Table of Contents

Overview
- Adaptation Options
- Parts

CircuitPython on ItsyBitsy
- Set up CircuitPython Quick Start!
- Further Information

CircuitPython Libraries
- The Adafruit Learn Guide Project Bundle
- The Adafruit CircuitPython Library Bundle
- Downloading the Adafruit CircuitPython Library Bundle
- The CircuitPython Community Library Bundle
- Downloading the CircuitPython Community Library Bundle
- Understanding the Bundle
- Example Files
- Copying Libraries to Your Board
- Understanding Which Libraries to Install
- Example: ImportError Due to Missing Library
- Library Install on Non-Express Boards
- Updating CircuitPython Libraries and Examples
- CircUp CLI Tool

Assemble the Bonnet
- Cable Testing
- Adapter Board Build
- ProtoBonnet Headers
- Wiring
- Circuit Check
- Solder the ItsyBitsy to the Board
- Trim the Excess Pin Lengths
- Fit Check
- Enclosure
- PCB Adapter Build
- Pi Headers
- Itsy Headers
- Mounting

Code and Use the ItsyBitsy Keyboard
- Libraries
- Text Editor
- Code.py
- Testing
- Customizing the Code
- Keybow Orientation
- HID Keyboard Basics
- Keyboard Press/Release
- Media Control
- Key Assignements
- Colors
- Animation
Overview

Use the ItsyBitsy M4/M0 as a custom USB mechanical keypad with CircuitPython!

The Pimoroni Keybow is a 12 mechanical keyswitch USB HID keyboard controller designed to work with a Raspberry Pi Zero.

We'll build a custom Pi Bonnet to adapt the Keybow for use with the Adafruit ItsyBitsy M4 or M0 microcontroller board and programmable in CircuitPython.

You can create custom key mapping, macros, media control and your own DotStar RGB lighting with this simple, fast booting, super-charged keypad!

Adaptation Options

You've got a couple of options of how to adapt the Keybow to work with an ItsyBitsy M4 -- you can wire up the pins on a ProtoBonnet as shown in this guide, or order this nifty PCB (printed circuit board) designed by John Park for this exact purpose.
Here are the Eagle CAD files for the board:

- ItsyBitsy_Keybow_04.brd
- ItsyBitsy_Keybow04.sch

Parts

Adafruit ItsyBitsy M4 Express featuring ATSAMD51

What's smaller than a Feather but larger than a Trinket? It's an Adafruit ItsyBitsy M4 Express featuring the Microchip ATSAMD51! Small,...

https://www.adafruit.com/product/3800
Pimoroni Keybow Mini Mechanical Keyboard Kit with Raspberry Pi
Discontinued - you can grab this Adafruit MacroPad RP2040 Starter Kit - 3x4 Keys + Encoder + OLED - ADABOX019 Essentials...
https://www.adafruit.com/product/4144

Adafruit Perma Proto Bonnet Mini Kit
Design your own Bonnet or pHAT, attach custom circuitry and otherwise dress your Pi Zero with this jaunty prototyping Bonnet kit! To add to the
https://www.adafruit.com/product/3203

"Wire Wrap" Thin Prototyping & Repair Wire - 200m 30AWG Blue
This stuff is called "wire-wrap wire" because it used to be used for wire-wrapping high-speed digital circuits on a special kind of contact board. It's pretty rare to see wire-wrapping in...
https://www.adafruit.com/product/1446

Break-away 0.1" 2x36-pin strip dual male header (10 pieces)
Breakaway header is like the duct tape of electronics. Its great for connecting things together, soldering to perf-boards, fits into any breadboard, etc. We go through these guys real...
https://www.adafruit.com/product/1539
CircuitPython on ItsyBitsy

CircuitPython () is a derivative of MicroPython () 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.

Set up CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :)

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

Further Information

For more detailed info on installing CircuitPython, check out Installing CircuitPython ().

Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).
Plug your Itsy M4 into your computer using a known-good USB cable.

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

Double-click the Reset button on your board, and you will see the DotStar RGB LED turn green. If it turns red, check the USB cable, try another USB port, etc.

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!

You will see a new disk drive appear called ITSYBOOT.

Drag the adafruit_circuitpython_etc.uf2 file to ITSYBOOT.
The LED will flash. Then, the ITSYBOOT drive will disappear and a new disk drive called CIRCUITPY will appear.

That’s it, you’re done! :)

CircuitPython Libraries

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called libraries. Some of them are built into CircuitPython. Others are stored on your CIRCUITPY drive in a folder called lib. Part of what makes CircuitPython so great is its ability to store code separately from the firmware itself. Storing code separately from the firmware makes it easier to update both the code you write and the libraries you depend.

