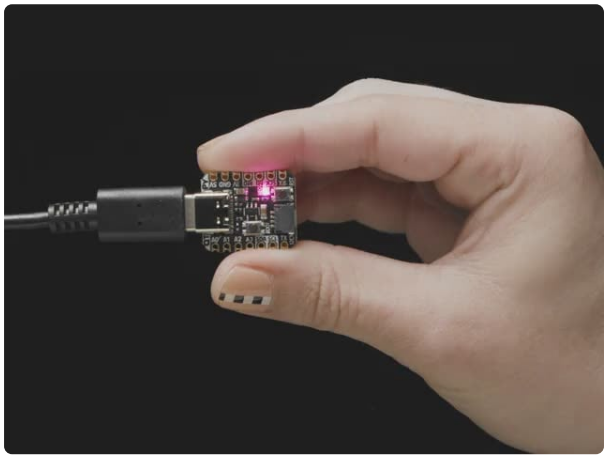
- Solid design
- Nice keyswitches
- How many times can I say "iconic" in this article?

The not so good:

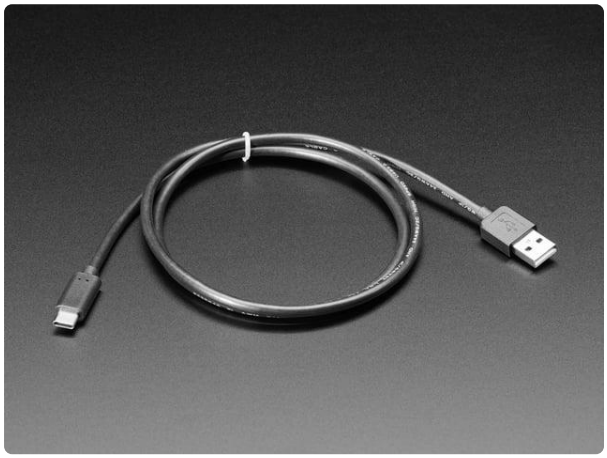
- Limited to 2-key rollover (+ modifier keys)
- Layout of modifier keys may frustrate some
- Missing F-keys, page up/down, etc.

The next page is Ladyada's original research about the keyboard protocol, which I followed closely while implementing the CircuitPython version. That's why it refers to Arduino. Don't worry, this really is a CircuitPython project.

Parts



Adafruit QT Py RP2040
What a cutie pie! Or is it... a QT Py? This diminutive dev board comes with one of our new favorite chip, the RP2040. It's been made famous in the new
https://www.adafruit.com/product/4900



USB Type A to Type C Cable - approx 1 meter / 3 ft long
As technology changes and adapts, so does Adafruit. This USB Type A to Type C cable will help you with the transition to USB C, even if you're still...
https://www.adafruit.com/product/4474

1 x NeXT Computer Keyboard
"Non-ADB" NeXT Computer Keyboard with 5-pin Mini-DIN connector

---

1 x Mini-DIN connector
Mini-DIN 5-pin receptacle, panel mount

https://www.digikey.com/en/products/detail/cui-devices/MD-50PL100/500828

M2.5x6 metal screws

2 x Mounting screws

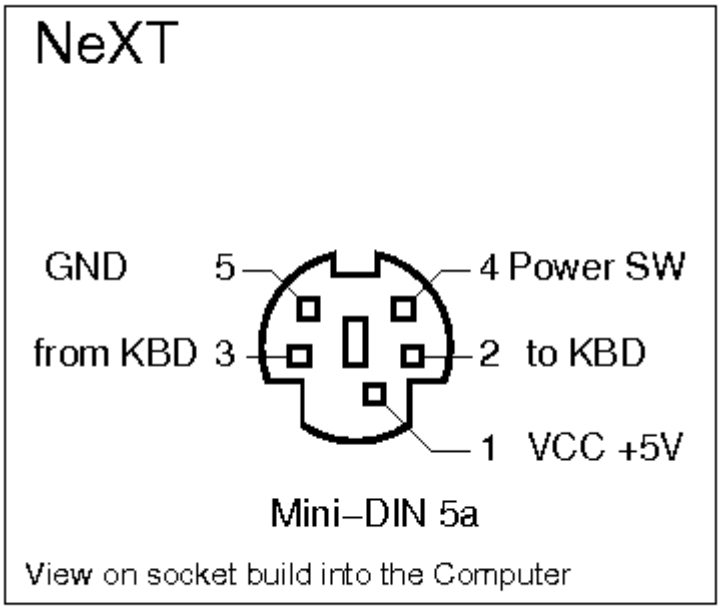
M2.5x6 metal screws

https://www.amazon.com/-/dp/B09WJ4WF9K

Research

The first thing to note is that the USB part (acting like a USB keyboard) is the easiest part of the project - there's already plenty of example code for how to do that with an Arduino Leonardo or Micro. The really tough part is figuring out how to read from the keyboard as it's not in any known or well documented protocol. The good news is whenever you're working with a really old technology, the computers back then were really slow and things weren't too complicated. Chances are whatever they did, it was meant to be simple and lightweight. Contrast this with a USB or Bluetooth or WiFi stack!

Our first stop is over at the awesome http://www.kbdbabel.org/conn/index.html (https://adafru.it/aTt) (http://archive.is/yqzKJ) (https://adafru.it/Cc5) where the nice author has documented the pinout of the keyboard. This is great because we won't accidentally smash the electronics with the wrong voltage. Also, it gives us a hint of how to talk to it. Power and ground at 5V are nice, easy to work with voltages. There's an RX and TX pin so at least we don't have to deal with a bi-directional or differential signal (whew).

