Custom HID Devices in CircuitPython

Created by Dan Halbert

https://learn.adafruit.com/custom-hid-devices-in-circuitpython

Last updated on 2024-08-04 02:40:08 PM EDT

Table of Contents

Overview

What is HID?

HID stands for "Human Interface Device". Keyboards, mice, digitizer tablets, joysticks, and game controllers are all HID devices. CircuitPython can emulate three standard HID devices by default: mouse, keyboard and consumer control. These are described in more detail in CircuitPython Essentials Guide (https://adafru.it/BhT) and the Customizing USB Devices Guide (https://adafru.it/VaJ).

All operating systems (e.g., Windows, macOS, or Linux) have built-in support for certain standard devices like keyboards and mice. But they vary about whether they support other less common devices. Often you need to install a driver to support a particular HID device, such as a specific game controller or a tablet.

Custom HID Devices

There are so many kinds of devices you might want to implement in CircuitPython, it cannot provide built-in support for them all. So CircuitPython lets you define and implement your own custom HID devices . With the right device definition and built-in OS support or a compatible driver, you can implement an existing HID device, or define your own custom variation.

Report Descriptors

Report Descriptors

To define an HID device, you need to supply an HID report descriptor. When you plug in an HID device, it sends its report descriptor(s) to the host computer. The report descriptor is binary data that specifies the device type and the details of the reports that the device sends and receives.

A report is binary data. A report sent from the device to the host computer is called an **IN** report. A report sent from the host to the device is an **OUT** report. **IN** and **OUT** are named from the perspective of the host.

For instance, a mouse report descriptor will declare that it is a device of type **Mouse**, with a certain number of **Buttons** and possibly a scroll **Wheel**. Every time you move the mouse, push a button, or move its scroll wheel, the mouse will send an IN report with 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.

Similarly, a keyboard will send IN reports saying which regular keys are pressed and which modifier keys (Shift, Ctrl, etc.) are pressed at the same time. In addition, most keyboards can receive OUT reports back from the host computer, which tell the keyboard to turn on and off its LEDs, such as the shift-lock indicator.

A Sample Report Descriptor

Below is a CircuitPython **boot.py** file that includes an example of a gamepad report descriptor.

The descriptor is a `bytes` string named `GAMEPAD_REPORT_DESCRIPTOR`. Note how the descriptor specifies a **Usage Page**, which is the general class of device, in this case, **Generic Desktop Controls,** and then a particular **Usage**, which is **Game Pad**. Then the descriptor specifies a **Report ID**, which here is 4. The report ID can be any value from 1 to 255, but must be unique for this report among all the reports in the devices presented by CircuitPython.

The descriptor then declares 16 on/off (0 or 1) **Buttons**, which fit in one bit each (**Report Count** of 16 and **Report Size** of 1), and four joystick axes, which are Usages **X**, **Y**, **Z**, and **Rz**. Each joystick value varies from -127 to 127, and fits in 8 bits.

The rest of the code creates a `Device` based on the descriptor, and includes it in a list of devices that also includes the default keyboard, mouse, and consumer control devices that CircuitPython usually presents. The Device constructor specifies the Report ID's used, and how many bytes are in the IN and OUT reports for each Report ID. In this case the IN report is 6 bytes long, and there is no OUT report. See the Customizing USB Devices Guide (https://adafru.it/VaJ) for more details about why this code needs to be in **boot.py**, and what `usb_hid.enable()` is doing.

```
# boot.py
import usb_hid

# This is only one example of a gamepad descriptor.
# It may not suit your needs, or be supported on your host computer.

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)
)
```

Creating a Report Descriptor

Writing your own HID report descriptor from scratch requires a lot of detailed knowledge, and is beyond the scope of this guide. However, you can often find the report descriptor for an existing HID device you want to emulate with a web search, and just use it as is, or modify it slightly for your purposes. If you have a device but don't have a published report descriptor for it, there are tools available to capture the report descriptor when you plug the device in. **usbhid-dump** is a such a tool you can use on Linux, and here are some similar tools for Windows and macOS (https://adafru.it/VaK). There are many others.

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.

The Microsoft Waratah tool (https://adafru.it/1a55) compiles HID report descriptors from a TOML-like description. It can make the process of writing report descriptors less error-prone.

You will also need to write a CircuitPython driver to handle your new device. The driver assembles a report and then passes it back to the host computer. You can see examples of HID devices drivers in the adafruit_hid (https://adafru.it/VaL) library. And the following pages in this guide will give you several examples of custom HID devices and drivers which you can just copy, or adapt for your own purposes.