Your board may ship with a lib folder already, it’s in the base directory of the drive. If not, simply create the folder yourself. When you first install CircuitPython, an empty lib directory will be created for you.
CircuitPython libraries work in the same way as regular Python modules so the Python docs are an excellent reference for how it all should work. In Python terms, you can place our library files in the lib directory because it's part of the Python path by default.

One downside of this approach of separate libraries is that they are not built in. To use them, one needs to copy them to the CIRCUITPY drive before they can be used. Fortunately, there is a library bundle.

The bundle and the library releases on GitHub also feature optimized versions of the libraries with the .mpy file extension. These files take less space on the drive and have a smaller memory footprint as they are loaded.

Due to the regular updates and space constraints, Adafruit does not ship boards with the entire bundle. Therefore, you will need to load the libraries you need when you begin working with your board. You can find example code in the guides for your board that depends on external libraries.

Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

The Adafruit Learn Guide Project Bundle

The quickest and easiest way to get going with a project from the Adafruit Learn System is by utilising the Project Bundle. Most guides now have a Download Project Bundle button available at the top of the full code example embed. This button downloads all the necessary files, including images, etc., to get the guide project up and running. Simply click, open the resulting zip, copy over the right files, and you're good to go!

The first step is to find the Download Project Bundle button in the guide you're working on.

The Download Project Bundle button is only available on full demo code embedded from GitHub in a Learn guide. Code snippets will NOT have the button available.
When you copy the contents of the Project Bundle to your CIRCUITYPY drive, it will replace all the existing content! If you don’t want to lose anything, ensure you copy your current code to your computer before you copy over the new Project Bundle content!

The Download Project Bundle button downloads a zip file. This zip contains a series of directories, nested within which is the code.py, any applicable assets like images or audio, and the lib/ folder containing all the necessary libraries. The following zip was downloaded from the Piano in the Key of Lime guide.

The Piano in the Key of Lime guide was chosen as an example. That guide is specific to Circuit Playground Express, and cannot be used on all boards. Do not expect to download that exact bundle and have it work on your non-CPX microcontroller.

When you open the zip, you’ll find some nested directories. Navigate through them until you find what you need. You’ll eventually find a directory for your CircuitPython version (in this case, 7.x). In the version directory, you’ll find the file and directory you
need: code.py and lib/. Once you find the content you need, you can copy it all over to your CIRCUITPY drive, replacing any files already on the drive with the files from the freshly downloaded zip.

In some cases, there will be other files such as audio or images in the same directory as code.py and lib/. Make sure you include all the files when you copy things over!

Once you copy over all the relevant files, the project should begin running! If you find that the project is not running as expected, make sure you've copied ALL of the project files onto your microcontroller board.

That's all there is to using the Project Bundle!

The Adafruit CircuitPython Library Bundle

Adafruit provides CircuitPython libraries for much of the hardware they provide, including sensors, breakouts and more. To eliminate the need for searching for each library individually, the libraries are available together in the Adafruit CircuitPython Library Bundle. The bundle contains all the files needed to use each library.

Downloading the Adafruit CircuitPython Library Bundle

You can download the latest Adafruit CircuitPython Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

Match up the bundle version with the version of CircuitPython you are running. For example, you would download the 6.x library bundle if you're running any version of CircuitPython 6, or the 7.x library bundle if you're running any version of CircuitPython 7, etc. If you mix libraries with major CircuitPython versions, you will get incompatible mpy errors due to changes in library interfaces possible during major version changes.

Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in boot_out.txt file on the CIRCUITPY
drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

There's also a py bundle which contains the uncompressed python files, you probably don't want that unless you are doing advanced work on libraries.

The CircuitPython Community Library Bundle

The CircuitPython Community Library Bundle is made up of libraries written and provided by members of the CircuitPython community. These libraries are often written when community members encountered hardware not supported in the Adafruit Bundle, or to support a personal project. The authors all chose to submit these libraries to the Community Bundle make them available to the community.

These libraries are maintained by their authors and are not supported by Adafruit. As you would with any library, if you run into problems, feel free to file an issue on the GitHub repo for the library. Bear in mind, though, that most of these libraries are supported by a single person and you should be patient about receiving a response. Remember, these folks are not paid by Adafruit, and are volunteering their personal time when possible to provide support.