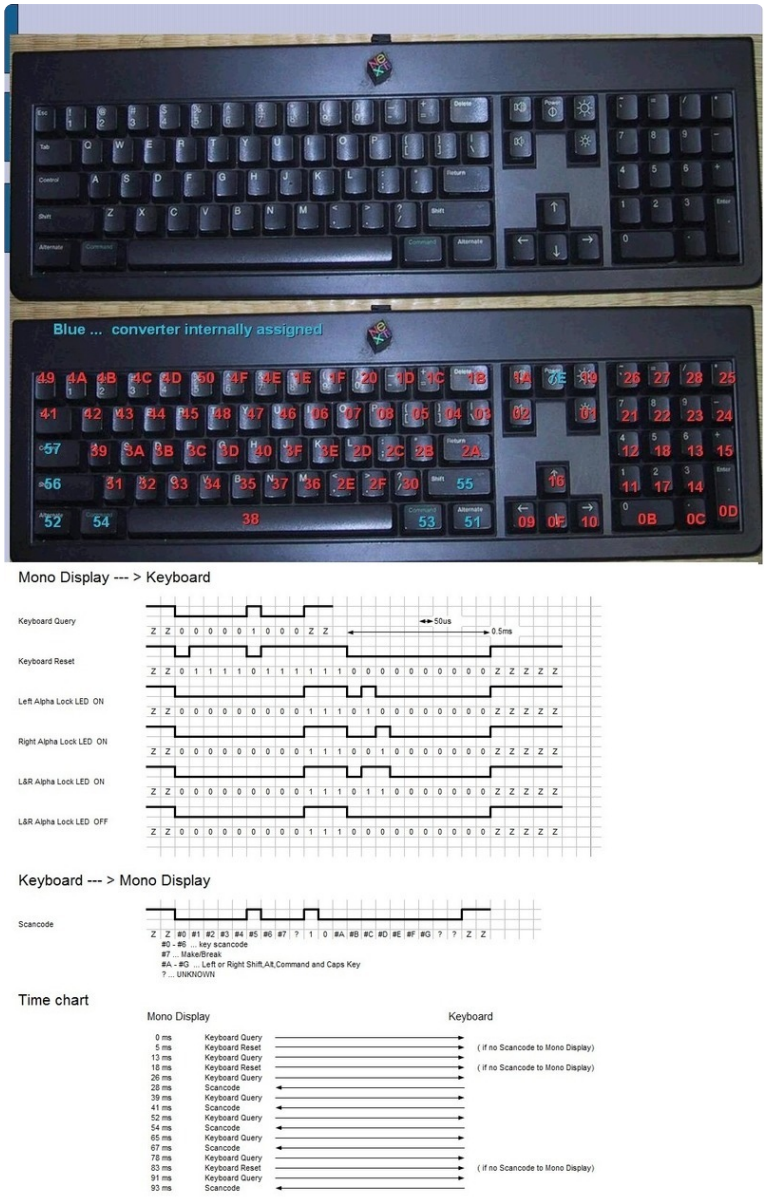


$Id: kbd_connector_next.fig,v 1.3 2007/12/09 20:23:55 akurz Exp $

Ok so now we can power it up. I applied +5V to VCC and ground to GND. I did see 5V on the "from KDB" pin, but unfortunately no actual data when keys were pressed. This

means that the keyboard isnt 'dumb' - it expects some sort of clock or reset signal on the "to KBD" pin. While one could try to figure it out cold, its a lot of effort.

Ideally, we'd have a NeXT that we could plug the keyboard into and 'sniff' the traffic, that is the easiest way to do it. Unfortunately, we don't have one. We were in crisis! But then we kept searching and looking around (btw, searching for "next keyboard" is not a very efficient way to locate this brand of keyboard!) and we lucked out when we found a Japanese website of serious keyboard enthusiast http://m0115.web.fc2.com/ (https://adafru.it/aTu) It is using frames so we weren't too optimistic we'd find a GitHub repo, but after a lot of clicking we found the holy grail of NeXT timing information:



Yes! This is exactly what we need, not only does he include the timing diagram, but also the timeline for resetting and querying. 50 microsecond timing is well within the

abilities of a 16 MHz microcontroller. Now we're ready to write code. (see the next section for the code listing)

The only thing remaining was the scancode table. By this point I, was 5 hours into this project and getting a little tired, when I realized that any operating system written for NeXT would have this all written up for me. In fact, there was an NetBSD port to NeXT and all the keyboard mapping data was there for me! Link to: NetBSD driver source (https://adafru.it/18hB), scancode table (https://adafru.it/18ic) via wayback machine.

Research II: Mouse

The NeXT keyboard also has the capability to connect the NeXT mouse. The keyboard's microcontroller reads the mouse movements and can send the mouse data to the computer on request.

The mouse uses a different connector, an 8-pin mini-DIN. Deskthority's wiki has the details of the connections (https://adafru.it/18ie).

In this case, though, it is enough to rely on the keyboard's ability to daisy chain, so we need to know how the keyboard sends back the data.

Hoping for the best, I plugged in my NeXT mouse and moved it around, hoping for some reports to come back. However, none did. Luckily, a little web searching showed me that it was necessary to send a second query command to get back mouse data; drak.org (https://adafru.it/18if) thoroughly documented the details. In a happy bit of synergy, they credit Adafruit for some of the original protocol information!

According to that website, the keyboard and mouse queries differ in one bit:

```
KB Query
-----_____-____----- (0x10 = 00010000)
Mouse Query
-----_-___-____----- (0x11 = 00010001)
```

Perhaps NeXT imagined that it might place even more "devices" on the keyboard bus in the future, as it looks like there are a lot of bits available to specify a device to query.