Radial Controller

A radial controller is a HID device that transmits information about something that turns, such as a knob. Microsoft supports certain kinds of radial controllers on Windows (https://adafru.it/VaM), and sells the Microsoft Surface Dial, which is a radial controller with haptic feedback. A radial controller reports relative rotation changes as you turn its dial, and has a momentary switch you activate by pressing down.

The example below is a simple implementation of a radial controller, without haptic feedback, that works on Windows 10 or later. If you use it on Windows, the interactions will be similar to those described by Microsoft for the Surface Dial (https://adafru.it/VaN).

macOS and Linux don't support radial controllers right now, so try this project on Windows.

Software and Hardware Components

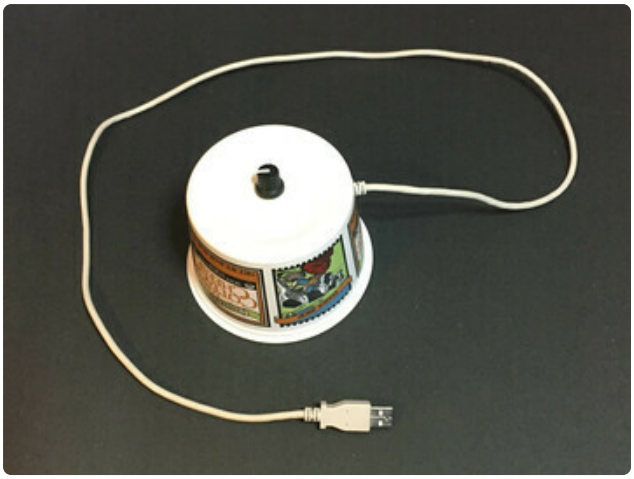
There are three pieces of software you use to implement this radial controller in CircuitPython. For hardware, you'll use a rotary encoder with a built-in switch.

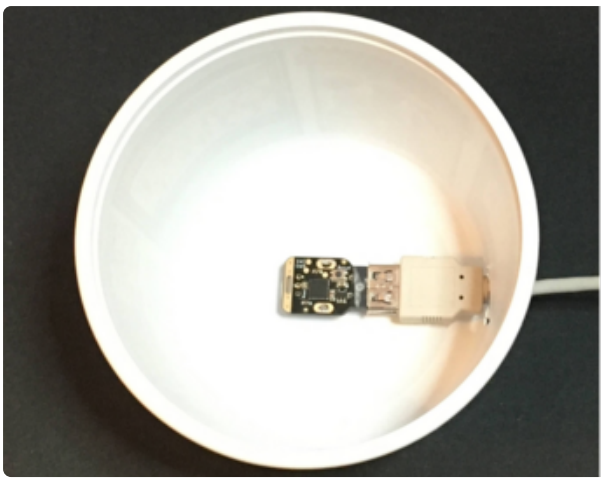
1. The adafruit_radial_controller library (https://adafru.it/VaO) (documentation (https://adafru.it/VaP)), which is in the Adafruit library bundle.
2. A **boot.py** file, which creates the device before USB starts.
3. A **code.py** file, which monitors a rotary encoder and passes state changes via the library. The **code.py** file also needs the `adafruit_hid` and `adafruit_debouncer` libraries.

This example is written for an Adafruit Rotary Trinkey (https://adafru.it/VaQ), but could be adapted for any board with a rotary encoder connected. You will need to change the pin names for other boards.

Note: The current radial controller report descriptor used in this example works, but is not perfect. We are still experimenting with some refinements.

This screnshot shows what you'll see on your **CIRCUITPY** drive when you have everything you need, except that you also need the **boot.py** file, which is missing from this picture.





The Rotary Trinkey will be easier to use if you mount it in something that can sit on your desk. For a finished project, you can make a 3D-printed enclosure reminiscent of the Media Dial (https://adafru.it/VaR) or the Microsoft Surface dial, with a base and a large top knob. But for a quick trial you can just mount the Trinkey in a food container or similar, as shown in the photos.

Installing the Project Code

Use the **Download Project Bundle** button in either code block below to download the project code. Unzip the download, and copy **boot.py**, **code.py** and the **lib** folder to your board.

boot.py

The **boot.py** file in this project creates the radial controller device, and sets it up to be available when **code.py** starts.

```
# SPDX-FileCopyrightText: Copyright (c) 2021 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: Unlicense

import usb_hid

import adafruit_radial_controller.device

REPORT_ID = 5

radial_controller_device = adafruit_radial_controller.device.device(REPORT_ID)
usb_hid.enable((radial_controller_device,))
```

code.py

The **code.py** program in this project first creates an `adafruit_radial_controller.RadialController`. Then it loops forever, continuously checks the state of the switch on the rotary encoder, and whether the encoder position has changed. If so, it sends a report with the changed state information. The encoder change is relative to the previous position.