Downloading the CircuitPython Community Library Bundle

You can download the latest CircuitPython Community Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

[Click for the latest CircuitPython Community Library Bundle release]

The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in boot_out.txt file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.
Understanding the Bundle

After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.

Open the bundle folder. Inside you'll find two information files, and two folders. One folder is the lib bundle, and the other folder is the examples bundle.

Now open the lib folder. When you open the folder, you'll see a large number of .mpy files, and folders.

Example Files

All example files from each library are now included in the bundles in an examples directory (as seen above), as well as an examples-only bundle. These are included for two main reasons:

- Allow for quick testing of devices.
- Provide an example base of code, that is easily built upon for individualized purposes.
Copying Libraries to Your Board

First open the lib folder on your CIRCUITPY drive. Then, open the lib folder you extracted from the downloaded zip. Inside you'll find a number of folders and .mpy files. Find the library you'd like to use, and copy it to the lib folder on CIRCUITPY.

If the library is a directory with multiple .mpy files in it, be sure to copy the entire folder to CIRCUITPY/lib.

This also applies to example files. Open the examples folder you extracted from the downloaded zip, and copy the applicable file to your CIRCUITPY drive. Then, rename it to code.py to run it.

If a library has multiple .mpy files contained in a folder, be sure to copy the entire folder to CIRCUITPY/lib.

Understanding Which Libraries to Install

You now know how to load libraries on to your CircuitPython-compatible microcontroller board. You may now be wondering, how do you know which libraries you need to install? Unfortunately, it's not always straightforward. Fortunately, there is an obvious place to start, and a relatively simple way to figure out the rest. First up: the best place to start.

When you look at most CircuitPython examples, you'll see they begin with one or more `import` statements. These typically look like the following:

- `import library_or_module`

However, `import` statements can also sometimes look like the following:

- `from library_or_module import name`
They can also have more complicated formats, such as including a `try / except` block, etc.

The important thing to know is that an `import` statement will always include the name of the module or library that you're importing.

Therefore, the best place to start is by reading through the `import` statements.

Here is an example import list for you to work with in this section. There is no setup or other code shown here, as the purpose of this section involves only the import list.

```python
import time
import board
import neopixel
import adafruit_lis3dh
import usb_hid
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.consumer_control_code import ConsumerControlCode
```

Keep in mind, not all imported items are libraries. Some of them are almost always built-in CircuitPython modules. How do you know the difference? Time to visit the REPL.

In the **Interacting with the REPL section** on The REPL page in this guide, the `help("modules")` command is discussed. This command provides a list of all of the built-in modules available in CircuitPython for your board. So, if you connect to the serial console on your board, and enter the REPL, you can run `help("modules")` to see what modules are available for your board. Then, as you read through the `import` statements, you can, for the purposes of figuring out which libraries to load, ignore the statement that import modules.

The following is the list of modules built into CircuitPython for the Feather RP2040. Your list may look similar or be anything down to a significant subset of this list for smaller boards.
Now that you know what you're looking for, it's time to read through the import statements. The first two, `time` and `board`, are on the modules list above, so they're built-in.

The next one, `neopixel`, is not on the module list. That means it's your first library! So, you would head over to the bundle zip you downloaded, and search for neopixel. There is a neopixel.mpy file in the bundle zip. Copy it over to the lib folder on your CI RCUITPY drive. The following one, `adafruit_lis3dh`, is also not on the module list. Follow the same process for adafruit_lis3dh, where you'll find adafruit_lis3dh.mpy, and copy that over.

The fifth one is `usb_hid`, and it is in the modules list, so it is built in. Often all of the built-in modules come first in the import list, but sometimes they don't! Don't assume that everything after the first library is also a library, and verify each import with the modules list to be sure. Otherwise, you'll search the bundle and come up empty!

The final two imports are not as clear. Remember, when `import` statements are formatted like this, the first thing after the `from` is the library name. In this case, the library name is `adafruit_hid`. A search of the bundle will find an adafruit_hid folder. When a library is a folder, you must copy the entire folder and its contents as it is in the bundle to the lib folder on your CIRCUITPY drive. In this case, you would copy the entire adafruit_hid folder to your CIRCUITPY/lib folder.

Notice that there are two imports that begin with `adafruit_hid`. Sometimes you will need to import more than one thing from the same library. Regardless of how many times you import the same library, you only need to load the library by copying over the adafruit_hid folder once.

That is how you can use your example code to figure out what libraries to load on your CircuitPython-compatible board!
There are cases, however, where libraries require other libraries internally. The internally required library is called a dependency. In the event of library dependencies, the easiest way to figure out what other libraries are required is to connect to the serial console and follow along with the `ImportError` printed there. The following is a very simple example of an `ImportError`, but the concept is the same for any missing library.