The response to the mouse query has information about the button states as well as the amount of mouse motion since the last query. Again, according to drak.org, the response looks like this:

```
Mouse Packet
-----_1XXXXXXX_-_2YYYYYYY_-----
1: Button 1, 0 = down
2: Button 2
XXXXXXX: X movement, 7 bit two's complement
YYYYYYY: Same thing for Y axis
```

That seems easy enough to code up. The plan is to alternately send a keyboard query and a mouse query, translating the response to USB HID. If the mouse is not plugged into the keyboard, everything still works fine: no mouse events are reported, but no errors occur either. So, the same CircuitPython code can work without caring whether there's a mouse attached.

Install CircuitPython

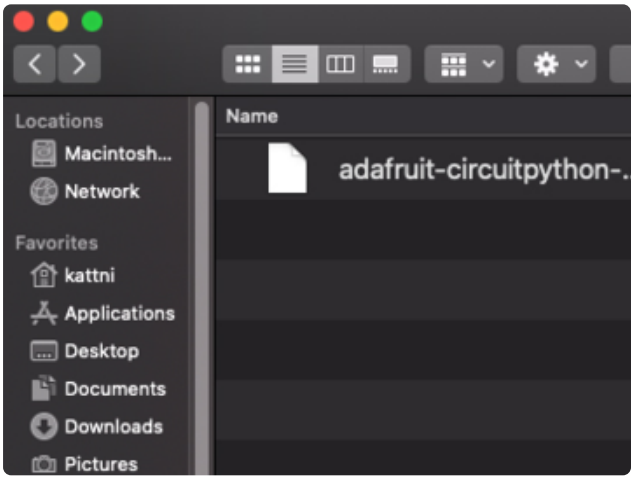
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

CircuitPython Quickstart

Follow this step-by-step to quickly get CircuitPython running on your board.

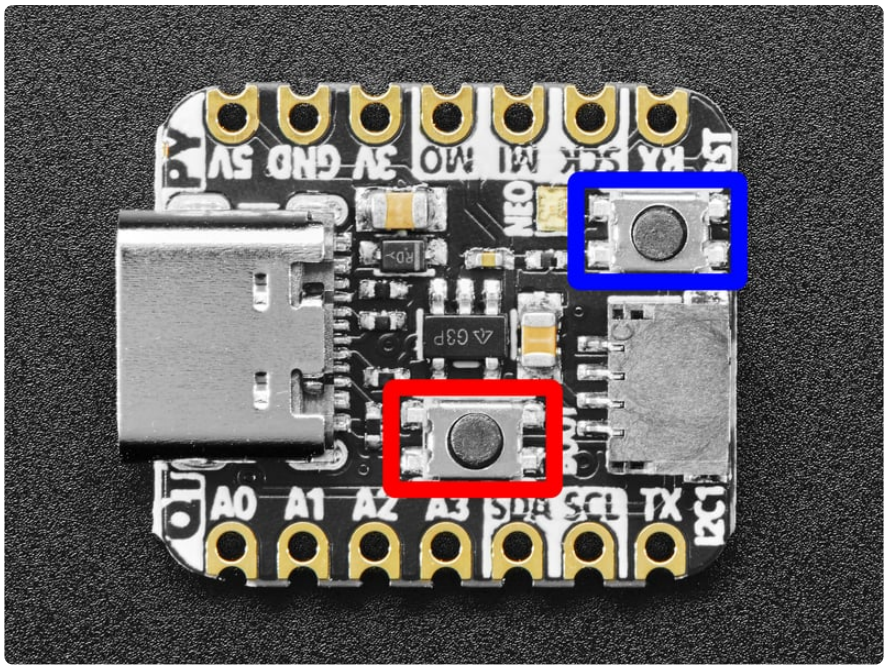
Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/RLD



Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.

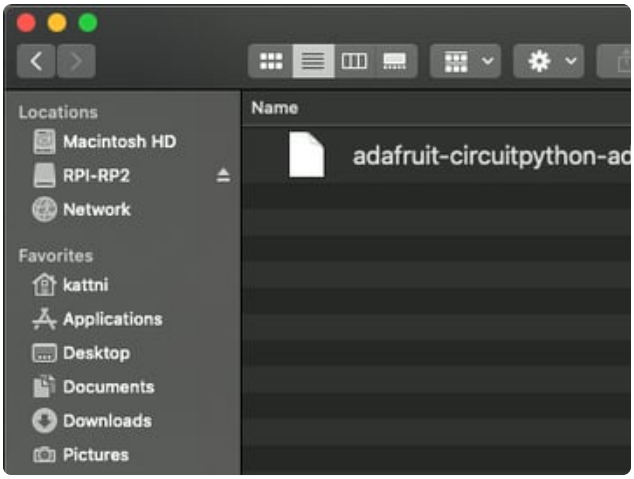


To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in blue above). **Continue to hold the BOOT/BOOTSEL button until the RPI-RP2 drive appears!**

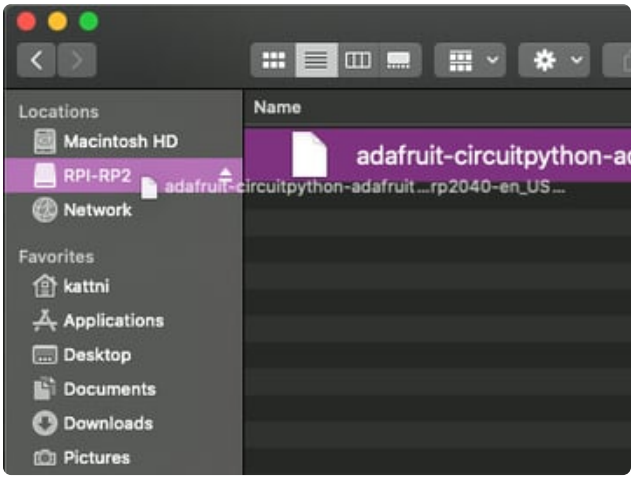
If the drive does not appear, release all the buttons, and then repeat the process above.