A single click of the encoder in either direction is a `rotaryio.IncrementalEncoder.position` change of just 1 or -1. But that value is not sent directly. Experimentation showed that such a small value is sometimes ignored by Windows unless many occur in quick succession. So this multiplies the actual position change by `DEGREE_TENTHS_MULTIPLIER`, which is 100 in the code. You can experiment with other multiplier values if you'd like. Try 20 or 50, for example.

The rotation value sent is an integer, in units of tenths of a degree. So for instance, a sending a 1 represents a 0.1 degree rotation. This is not actually the rotation angle of the physical encoder we're using, but it is what the Microsoft driver expects.

```
# SPDX-FileCopyrightText: Copyright (c) 2021 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: Unlicense

import board
import digitalio
import rotaryio
import usb_hid

from adafruit_debouncer import Debouncer
import adafruit_radial_controller

switch = digitalio.DigitalInOut(board.SWITCH)
switch.pull = digitalio.Pull.DOWN
debounced_switch = Debouncer(switch)

encoder = rotaryio.IncrementalEncoder(board.ROTA, board.ROTB)

radial_controller = adafruit_radial_controller.RadialController(usb_hid.devices)

last_position = 0
DEGREE_TENTHS_MULTIPLIER = 100

while True:
    debounced_switch.update()
    if debounced_switch.rose:
        radial_controller.press()
    if debounced_switch.fell:
        radial_controller.release()

    position = encoder.position
    delta = position - last_position
    if delta != 0:
```

```
        radial_controller.rotate(delta * DEGREE_TENTHS_MULTIPLIER)
        last_position = position
```

N-Key Rollover (NKRO) Keyboard

by Jeff Epler (https://adafru.it/KPb)

CircuitPython's standard USB keyboard descriptor only supports pressing up to 6 non-modifier keys at a time, called 6-Key Rollover or 6KRO. This example shows how you can use an alternate USB descriptor to enable unlimited rollover (also called N-Key Rollover or NKRO) using the Adafruit MacroPad.

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

Drag the contents of the uncompressed bundle directory onto your MacroPad board **CIRCUITPY** drive, replacing any existing files or directories with the same names, and adding any new ones that are necessary. Once installed, the folder structure on **CIRCUITPY** should look like the image below.

Then, because this code requires a modified USB descriptor to be set in **boot.py**, click the reset button once to make those settings active. (To reverse them later, remove **boot.py** and reset the board again). You can check **boot_out.txt** which will contain the line `enabled HID with custom keyboard device` when **boot.py** is properly installed.


```
# SPDX-FileCopyrightText: 2021 Jeff Epler for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import keypad
import board
import usb_hid
from adafruit_hid.keyboard import Keyboard, find_device
from adafruit_hid.keycode import Keycode
```

```
key_pins = (
    board.KEY1,
    board.KEY2,
    board.KEY3,
    board.KEY4,
    board.KEY5,
    board.KEY6,
    board.KEY7,
    board.KEY8,
    board.KEY9,
    board.KEY10,
    board.KEY11,
    board.KEY12,
)

keys = keypad.Keys(key_pins, value_when_pressed=False, pull=True)

class BitmapKeyboard(Keyboard):
    def __init__(self, devices):
        device = find_device(devices, usage_page=0x1, usage=0x6)

        try:
            device.send_report(b'\0' * 16)
        except ValueError:
            print("found keyboard, but it did not accept a 16-byte report. check
that boot.py is installed properly")

        self._keyboard_device = device

        # report[0] modifiers
        # report[1:16] regular key presses bitmask
        self.report = bytearray(16)

        self.report_modifier = memoryview(self.report)[0:1]
        self.report_bitmap = memoryview(self.report)[1:]

    def _add_keycode_to_report(self, keycode):
        modifier = Keycode.modifier_bit(keycode)
        if modifier:
            # Set bit for this modifier.
            self.report_modifier[0] |= modifier
        else:
            self.report_bitmap[keycode >> 3] |= 1 << (keycode & 0x7)

    def _remove_keycode_from_report(self, keycode):
        modifier = Keycode.modifier_bit(keycode)
        if modifier:
            # Set bit for this modifier.
            self.report_modifier[0] &= ~modifier
        else:
            self.report_bitmap[keycode >> 3] &= ~(1 << (keycode & 0x7))

    def release_all(self):
        for i in range(len(self.report)):
            self.report[i] = 0
        self._keyboard_device.send_report(self.report)

kbd = BitmapKeyboard(usb_hid.devices)

keymap = [
    Keycode.ONE, Keycode.TWO, Keycode.THREE,
    Keycode.Q, Keycode.W, Keycode.E,
    Keycode.A, Keycode.S, Keycode.D,
    Keycode.Z, Keycode.X, Keycode.C]

while True:
    ev = keys.events.get()
    if ev is not None:
        key = keymap[ev.key_number]
```

```
        if ev.pressed:
            kbd.press(key)
        else:
            kbd.release(key)
```