**Example: ImportError Due to Missing Library**

If you choose to load libraries as you need them, or you're starting fresh with an existing example, you may end up with code that tries to use a library you haven't yet loaded. This section will demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

This demonstration will only return an error if you do not have the required library loaded into the lib folder on your CIRCUITPY drive.

Let's use a modified version of the Blink example.

```python
import board
import time
import simpleio

led = simpleio.DigitalOut(board.LED)

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.
You have an `ImportError`. It says there is no module named `simpleio`. That's the one you just included in your code!

Click the link above to download the correct bundle. Extract the lib folder from the downloaded bundle file. Scroll down to find simpleio.mpy. This is the library file you're looking for! Follow the steps above to load an individual library file.

The LED starts blinking again! Let's check the serial console.

No errors! Excellent. You've successfully resolved an `ImportError`!

If you run into this error in the future, follow along with the steps above and choose the library that matches the one you're missing.

## Library Install on Non-Express Boards

If you have an M0 non-Express board such as Trinket M0, Gemma M0, QT Py M0, or one of the M0 Trinkeys, you'll want to follow the same steps in the example above to install libraries as you need them. Remember, you don't need to wait for an `ImportError` if you know what library you added to your code. Open the library bundle you downloaded, find the library you need, and drag it to the lib folder on your CIRCUITPY drive.

You can still end up running out of space on your M0 non-Express board even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find suggestions on the [Troubleshooting page](#).

## Updating CircuitPython Libraries and Examples

Libraries and examples are updated from time to time, and it's important to update the files you have on your CIRCUITPY drive.

To update a single library or example, follow the same steps above. When you drag the library file to your lib folder, it will ask if you want to replace it. Say yes. That's it!
A new library bundle is released every time there's an update to a library. Updates include things like bug fixes and new features. It's important to check in every so often to see if the libraries you're using have been updated.

**CircUp CLI Tool**

There is a command line interface (CLI) utility called CircUp () that can be used to easily install and update libraries on your device. Follow the directions on the install page within the CircUp learn guide (). Once you've got it installed you run the command `circup update` in a terminal to interactively update all libraries on the connected CircuitPython device. See the usage page in the CircUp guide () for a full list of functionality.

---

**Assemble the Bonnet**

The Keybow was designed to use power, ground, twelve IO pins for the switches, and the two SPI clock and data lines (for the DotStar LEDs) from the Raspberry Pi Zero.

To use the Keybow with the ItsyBitsy M4 or M0, we will redirect those Keybow pins from the 2x20 connector to an ItsyBitsy's layout. The Adafruit ProtoBonnet works well for adapting the Keybow's switch and DotStar pinout to the ItsyBitsy M4/M0.

Here's how the pins are mapped:

<table>
<thead>
<tr>
<th>Pi GPIO</th>
<th>Pi Physical Pin</th>
<th>ItsyBitsy M4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

©Adafruit Industries
<table>
<thead>
<tr>
<th>PIN</th>
<th>Channel</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>5V</td>
<td>2</td>
<td>Vhi</td>
</tr>
<tr>
<td>GPIO17</td>
<td>11</td>
<td>A2</td>
</tr>
<tr>
<td>GPIO27</td>
<td>13</td>
<td>A1</td>
</tr>
<tr>
<td>GPIO22</td>
<td>15</td>
<td>A5</td>
</tr>
<tr>
<td>GPIO23</td>
<td>16</td>
<td>A0</td>
</tr>
<tr>
<td>GPIO24</td>
<td>18</td>
<td>A4</td>
</tr>
<tr>
<td>MOSI</td>
<td>19</td>
<td>MO</td>
</tr>
<tr>
<td>SCLCK</td>
<td>23</td>
<td>SCK</td>
</tr>
<tr>
<td>GPIO5</td>
<td>29</td>
<td>A3</td>
</tr>
<tr>
<td>GPIO6</td>
<td>31</td>
<td>D10</td>
</tr>
<tr>
<td>GPIO12</td>
<td>32</td>
<td>D9</td>
</tr>
<tr>
<td>GPIO13</td>
<td>33</td>
<td>D7</td>
</tr>
<tr>
<td>GPIO16</td>
<td>36</td>
<td>D12</td>
</tr>
<tr>
<td>GPIO26</td>
<td>37</td>
<td>D2</td>
</tr>
<tr>
<td>GPIO20</td>
<td>38</td>
<td>D11</td>
</tr>
<tr>
<td>GND</td>
<td>(many)</td>
<td>GND</td>
</tr>
</tbody>
</table>

The Keybow to ItsyBitsy pin mapping looks like this:
Here is the same circuit diagrammed on the PermaBonnet:
Alternately, here's what a PCB for the adapter looks like. It's included here as it may make the connections a bit clearer to follow than the wire diagram above.
Cable Testing

Initially, I tested the connections using M/F jumper cables and an ItsyBitsy M4 with male header pins soldered on (the final build required a fresh ItsyBitsy due to the different header pin orientation used).