You can also start with your board unplugged from USB, press and hold the BOOTSEL button (highlighted in red above), continue to hold it while plugging it into USB, and wait for the drive to appear before releasing the button.

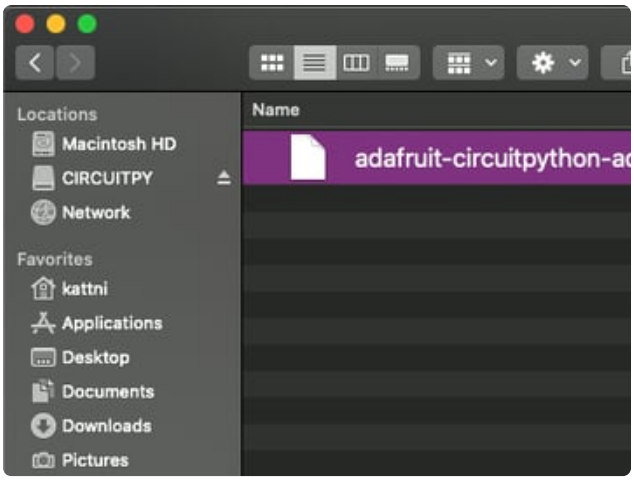
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



You will see a new disk drive appear called **RPI-RP2**.



Drag the **adafruit_circuitpython_etc.uf2** file to **RPI-RP2.**



The **RPI-RP2** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it, you're done! :)

Safe Mode

You want to edit your **code.py** or modify the files on your **CIRCUITPY** drive, but find that you can't. Perhaps your board has gotten into a state where **CIRCUITPY** is read-only. You may have turned off the **CIRCUITPY** drive altogether. Whatever the reason, safe mode can help.

Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in **boot.py** (where you can set **CIRCUITPY** read-only or turn it off completely). Second, it does not run the code in **code.py**. And finally, it does not automatically soft-reload when data is written to the **CIRCUITPY** drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the **CIRCUITPY** drive.

Entering Safe Mode

To enter safe mode when using CircuitPython, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED (highlighted in green above) will blink yellow during that time. If you press reset during that 1000ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

If you successfully enter safe mode on CircuitPython, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.

CircuitPython is in safe mode because you pressed the reset button during boot.
Press again to exit safe mode.

Press any key to enter the REPL. Use CTRL-D to reload.
```

You can now edit the contents of the **CIRCUITPY** drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

Flash Resetting UF2

If your board ever gets into a really weird state and CIRCUITPY doesn't show up as a disk drive after installing CircuitPython, try loading this 'nuke' UF2 to RPI-RP2. which

will do a 'deep clean' on your Flash Memory. **You will lose all the files on the board**, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.

Download flash erasing "nuke" UF2

https://adafru.it/RLE

Coding the Keyboard

Text Editor

Adafruit recommends using the **Mu** editor for editing your CircuitPython code. You can get more info in this guide (https://adafru.it/ANO).

Alternatively, you can use any text editor that saves simple text files.

Download the Project Bundle

Your project will use a specific set of CircuitPython libraries and the **code.py** file. To get everything you need, click on the **Download Project Bundle** link below, and uncompress the .zip file.

Hook your QT Py/board to your computer via a known good USB data+power cable. It should show up as a thumb drive named **CIRCUITPY**.

Using File Explorer/Finder (depending on your Operating System), drag the contents of the uncompressed bundle directory onto your board's **CIRCUITPY** drive, replacing any existing files or directories with the same names, and adding any new ones that are necessary.

Continue below the program listing for a breakdown of how the program functions, section by section.

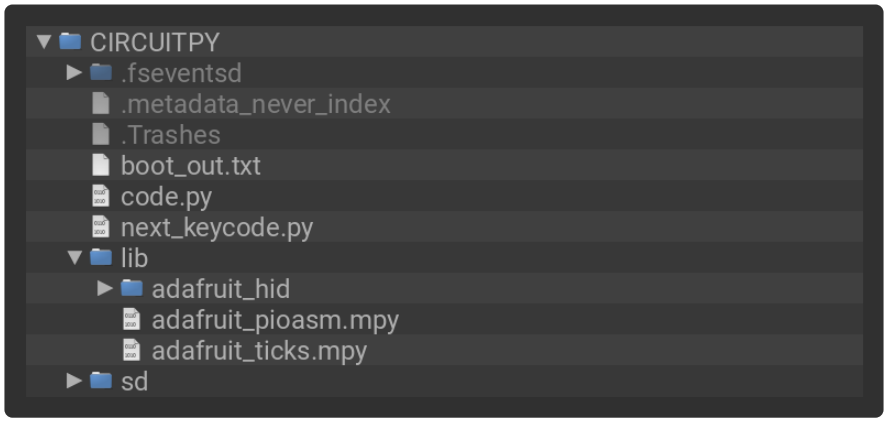
```
# SPDX-FileCopyrightText: 2022 Jeff Epler for Adafruit Industries
# SPDX-License-Identifier: MIT
import array
import time

import board
import rp2pio
import usb_hid
from keypad import Keys
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.keyboard import Keyboard
```