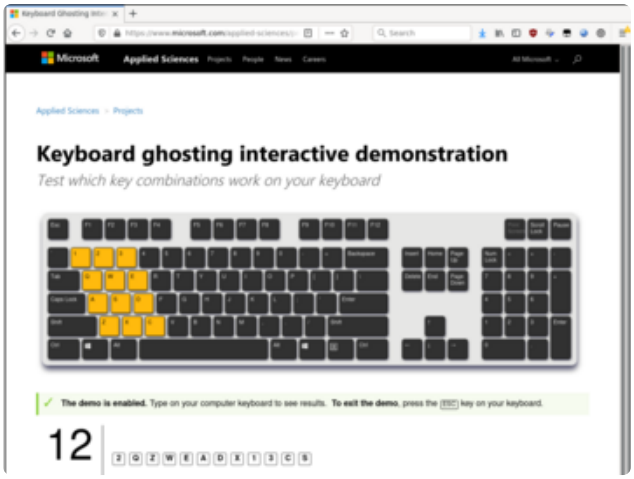
```
# SPDX-FileCopyrightText: 2021 Jeff Epler for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import usb_hid

REPORT_ID = 0x4
REPORT_BYTES = 16
bitmap_keyboard_descriptor = bytes((
        0x05, 0x01,                     # Usage Page (Generic Desktop),
        0x09, 0x06,                     # Usage (Keyboard),
        0xA1, 0x01,                     # Collection (Application),
        0x85, REPORT_ID,                #   Report ID
        # bitmap of modifiers
        0x75, 0x01,                     #   Report Size (1),
        0x95, 0x08,                     #   Report Count (8),
        0x05, 0x07,                     #   Usage Page (Key Codes),
        0x19, 0xE0,                     #   Usage Minimum (224),
        0x29, 0xE7,                     #   Usage Maximum (231),
        0x15, 0x00,                     #   Logical Minimum (0),
        0x25, 0x01,                     #   Logical Maximum (1),
        0x81, 0x02,                     #   Input (Data, Variable,
Absolute), ;Modifier byte
        # LED output report
        0x95, 0x05,                     #   Report Count (5),
        0x75, 0x01,                     #   Report Size (1),
        0x05, 0x08,                     #   Usage Page (LEDs),
        0x19, 0x01,                     #   Usage Minimum (1),
        0x29, 0x05,                     #   Usage Maximum (5),
        0x91, 0x02,                     #   Output (Data, Variable, Absolute),
        0x95, 0x01,                     #   Report Count (1),
        0x75, 0x03,                     #   Report Size (3),
        0x91, 0x03,                     #   Output (Constant),
        # bitmap of keys
        0x95, (REPORT_BYTES-1)*8,       #   Report Count (),
        0x75, 0x01,                     #   Report Size (1),
        0x15, 0x00,                     #   Logical Minimum (0),
        0x25, 0x01,                     #   Logical Maximum(1),
        0x05, 0x07,                     #   Usage Page (Key Codes),
        0x19, 0x00,                     #   Usage Minimum (0),
        0x29, (REPORT_BYTES-1)*8-1,     #   Usage Maximum (),
        0x81, 0x02,                     #   Input (Data, Variable, Absolute),
        0xc0                            # End Collection
))

bitmap_keyboard = usb_hid.Device(
    report_descriptor=bitmap_keyboard_descriptor,
    usage_page=0x1,
    usage=0x6,
    report_ids=(REPORT_ID,),
    in_report_lengths=(REPORT_BYTES,),
    out_report_lengths=(1,),
)

usb_hid.enable(
    (
        bitmap_keyboard,
        usb_hid.Device.MOUSE,
        usb_hid.Device.CONSUMER_CONTROL,
    )
)
print("enabled HID with custom keyboard device")
```



You can use Microsoft's Keyboard ghosting interactive demonstration (https://adafru.it/TUC) (it's a web page, so it should work on all kinds of computers) to see that you can press all the keys at the same time.

In **boot.py,** the special USB HID descriptor is established. **code.py** includes a version of the `Keyboard` class that works with the NKRO descriptor, as well as lines to establish the `Keys` object and to react to presses & releases by sending HID reports to the connected PC.