This was good for proving out the pin assignments, but not entirely as ergonomic a package as desired.

Adapter Board Build

Now, we'll build the adapter board to use the ItsyBitsy seamlessly with the Keybow.
To build the proper adapter board, we'll use the ProtoBonnet. To it, solder the 2x20 male header pins and the two 14 pin, single-row headers spaced for the ItsyBitsy as shown here.

A good way to align the single-row pins is to tape the ItsyBitsy to the board as shown, then flip the board over for soldering.

Wait until the end to solder the ItsyBitsy itself onto the male headers, so that you can have room to solder the wiring in place.
Carefully follow the wiring diagrams and begin soldering short lengths of wire to make the connections between the Pi pin breakout row on the ProtoBonnet and the associated pins of the ItsyBitsy.

I found it easier to tack-solder some of the wires from the bottom side before flipping over to fully solder.

You'll notice there is one Pi pin we need to use that isn't broken out to the ProtoBonnet -- GPIO 26, which runs to D2 on the ItsyBitsy. For this one, we'll solder the wire directly to the header pin as shown.

This type of wiring can be exacting and requires a lot of concentration. Be sure to double check your work and take breaks!
Circuit Check

Once you’re done soldering the circuit, use your multimeter in continuity mode to check each connection. Set one probe on a GPIO header pin and the other one on the associated ItsyBitsy header pin to make sure you have continuity.

Additionally, check to make sure all ground connections are shared, and that there are no shorts between power and ground.

Solder the ItsyBitsy to the Board

Now you can place the ItsyBitsy M4/M0 back onto the header pins and solder them all in place.
Trim the Excess Pin Lengths

While wearing eye protection, use diagonal flush cutters to trim the excess lengths of the ItsyBitsy header pins ONLY! Do not trim the 2x20 Raspberry Pi Bonnet headers!!
Fit Check

Before connecting the ItsyBitsy adapter board to the acrylic base, do a test fit to make sure the board connects properly to the Keybow.
Now, you can connect the ItsyBitsy adapter board to the Keybow base and fully assemble the board using M2.5 nylon standoffs and screws.

Follow the original Keybow instructions () for this, substituting the ItsyBitsy adapter board for the Pi Zero. Since the wiring and header pins make this board a bit taller than the Pi Zero you may need an extra bit of room to fit it by adding a longer screws in a couple of spots. The nylon M2.5 screw and standoff kit is great for this.
PCB Adapter Build

If you choose to order a PCB adapter instead, here's how you'll assemble it.

**Pi Headers**

Place the 2x20 male Pi header pins into the board as shown.

You can use the Keybow to hold the pins in place as you solder them from the back side of the board.
Itsy Headers

Next, fit the two ItsyBitsy male header pin rows into the board as shown.

You can hold them in place with some poster putty or tape and then flip the board over and solder the pins in from the back. Push the pins flush with the board with the soldering iron as you go for the best fit, or trim the extra points off later with some flush cutters. (The Pi Zero we're replacing has a totally flat back.)

If you want to go the extra mile, do not solder the ItsyBitsy in place yet! We're going to remove the headers' black plastic spacers to get a super low profile so the ItsyBitsy will fit neatly in the case.

Alternately, you can solder the ItsyBitsy to the unmodified pins, but you may need some extra spacers for the acrylic case.

Carefully pry up the black plastic spacers, you'll be left with bare pins (or a very tiny bed of nails depending on your perspective).

Now, solder the ItsyBitsy M4 to the pins, and then trim off the excess.
Mounting
You can now mount the Itsy/Pi adapter to the acrylic case -- it has the same dimensions as the Pi Zero it is replacing.

Affix the Keybow to the board and off we go!
Next, we'll code the ItsyBitsy Keybow in CircuitPython so it can be customized to send HID USB keyboard commands and light the DotStar LEDs.

Code and Use the ItsyBitsy Keyboard
Libraries

Once your ItsyBitsy is set up with CircuitPython 5.3.0 or greater, you'll also need to add some libraries. Follow this page for information on how to download and add libraries to your ItsyBitsy.

From the library bundle you downloaded in that guide page, transfer the following libraries onto the ItsyBity /lib directory on the CIRCUITPY drive:

- adafruit_bus_device
- adafruit_dotstar
- adafruit_hid
- adafruit_led_animation
- neopixel
- simpleio

Text Editor

Adafruit recommends using the Mu editor for using your CircuitPython code with the ItsyBitsy boards. You can get more info in this guide.

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

Code.py

Copy the code below and paste it into Mu. Then, save it to your ItsyBitsy as code.py.