```
from adafruit_hid.keyboard import Keycode
from adafruit_hid.mouse import Mouse
from adafruit_pioasm import Program
from adafruit_ticks import ticks_add, ticks_less, ticks_ms
from next_keycode import (
    cc_value,
    is_cc,
    next_modifiers,
    next_scancodes,
    shifted_codes,
    shift_modifiers,
)


# Compared to a modern mouse, the DPI of the NeXT mouse is low. Increasing this
# number makes the pointer move further faster, but it also makes moves chunky.
# Customize this number according to the trade-off you want, but also check
# whether your operating system can assign a higher "sensitivity" or
# "acceleration" for the mouse.
MOUSE_SCALE = 8

# Customize the power key's keycode. You can change it to `Keycode.POWER` if
# you really want to accidentally power off your computer!
POWER_KEY_SENDS = Keycode.F1

# according to https://journal.spencerwnelson.com/entries/nextkb.html the
# keyboard's timing source is a 455MHz crystal, and the serial data rate is
# 1/24 the crystal frequency. This differs by a few percent from the "50us" bit
# time reported in other sources.
NEXT_SERIAL_BUS_FREQUENCY = round(455_000 / 24)

pio_program = Program(
    """
top:
    set pins, 1
    pull block             ; wait for send request
    out x, 1               ; trigger receive?
    out y, 7               ; get count of bits to transmit (minus 1)

bitloop:
    out pins, 1        [7] ; send next bit
    jmp y--, bitloop   [7] ; loop if bits left to send

    set pins, 1            ; idle the bus after last bit
    jmp !x, top            ; to top if no scancode expected

    set pins, 1            ; mark bus as idle so keyboard will send
    set y, 19              ; 20 bits to receive

    wait 0, pin 0 [7]      ; wait for falling edge plus half bit time
recvloop:
    in pins, 1 [7]         ; sample in the middle of the bit
    jmp y--, recvloop [7]  ; loop until all bits read

    push                   ; send report to CircuitPython
"""
)


def pack_message(bitcount, data, trigger_receive=False):
    if bitcount > 24:
        raise ValueError("too many bits in message")
    trigger_receive = bool(trigger_receive)
    message = (
        (trigger_receive << 31) | ((bitcount - 1) << 24) | (data << (24 - bitcount))
    )
    return array.array("I", [message])
```

```
def pack_message_str(bitstring, trigger_receive=False):
    bitcount = len(bitstring)
    data = int(bitstring, 2)
    return pack_message(bitcount, data, trigger_receive=trigger_receive)


def set_leds(i):
    return pack_message_str(f"0000000001110{i:02b}0000000")


QUERY = pack_message_str("000001000", True)
MOUSEQUERY = pack_message_str("010001000", True)
RESET = pack_message_str("01111011111110000000000")

BIT_BREAK = 1 << 11
BIT_MOD = 1


def is_make(report):
    return not bool(report & BIT_BREAK)


def is_mod_report(report):
    return not bool(report & BIT_MOD)


def extract_bits(report, *positions):
    result = 0
    for p in positions:
        result = (result << 1)
        if report & (1 << p):
            result |= 1
        #result = (result << 1) | bool(report & (1<<p))
    return result

# keycode bits are backwards compared to other information sources
# (bit 0 is first)
def keycode(report):
    return extract_bits(report, 12, 13, 14, 15, 16, 17, 18)


def modifiers(report):
    return (report >> 1) & 0x7F


sm = rp2pio.StateMachine(
    pio_program.assembled,
    first_in_pin=board.MISO,
    pull_in_pin_up=1,
    first_set_pin=board.MOSI,
    set_pin_count=1,
    first_out_pin=board.MOSI,
    out_pin_count=1,
    frequency=16 * NEXT_SERIAL_BUS_FREQUENCY,
    in_shift_right=False,
    wait_for_txstall=False,
    out_shift_right=False,
    **pio_program.pio_kwargs,
)


def signfix(num, sign_pos):
    """Fix a signed number if the bit with weight `sign_pos` is actually the sign
bit"""
    if num & sign_pos:
        return num - 2*sign_pos
    return num

class KeyboardHandler:
```

```
    def __init__(self):
        self.old_modifiers = 0
        self.cc = ConsumerControl(usb_hid.devices)
        self.kbd = Keyboard(usb_hid.devices)
        self.mouse = Mouse(usb_hid.devices)

    def set_key_state(self, key, state):
        if state:
            if isinstance(key, tuple):
                old_report_modifier = self.kbd.report_modifier[0]
                self.kbd.report_modifier[0] = 0
                self.kbd.press(*key)
                self.kbd.release_all()
                self.kbd.report_modifier[0] = old_report_modifier
            else:
                self.kbd.press(key)
        else:
            if isinstance(key, tuple):
                pass
            else:
                self.kbd.release(key)

    def handle_mouse_report(self, report):
        if report == 1536: # the "nothing happened" report
            return

        dx = extract_bits(report, 11,12,13,14,15,16,17)
        dx = -signfix(dx, 64)
        dy = extract_bits(report, 0,1,2,3,4,5,6)
        dy = -signfix(dy, 64)
        b1 = not extract_bits(report, 18)
        b2 = not extract_bits(report, 7)

        self.mouse.report[0] = (
            Mouse.MIDDLE_BUTTON if (b1 and b2) else
            Mouse.LEFT_BUTTON if b1 else
            Mouse.RIGHT_BUTTON if b2
            else 0)
        if dx or dy:
            self.mouse.move(dx * MOUSE_SCALE, dy * MOUSE_SCALE)
        else:
            self.mouse._send_no_move() # pylint: disable=protected-access

    def handle_report(self, report_value):
        if report_value == 1536: # the "nothing happened" report
            return

        # Handle modifier changes
        mods = modifiers(report_value)
        changes = self.old_modifiers ^ mods
        self.old_modifiers = mods
        for i in range(7):
            bit = 1 << i
            if changes & bit:  # Modifier key pressed or released
                self.set_key_state(next_modifiers[i], mods & bit)

        # Handle key press/release
        code = next_scancodes.get(keycode(report_value))
        if mods & shift_modifiers:
            code = shifted_codes.get(keycode(report_value), code)
        make = is_make(report_value)
        if code:
            if is_cc(code):
                if make:
                    self.cc.send(cc_value(code))
            else:
                self.set_key_state(code, make)

keys = Keys([board.SCK], value_when_pressed=False)
```

```
handler = KeyboardHandler()

recv_buf = array.array("I", [0])

time.sleep(0.1)
sm.write(RESET)
time.sleep(0.1)

for _ in range(4):
    sm.write(set_leds(3))
    time.sleep(0.1)
    sm.write(set_leds(0))
    time.sleep(0.1)

print("Keyboard ready!")

try:
    while True:
        if (event := keys.events.get()):
            handler.set_key_state(POWER_KEY_SENDS, event.pressed)

        sm.write(QUERY)
        deadline = ticks_add(ticks_ms(), 100)
        while ticks_less(ticks_ms(), deadline):
            if sm.in_waiting:
                sm.readinto(recv_buf)
                value = recv_buf[0]
                handler.handle_report(value)
                break
        else:
            print("keyboard did not respond - resetting")
            sm.restart()
            sm.write(RESET)
            time.sleep(0.1)

        sm.write(MOUSEQUERY)
        deadline = ticks_add(ticks_ms(), 100)
        while ticks_less(ticks_ms(), deadline):
            if sm.in_waiting:
                sm.readinto(recv_buf)
                value = recv_buf[0]
                handler.handle_mouse_report(value)
                break
        else:
            print("keyboard did not respond - resetting")
            sm.restart()
            sm.write(RESET)
            time.sleep(0.1)
finally:  # Release all keys before e.g., code is reloaded
    handler.kbd.release_all()
```



How it works

Preliminaries

Note how one of the `import` lines imports `from next_keycode`. This file, included alongside **code.py** in the project bundle, has the long list of correspondences between NeXT keyboard codes and USB keyboard codes.

After the required `import` lines, there is a line where you can customize the USB HID code sent by the power button. The guide defaults it to F1, because unexpectedly powering off a computer is not so great. Note that the button does not work to power on the computer, either, no matter how you configure it:

```
# SPDX-FileCopyrightText: 2022 Jeff Epler for Adafruit Industries
# SPDX-License-Identifier: MIT
import array
import time

import board
import rp2pio
import usb_hid
from keypad import Keys
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keyboard import Keycode
from adafruit_pioasm import Program
from adafruit_ticks import ticks_add, ticks_less, ticks_ms
from next_keycode import (
    cc_value,
    is_cc,
    next_modifiers,
    next_scancodes,
    shifted_codes,
    shift_modifiers,
)

# Customize the power key's keycode. You can change it to `Keycode.POWER` if
# you really want to accidentally power off your computer!
POWER_KEY_SENDS = Keycode.F1
```

Next, the code defines a program for the PIO peripheral to actually communicate with the keyboard.

A message can be a varying number of bits, and it can optionally call for a response from the keyboard. The PIO program expects this number first, accompanied by the bits to be sent.

Then, if the "response" bit is set, it will await a response from the keyboard. A response is always treated as consisting of 20 bits.

```
# according to https://journal.spencerwnelson.com/entries/nextkb.html the
# keyboard's timing source is a 455MHz crystal, and the serial data rate is
```

```
# 1/24 the crystal frequency. This differs by a few percent from the "50us" bit
# time reported in other sources.
NEXT_SERIAL_BUS_FREQUENCY = round(455_000 / 24)

pio_program = Program(
    """
top:
    set pins, 1
    pull block              ; wait for send request
    out x, 1                ; trigger receive?
    out y, 7                ; get count of bits to transmit (minus 1)

bitloop:
    out pins, 1         [7] ; send next bit
    jmp y--, bitloop    [7] ; loop if bits left to send

    set pins, 1             ; idle the bus after last bit
    jmp !x, top             ; to top if no scancode expected

    set pins, 1             ; mark bus as idle so keyboard will send
    set y, 19               ; 20 bits to receive

    wait 0, pin 0 [7]       ; wait for falling edge plus half bit time
recvloop:
    in pins, 1 [7]          ; sample in the middle of the bit
    jmp y--, recvloop [7]   ; loop until all bits read

    push                    ; send report to CircuitPython
"""
)
```

The next block of code is concerned with constructing messages to send to the keyboard. `QUERY` and `RESET` messages are defined for use later, as well as a function to set the keyboard LEDs:

```
def pack_message(bitcount, data, trigger_receive=False):
    if bitcount > 24:
        raise ValueError("too many bits in message")
    trigger_receive = bool(trigger_receive)
    message = (
        (trigger_receive << 31) | ((bitcount - 1) << 24) | (data << (24 - bitcount))
    )
    return array.array("I", [message])


def pack_message_str(bitstring, trigger_receive=False):
    bitcount = len(bitstring)
    data = int(bitstring, 2)
    return pack_message(bitcount, data, trigger_receive=trigger_receive)


def set_leds(i):
    return pack_message_str(f"0000000001110{i:02b}0000000")