```python
# SPDX-FileCopyrightText: 2020 John Park for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# ItsyBitsy Keypad
# Uses ItsyBitsy M4/M0 plus Pimoroni Keybow
# To build a customizable USB keypad

import time
import board
from digitalio import DigitalInOut, Direction, Pull
import usb_hid
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keycode import Keycode
from adafruit_hid.consumer_control import ConsumerControl
```

©Adafruit Industries
from adafruit_hid.consumer_control_code import ConsumerControlCode
import adafruit_dotstar as dotstar

dots = dotstar.DotStar(board.SCK, board.MOSI, 12, brightness=0.4)

RED = 0xFF0000
AMBER = 0xAA9900
BLUE = 0x0066FF
MAGENTA = 0xFF00FF
PURPLE = 0x3B0F85
BLACK = 0x000000

kbd = Keyboard(usb_hid.devices)
cc = ConsumerControl(usb_hid.devices)

orientation = 1  # 0 = portrait/vertical, 1 = landscape/horizontal
if orientation == 0:
    key_dots = [0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11]
    # 0  #4  #8
    # 1  #5  #9
    # 2  #6  #10
    # 3  #7  #11
if orientation == 1:
    key_dots = [3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8]
    # 3  #2  #1  #0
    # 7  #6  #5  #4
    # 11 #10 #9  #8

def dot_on(dot, color):
    dots[dot] = color

def dot_off(dot):
    dots[dot] = BLACK

# Pin definitions
if orientation == 0:  # 0 = portrait/vertical
    pins = [
        board.D11,
        board.D12,
        board.D2,
        board.D10,
        board.D9,
        board.D7,
        board.A5,
        board.A4,
        board.A3,
        board.A2,
        board.A1,
        board.A0,
    ]
if orientation == 1:  # 1 = landscape/horizontal
    pins = [
        board.A2,
        board.A5,
        board.D10,
        board.D11,
        board.A1,
        board.A4,
        board.A3,
        board.A0,
        board.A3,
        board.D7,
        board.D2,
    ]

# the two command types -- MEDIA for ConsumerControlCodes, KEY for Keycodes
# this allows button press to send the correct HID command for the type specified
MEDIA = 1
KEY = 2
keymap = {
    (0): (AMBER, MEDIA, ConsumerControlCode.PLAY_PAUSE),
    (1): (AMBER, MEDIA, ConsumerControlCode.MUTE),
    (2): (AMBER, MEDIA, ConsumerControlCode.VOLUME_DECREMENT),
    (3): (AMBER, MEDIA, ConsumerControlCode.VOLUME_INCREMENT),
    (4): (BLUE, KEY, (Keycode.GUI, Keycode.C)),
    (5): (BLUE, KEY, (Keycode.GUI, Keycode.V)),
    (6): (MAGENTA, KEY, [Keycode.UP_ARROW]),
    (7): (PURPLE, KEY, [Keycode.BACKSPACE]),
    (8): (BLUE, KEY, [Keycode.SPACE]),
    (9): (MAGENTA, KEY, [Keycode.LEFT_ARROW]),
    (10): (MAGENTA, KEY, [Keycode.DOWN_ARROW]),
    (11): (MAGENTA, KEY, [Keycode.RIGHT_ARROW]),
}
switches = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
for i in range(12):
    switches[i] = DigitalInOut(pins[i])
    switches[i].direction = Direction.INPUT
    switches[i].pull = Pull.UP

switch_state = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print("ItsyBitsy Keybow")

# Starup lights
for k in range(12):
    dot_on(key_dots[k], RED)
    time.sleep(0.05)
    dot_on(key_dots[k], keymap[k][0])  # use individual key colors from set
    time.sleep(0.05)

while True:
    for button in range(12):
        if switch_state[button] == 0:
            if not switches[button].value:
                try:
                    if keymap[button][1] == KEY:
                        kbd.press(*keymap[button][2])
                    else:
                        cc.send(keymap[button][2])
                except ValueError:
                    pass
                dot_on(key_dots[button], RED)
                switch_state[button] = 1
                print("pressed key{}").format(button)

        if switch_state[button] == 1:
            if switches[button].value:
                try:
                    if keymap[button][1] == KEY:
                        kbd.release(*keymap[button][2])
dot_on(key_dots[button], keymap[button][0])
                except ValueError:
                    pass
                print("released key{}").format(button)
                switch_state[button] = 0
                time.sleep(0.01)  # debounce
With the code.py loaded onto your ItsyBitsy M4/M0, you'll see the Keybow DotStar LEDs light up with a startup animation, and then they'll settle into their color coded default stated.

Try pressing the four amber top row keys -- they control media play/pause, mute, and volume up/down.

To test the dark blue copy/paste keys, first select some text and hit the left copy key. Then, place your cursor in a text field and press the right paste key.

The magenta keys are up/down/left/right arrows for navigation.

Be careful with the purple key, it's the delete key!

And the cyan key is the spacebar.
Customizing the Code

There are nearly limitless customization you can do to make your ItsyBitsy Keybow work the way you want! Here are a few to try:

- color
- key assignments
- orientation
- animation
Keybow Orientation

We can use the Keybow in a vertical or horizontal orientation by re-assigning the relationship in code between the physical pins and their logical representations. This means that we can rotate the board by only changing a single variable and the key colors and assignments will update automatically.

So, for example, if we use this assignment in vertical orientation:

```
a b c
d e f
g h 1
2 3 4
```

It will become this when set to horizontal orientation:

```
a b c d
e f g h
1 2 3 4
```

This is accomplished by setting the `orientation` variable to 0 or 1, which then chooses which `pins` list to use. These set the logical order from 0-11 to use the physical pins on the ItsyBitsy that are wired via the ProtoBonnet to the Keybow.

```python
tonishment = 0  # 0 = portrait/vertical, 1 = landscape/horizontal

# Pin definitions
if orientation == 0:  # 0 = portrait/vertical
    pins = [board.D11,
             board.D12,
             board.D2,
             board.D10,
             board.D1,
             board.D7,
             board.A5,
             board.A4,
             board.A3,
             board.A6,
             board.A10,
             board.A8]
```

©Adafruit Industries
Here's an example of the vertical orientation and some alternate keycaps.

The `orientation` variable also is used to remap the DotStar physical order to the appropriate logical order.

The physical order of the DotStar LEDs can be seen in the diagram to the left.

So, to use these with our keymap ordering, the code below is used, with the if `orientation == 1:` list:
if orientation == 0:
    key_dots = [0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11]
    # 0  #4  #8
    # 1  #5  #9
    # 2  #6  #10
    # 3  #7  #11
if orientation == 1:
    key_dots = [3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8]
    # 3  #2  #1  #0
    # 7  #6  #5  #4
    # 11 #10 #9  #8

USB keyboards and mice show up on your computer as 'HID' devices, which stands for 'Human Interface Device'

HID Keyboard Basics

This guide page has a great intro to CircuitPython HID Keyboard.

For even more details, check out the documentation at https://circuitpython.readthedocs.io/projects/hid/en/latest/ which includes all of the keycodes and media codes you can use.

By importing the adafruit_hid library into our program we can make calls to send keyboard keys and media keys.

Keyboard Press/Release

Using the HID library in CircuitPython, we can send this command to "type" the letter 'a':

```python
kbd.press(Keycode.A)
```

```python
kbd.release(Keycode.A)
```

This would send a lowercase 'a' to the computer just as if you had typed it yourself. To send a capital 'A', we'd add the shift key to the command like this:

```python
kbd.press(Keycode.SHIFT, Keycode.A)
```

```python
kbd.release(Keycode.SHIFT, Keycode.A)
```

This is pretty cool, since it means we can layer on lots of keys all at the same time, just like you do on your physical keyboard when using keyboard shortcuts!
So, if there's some keyboard shortcut you want to use (or create for yourself in something like Quicksilver or AutoKeys) that is command+option+ctrl+a the CircuitPython code would look like this:

```python
kbd.press(Keycode.GUI, Keycode.ALT, Keycode.CONTROL, Keycode.A)
```

```python
kbd.release(Keycode.GUI, Keycode.ALT, Keycode.CONTROL, Keycode.A)
```

The adafruit_hid library allows for operating system specific names such as 'Keycode.COMMAND' on macOS which is 'Keycode.WINDOWS' on Windows. Or, you can use the generic 'Keycode.GUI' on any operating system. Same goes for 'ALT/OPTION'.

### Media Control

There is a second command we'll use when we want to adjust volume, play/pause, skip tracks, and so on with media such as songs and videos. These are often represented on a physical keyboard as icons silkscreened onto the rightmost function keys.

In USB HID speak, these are known as "Consumer Control codes". To play or pause a track we'll use this command:

```python
cc.send(ConsumerControlCode.PLAY_PAUSE)
```
Key Assignements

With that in mind, you can now fully customize the function of each key by editing this section of the code:

```
keymap = {
    (0): (AMBER, MEDIA, ConsumerControlCode.PLAY_PAUSE),
    (1): (AMBER, MEDIA, ConsumerControlCode.MUTE),
    (2): (AMBER, MEDIA, ConsumerControlCode.VOLUME_DECREMENT),
    (3): (AMBER, MEDIA, ConsumerControlCode.VOLUME_INCREMENT),
    (4): (BLUE, KEY, (Keycode.GUI, Keycode.C)),
    (5): (BLUE, KEY, (Keycode.GUI, Keycode.V)),
    (6): (MAGENTA, KEY, [Keycode.UP_ARROW]),
    (7): (PURPLE, KEY, [Keycode.BACKSPACE]),
    (8): (BLUE, KEY, [Keycode.SPACE]),
    (9): (MAGENTA, KEY, [Keycode.LEFT_ARROW]),
    (10): (MAGENTA, KEY, [Keycode.DOWN_ARROW]),
    (11): (MAGENTA, KEY, [Keycode.RIGHT_ARROW]),
}
```

Note, in order to use a single stroke keycode, you'll surround it in [brackets], while a multi-stroke keycode will have its own (parentheses) as shown here:

```
(4): (BLUE, KEY, [Keycode.C]),
(5): (BLUE, KEY, (Keycode.SHIFT, Keycode.C)),
```

Since we can use the orientation variable as shown above, the keymap values pertain to the keys starting at the origin key in the upper left corner of the board and moving left to right, top to bottom toward the last key in the lower right corner in either case.

Colors

On easy way to customize colors is to simply change the `keymap` color assignments, or, you can create your own colors to augment this list:

```
RED = 0xFF0000
AMBER = 0xAA9900
BLUE = 0x0066FF
MAGENTA = 0xFF00FF
PURPLE = 0x3B0F85
BLACK = 0x000000
```

Animation

In this code we're doing a simple animation at startup by calling the `dot_on()` function with different color settings.
However, if you want to get HIGHLY FANCY, -- and who doesn't?! -- you can use the adafruit_led_animation library to create beautiful effects. Here's a great guide for getting started with [CircuitPython LED Animation](https://circuitpython.org/tutorial/ledanimation).

Here's an example of some adafruitLedAnimation library sequences running on the ItsyBitsy Keybow's DotStars. (Note, this is an animation example only and doesn't have key functions.)

```python
import board
from adafruit_led_animation.sequence import AnimationSequence
from adafruit_led_animation.animation.comet import Comet
from adafruit_led_animation.animation.chase import Chase
from adafruit_led_animation.animation.blink import Blink
from adafruit_led_animation.helper import PixelMap
from adafruit_led_animation.color import RED, BLUE, GREEN
import adafruit_dotstar as dotstar

import board

import board
from adafruit_led_animation.sequence import AnimationSequence
from adafruit_led_animation.animation.comet import Comet
from adafruit_led_animation.animation.chase import Chase
from adafruit_led_animation.sequence import AnimationSequence
from adafruit_led_animation.animation.blink import Blink
from adafruit_led_animation.helper import PixelMap
from adafruit_led_animation.color import RED, BLUE, GREEN
import adafruit_dotstar as dotstar

import board

dots = dotstar.DotStar(board.SCK, board.MOSI, 12, brightness=0.85, auto_write=False)

def pixel_grid(dots):
    return PixelMap(dots, 
        [3, 2, 1, 0], 
        [7, 6, 5, 4], 
        [11, 10, 9, 8],
        individual_pixels=True)

blink = Blink(pixel_grid, speed=0.5, color=BLUE)
comet = Comet(pixel_grid, speed=0.03, color=RED, tail_length=12)
chase = Chase(pixel_grid, speed=0.1, color=GREEN, size=1, spacing=3)
animations = AnimationSequence(speed=0.1, color=GREEN, size=1, spacing=3)

while True:
    animations.animate()
```

©Adafruit Industries