QUERY = pack_message_str("000001000", True)
MOUSEQUERY = pack_message_str("010001000", True)
RESET = pack_message_str("01111011111110000000000")
```

When a key event is sent back to CircuitPython, we need to decode it. These constants and functions help with that task, allowing the keycodes, modifiers (keys

like CTRL and ALT which don't directly generate characters), and make/break state to be checked:

```
BIT_BREAK = 1 << 11
BIT_MOD = 1


def is_make(report):
    return not bool(report & BIT_BREAK)


def is_mod_report(report):
    return not bool(report & BIT_MOD)


def extract_bits(report, *positions):
    result = 0
    for p in positions:
        result = (result << 1)
        if report & (1 << p):
            result |= 1
        #result = (result << 1) | bool(report & (1<<p))
    return result


# keycode bits are backwards compared to other information sources
# (bit 0 is first)
def keycode(report):
    return extract_bits(report, 12, 13, 14, 15, 16, 17, 18)


def modifiers(report):
    return (report >> 1) & 0x7F
```

The power button is connected directly to one of the wires, so read it like a 1-key keypad instead:

```
keys = Keys([board.SCK], value_when_pressed=False)
```

There is also code concerned with decoding mouse messages, extracting the correct bits and assembling them into signed "delta" (movement) values for X and Y. It treats "left+right" button as middle button, which was a common thing on Linux computers with two button mice back in the 90s. Another useful thing to do when both buttons are pressed would be to look at the Y movement and send wheel motion instead of regular motion, which is pretty straightforward with the HID library.

```
def handle_mouse_report(self, report):
        if report == 1536: # the "nothing happened" report
            return

        dx = extract_bits(report, 11,12,13,14,15,16,17)
        dx = -signfix(dx, 64)
        dy = extract_bits(report, 0,1,2,3,4,5,6)
        dy = -signfix(dy, 64)
        b1 = not extract_bits(report, 18)
        b2 = not extract_bits(report, 7)

        self.mouse.report[0] = (
```

```
            Mouse.MIDDLE_BUTTON if (b1 and b2) else
            Mouse.LEFT_BUTTON if b1 else
            Mouse.RIGHT_BUTTON if b2
            else 0)
        if dx or dy:
            self.mouse.move(dx * MOUSE_SCALE, dy * MOUSE_SCALE)
        else:
            self.mouse._send_no_move() # pylint: disable=protected-access
```

Reset the keyboard, then blink the LEDs to show that everything's working:

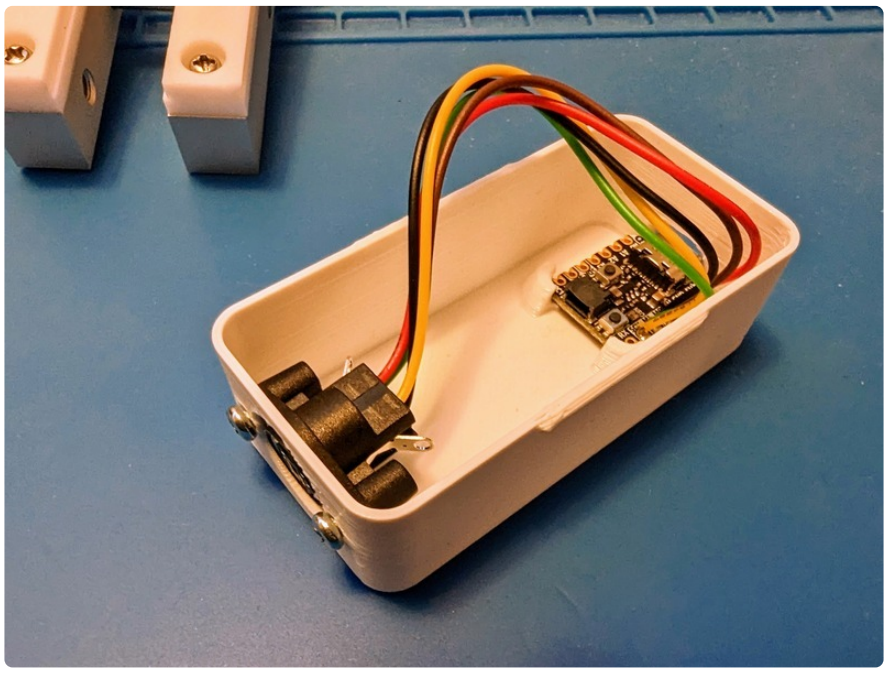
```
time.sleep(0.1)
sm.write(RESET)
time.sleep(0.1)

for _ in range(4):
    sm.write(set_leds(3))
    time.sleep(0.1)
    sm.write(set_leds(0))
    time.sleep(0.1)

print("Keyboard ready!")
```

In the forever loop, check for keyboard messages as well as for the independent power key. Act on each value as appropriate. When the program is stopped (for instance, because you updated **code.py**), use a '`finally` clause' to release all keys before actually exiting:

```
try:
    while True:
        if (event := keys.events.get()):
            handler.set_key_state(POWER_KEY_SENDS, event.pressed)

        sm.write(QUERY)
        deadline = ticks_add(ticks_ms(), 100)
        while ticks_less(ticks_ms(), deadline):
            if sm.in_waiting:
                sm.readinto(recv_buf)
                value = recv_buf[0]
                handler.handle_report(value)
                break
        else:
            print("keyboard did not respond - resetting")
            sm.restart()
            sm.write(RESET)
            time.sleep(0.1)

        sm.write(MOUSEQUERY)
        deadline = ticks_add(ticks_ms(), 100)
        while ticks_less(ticks_ms(), deadline):
            if sm.in_waiting:
                sm.readinto(recv_buf)
                value = recv_buf[0]
                handler.handle_mouse_report(value)
                break
        else:
            print("keyboard did not respond - resetting")
            sm.restart()
            sm.write(RESET)
            time.sleep(0.1)
finally:  # Release all keys before e.g., code is reloaded
    handler.kbd.release_all()
```

Until you've wired up the keyboard, CircuitPython won't do anything but show the message "keyboard did not respond - resetting" in the Serial REPL. You'll need to actually connect your keyboard to the QT Py as detailed on the following page.

Wiring the adapter & printing the enclosure



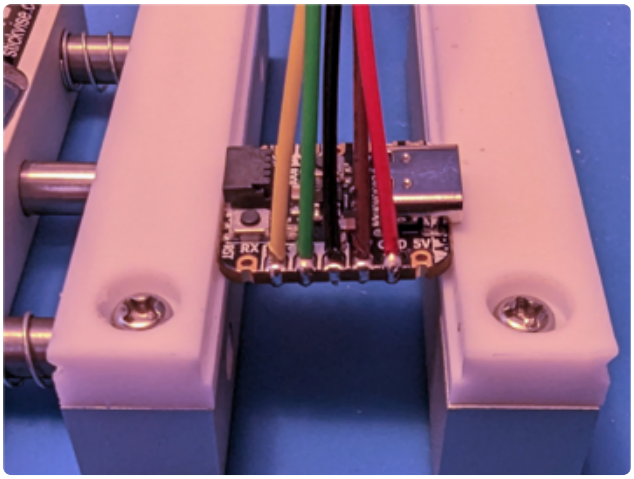
Wiring the adapter

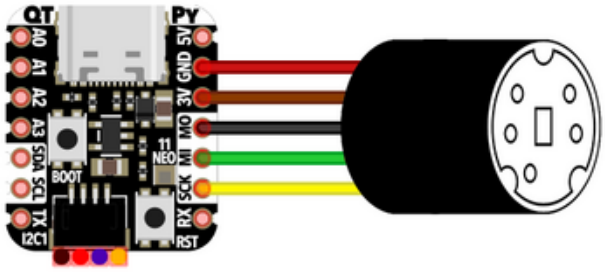
Just a few connections are needed between the QT Py RP2040 and the keyboard connector.

Watch out here, because the DIN wire colors do not correspond to classical wire coloring!

The original voltage specification of the keyboard is 5V, but my sample worked reliably with just 3.3V, which means no voltage level shifters are required.

The mini-DIN connector has pre-stripped, pre-tinned wires. Just insert them into the QT Py RP2040 from the **top** side and solder as follows:





Brown - VCC to QT Py **3V**
Black - data to KBD to QT Py **MO**
Green - **data from KBD** to QT Py **MI**
Yellow - **power SW** to QT Py **SCK**
Red - Ground to QT Py **GND**

fritzing

At this point, you you can test the software and hardware to make sure everything works. If not, verify that you've correctly copied the code and re-check the wiring. If problems persist, open the Serial REPL to check for any messages that may help you diagnose the problem.

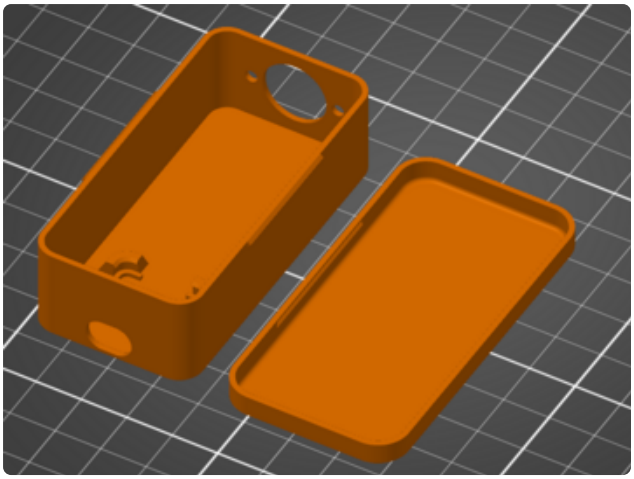
Printing the enclosure

Ladyada's original design went in a mint tin, as was the style in those days. I've prepared a 3D-printable enclosure with similar dimensions to the classic tin. To print, flip each part so that the large flat face is on the 3D printer bed. Use standard slicer settings, no support needed.

The zip file below contains all you need: STL and Step files for the top and bottom of the case, and the FreeCAD design file in case you want to make any modifications. (FreeCAD is a free and open source CAD package for Linux, Mac and Windows available at freecadweb.org (https://adafru.it/TFI))

qtmint.zip

https://adafru.it/18iA



Use "rotate" or "lay flat" in your slicer to flip the lid so that its large flat face is on the 3D printer bed.

To install the converter in the enclosure, place the side of the QT Py with the Stemma connector against the two PCB clips, then gently but firmly press down on the USB connector until the board clicks into place.

Use 2 M2.5x6 screw to secure the Mini-DIN connector to the enclosure.

Carefully route the wires out of the way and then snap on the enclosure lid.

If you need to, you can remove the QT Py by gently levering up on the USB connector while flexing that side of the box outwards.

Here's how the completed adapter should look on each end, with the connectors secure and accessible.

Plug the USB cable and the NeXT keyboard cable into the adapter, then plug the USB cable into your computer or laptop: