# Table of Contents

## Overview
- Parts

## CircuitPython
- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

## Project Code
- Text Editor
- Download the Project Bundle

## Custom Configurations

## Mouse and Media Controls
- Media Control ("Consumer Control" codes)
- Mouse Control
- Experimental Stuff

## Going Further
Overview

When you have a key-festooned unit called a MACROPAD, it’s only natural that one of the first things to try would be application hotkeys or macros. Anything less would be like a dinosaur tour without any dinosaurs!

Press one of MACROPAD’s 12 keys to send a shortcut, function key or whole sequence of keystrokes to a connected computer. The OLED display provides a map, while LEDs under each key offer color-coded groups or themes. Turn the dial to select among different application sets.

This is one of those projects that you can simply find everyday use for as-is, or peer inside the code to see how CircuitPython makes this all pretty simple. Additionally, hotkey configuration files for different desktop applications are easily created, modified and shared.

Parts

You can buy a kit with all the parts with Kailh Red keys and clear keycaps or build your own custom configuration:
Adafruit MacroPad RP2040 Starter Kit - 3x4 Keys + Encoder + OLED
Strap yourself in, we're launching in T-minus 10 seconds...Destination? A new Class M planet called MACROPAD! M here stands for Microcontroller because this 3x4 keyboard controller...
https://www.adafruit.com/product/5128

- or -

Adafruit MACROPAD RP2040 Bare Bones - 3x4 Keys + Encoder + OLED
Strap yourself in, we're launching in T-minus 10 seconds...Destination? A new Class M planet called MACROPAD! M here, stands for Microcontroller because this 3x4 keyboard...
https://www.adafruit.com/product/5100

Adafruit MacroPad RP2040 Enclosure + Hardware Add-on Pack
Dress up your Adafruit Macropad with PaintYourDragon's fabulous decorative silkscreen enclosure and hardware kit. You get the two custom PCBs that are cut to act as a protective...
https://www.adafruit.com/product/5103
Kailh Mechanical Key Switches - 10 packs
-Cherry MX Compatible
For crafting your very own custom keyboard, these Kailh mechanical key switches are deeee-luxe! Come in a pack of 10 switches, plenty to make a...
https://www.adafruit.com/product/4996

Clear DSA Keycaps for MX Compatible Switches - 10 pack
Get ready to clacky to your heart's content. Here is a 10 pack of clear transparent DSA keycaps for your next mechanical keyboard or
https://www.adafruit.com/product/5013

Translucent Keycaps for MX Compatible Switches - 10 pack
Get ready to clacky to your heart's content. Here is a 10 pack of translucent keycaps for your next mechanical keyboard or
https://www.adafruit.com/product/4956

CircuitPython

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.
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

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

Save it wherever is convenient for you.

The BOOT button is the button switch in the rotary encoder! To engage the BOOT button, simply press down on the rotary encoder.

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 doesn't even show up as a disk drive when installing CircuitPython, try loading this 'nuke' UF2 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

Project Code

Text Editor

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

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, along with a folder full of key configuration 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's CIRCUITPY drive, replacing any existing files or directories with the same names, and adding any new ones that are necessary.

Inside the macros folder, you'll likely want to pluck out any macro settings files to discard. They're prefixed with "mac" and "win" for MacOS and Windows since these systems have different modifier keys. These files are just a starting point for reference...more likely, you'll start creating your own.

If updating from a prior version of this code, move any key configuration files that you've created or edited to a safe place first, so they're not overwritten or lost, then move them back to the CIRCUITPY/macros folder after updating.

```python
# SPDX-FileCopyrightText: 2021 Phillip Burgess for Adafruit Industries
# SPDX-License-Identifier: MIT

===
A macro/hotkey program for Adafruit MACROPAD. Macro setups are stored in the
/macros folder (configurable below), load up just the ones you're likely to
use. Plug into computer's USB port, use dial to select an application macro
set, press MACROPAD keys to send key sequences and other USB protocols.
===

# pylint: disable=import-error, unused-import, too-few-public-methods
import os
import time
import displayio
import terminalio
from adafruit_display_shapes.rect import Rect
from adafruit_display_text import label
from adafruit_macropad import MacroPad

# CONFIGURABLES  ------------------------
MACRO_FOLDER = '/macros'

# CLASSES AND FUNCTIONS  ------------------
class App:
    """Class representing a host-side application, for which we have a set
    of macro sequences. Project code was originally more complex and
    this was helpful, but maybe it's excessive now?""
def __init__(self, appdata):
    self.name = appdata['name']
    self.macros = appdata['macros']
```
def switch(self):
    """ Activate application settings; update OLED labels and LED colors. """
    group[13].text = self.name  # Application name
    for i in range(12):
        if i < len(self.macros):  # Key in use, set label + LED color
            macropad.pixels[i] = self.macros[i][0]
            group[i].text = self.macros[i][1]
        else:  # Key not in use, no label or LED
            macropad.pixels[i] = 0
            group[i].text = ''
    macropad.keyboard.release_all()
    macropad.consumer_control.release()
    macropad.mouse.release_all()
    macropad.stop_tone()
    macropad.pixels.show()
    macropad.display.refresh()

    # INITIALIZATION -----------------------
    macropad = MacroPad()
    macropad.display.auto_refresh = False
    macropad.pixels.auto_write = False

    # Set up displayio group with all the labels
    group = displayio.Group()
    for key_index in range(12):
        x = key_index % 3
        y = key_index // 3
        group.append(label.Label(terminalio.FONT, text='', color=0xFFFFFF,
                                anchored_position=((macropad.display.width - 1) * x / 2,
                                macropad.display.height - 1 - (3 - y) * 12),
                                anchor_point=(x / 2, 1.0)))
        group.append(Rect(0, 0, macropad.display.width, 12, fill=0xFFFFFF))
        group.append(label.Label(terminalio.FONT, text='', color=0x000000,
                                 anchored_position=(macropad.display.width/2, -2),
                                 anchor_point=(0.5, 0.0)))
    macropad.display.show(group)

    # Load all the macro key setups from .py files in MACRO_FOLDER
    apps = []
    files = os.listdir(MACRO_FOLDER)
    files.sort()
    for filename in files:
        if filename.endswith('.py') and not filename.startswith('._'):
            try:
                module = __import__(MACRO_FOLDER + '/' + filename[:-3])
                apps.append(App(module.app))
            except (SyntaxError, ImportError, AttributeError, KeyError, NameError,
                    IndexError, TypeError) as err:
                print("ERROR in", filename)
                import traceback
                traceback.print_exception(err, err, err.__traceback__)
        if not apps:
            group[13].text = 'NO MACRO FILES FOUND'
            macropad.display.refresh()
    while True:
        pass

    last_position = None
    last_encoder_switch = macropad.encoder_switch_debounced.pressed
    app_index = 0
    apps[app_index].switch()
# MAIN LOOP ----------------------------

```python
while True:
    # Read encoder position. If it's changed, switch apps.
    position = macropad.encoder
    if position != last_position:
        app_index = position % len(apps)
        apps[app_index].switch()
        last_position = position

    # Handle encoder button. If state has changed, and if there's a
    # corresponding macro, set up variables to act on this just like
    # the keypad keys, as if it were a 13th key/macro.
    encoder_switch = macropad.encoder_switch_debounced.pressed
    if encoder_switch != last_encoder_switch:
        last_encoder_switch = encoder_switch
        if len(apps[app_index].macros) < 13:
            continue    # No 13th macro, just resume main loop
        key_number = 12 # else process below as 13th macro
        pressed = encoder_switch
    else:
        event = macropad.keys.events.get()
        if not event or event.key_number >= len(apps[app_index].macros):
            continue # No key events, or no corresponding macro, resume loop
        key_number = event.key_number
        pressed = event.pressed

    # If code reaches here, a key or the encoder button WAS pressed/released
    # and there IS a corresponding macro available for it...other situations
    # are avoided by 'continue' statements above which resume the loop.

    sequence = apps[app_index].macros[key_number][2]
    if pressed:
        # 'sequence' is an arbitrary-length list, each item is one of:
        # Positive integer (e.g. Keycode.KEYPAD_MINUS): key pressed
        # Negative integer: (absolute value) key released
        # Float (e.g. 0.25): delay in seconds
        # String (e.g. "Foo"): corresponding keys pressed & released
        # List []: one or more Consumer Control codes (can also do float delay)
        # Dict {}: mouse buttons/motion (might extend in future)
        if key_number < 12: # No pixel for encoder button
            macropad.pixels[key_number] = 0xFFFFFF
        macropad.pixels.show()
        for item in sequence:
            if isinstance(item, int):
                if item >= 0:
                    macropad.keyboard.press(item)
                else:
                    macropad.keyboard.release(-item)
            elif isinstance(item, float):
                time.sleep(item)
            elif isinstance(item, str):
                macropad.keyboard_layout.write(item)
            elif isinstance(item, list):
                for code in item:
                    if isinstance(code, int):
                        macropad.consumer_control.release()
                        macropad.consumer_control.press(code)
                    elif isinstance(code, float):
                        time.sleep(code)
            elif isinstance(item, dict):
                if 'buttons' in item:
                    if item['buttons'] >= 0:
                        macropad.mouse.press(item['buttons'])
                    else:
                        macropad.mouse.release(-item['buttons'])
                macropad.mouse.move(item['x'] if 'x' in item else 0,
```
item['y'] if 'y' in item else 0,
item['wheel'] if 'wheel' in item else 0)

if 'tone' in item:
    if item['tone'] > 0:
        macropad.stop_tone()
        macropad.start_tone(item['tone'])
    else:
        macropad.stop_tone()
elif 'play' in item:
    macropad.play_file(item['play'])
else:
    # Release any still-pressed keys, consumer codes, mouse buttons
    # Keys and mouse buttons are individually released this way (rather
    # than release_all()) because pad supports multi-key rollover, e.g.
    # could have a meta key or right-mouse held down by one macro and
    # press/release keys/buttons with others. Navigate popups, etc.
    for item in sequence:
        if isinstance(item, int):
            if item >= 0:
                macropad.keyboard.release(item)
        elif isinstance(item, dict):
            if 'buttons' in item:
                if item['buttons'] >= 0:
                    macropad.mouse.release(item['buttons'])
            else:
                if 'tone' in item:
                    macropad.stop_tone()
                    macropad.consumer_control.release()
        if key_number < 12: # No pixel for encoder button
            macropad.pixels[key_number] = apps[app_index].macros[key_number][0]
            macropad.pixels.show()
```python
app = {
    'name': 'Mac Safari',  # REQUIRED dict, must be named 'app'
    'macros': [
        # List of button macros...
        # COLOR    LABEL    KEY SEQUENCE
        # 1st row ----------
        (0x004000, '< Back', [Keycode.COMMAND, '[']),
        (0x004000, 'Fwd >', [Keycode.COMMAND, ']')),  # Scroll up
        # 2nd row ----------
        (0x202000, '< Tab', [Keycode.CONTROL, Keycode.SHIFT, Keycode.TAB]),
        (0x202000, 'Tab >', [Keycode.CONTROL, Keycode.TAB]),
        (0x400000, 'Down', ' '),  # Scroll down
        # 3rd row ----------
        (0x000040, 'Reload', [Keycode.COMMAND, 'r']),
        (0x000040, 'Home', [Keycode.COMMAND, 'H']),
        (0x000040, 'Private', [Keycode.COMMAND, 'N']),
        # 4th row ----------
        (0x000000, 'Ada', [Keycode.COMMAND, 'n', -Keycode.COMMAND, 'www.adafruit.com
']),  # Adafruit in new window
        (0x800000, 'Digi', [Keycode.COMMAND, 'n', -Keycode.COMMAND, 'www.digikey.com
']),  # Digi-Key in new window
        (0x101010, 'Hacks', [Keycode.COMMAND, 'n', -Keycode.COMMAND, 'www.hackaday.com
']),  # Hack-a-Day in new win
        # Encoder button ---
        (0x000000, '', [Keycode.COMMAND, 'w'])  # Close window/tab
    ]
}
```

This is just a single Python dictionary (which must be named 'app' for the project code to find it) containing two keys: 'name' and 'macros'.

'name' is what's shown across the top of MACROPAD's display when turning the knob to switch settings. This must be a short string in quotes, no more than 20 characters, for example 'Safari'.

'macros' is a list [enclosed in square brackets] of tuples (each enclosed in parenthesis) — one for each of MACROPAD's 12 keys, and optionally one more for the encoder button. These appear sequentially, first item for the top-left key, second for top-center, and so forth. The examples have some Python comments to explain what's happening.

Each of these tuples contains three elements:

1. A hexadecimal RGB color value for the corresponding LED. You may want to avoid bright values, as a whole keypad of these can be distracting. If you like, color-code related groups of keys, or theme whole applications so you can tell what's active at a glance.
2. A brief description, in quotes. This is what's displayed on MACROPAD's OLED screen. BRIEF is the operative word here, ideally six characters or less. You can sometimes sneak in a 7-character label if adjoining items are shorter.
3. A key sequence, which can either be:
   1. A single character or a string, in quotes, if the key sequence is just regular keypresses (including SHIFT for uppercase characters).
2. A list [enclosed in square brackets] of key code constants and quoted-strings, which will be issued in-order.

Most letters, numbers and symbols are best done as quoted strings...but for special navigation-type keys...arrows, COMMAND/CONTROL/OPTION, function keys and so forth...key constants are required. We can see a complete list of these by typing in the CircuitPython REPL:

```python
>>> from adafruit_hid.keycode import Keycode
>>> print(dir(Keycode))
```

The Safari example was chosen because it demonstrates most permutations quite nicely...

The third key, “Up,” for example...in Safari, this is done with SHIFT+SPACE. There’s no such thing as an “uppercase space,” so we can’t just use a quoted string here. Keycode.SHIFT is a constant telling the code to press and hold the SHIFT key, and then the quoted space is issued. SHIFT is automatically released at the end of the sequence.

You can see other keys doing similar operations, sometimes with Keycode.COMMAND or sometimes multiple modifiers (“Previous Tab,” for example, is CONTROL+SHIFT+TAB).

The bottom three keys show how to press and release keys mid-sequence. Each of these opens a new window (COMMAND+n) and enters a URL. We want the COMMAND key released after pressing 'n', so the characters in the URL string type normally, not as list of commands. A negative Keycode value is used to indicate “release this key now”: -Keycode.COMMAND in the example.

Instead of a Keycode or string, a floating-point number inserts a pause in the key-pressing sequence. The duration is in seconds, either whole or partial. For example:
This would press (and keep held) the 'Control' key, pause one half second, type an “a,” pause one second, then type a “b” (Control is implicitly released at the end of the sequence). Even if pausing for whole seconds, the decimal must be included...this is how it distinguishes from Keycodes, which are integer values.

Though you could store passwords in there, this is strongly discouraged, since the CircuitPython code is not protected or secure. Anyone with access to your MACROPAD can read these files!

---

**Mouse and Media Controls**

It started simply enough with a few key presses, but folks had bigger ideas for things to automate. Could we work in media controls like play/pause and volume, or mouse control?

The adafruit_hid library certainly provides access to such things, but a bigger question was: could these be added without breaking compatibility with all the existing macro files that users have created and shared? The first problem is simple and the code was extended to handle it, but to handle this second problem, the approach is a little obtuse. I apologize, but this seemed the best way to keep those existing files working and not require a do-over.

It relies on the fact that these macro files aren’t simply structured text like XML or JSON, they’re snippets of actual Python code. Python can distinguish between different data types — integers, lists, dictionaries and so forth — so we leverage that to distinguish key codes from media control, mouse and so forth.

Consider that the last item for each entry in a macro file is normally either a quoted string (which gets typed verbatim), or a list of Keycodes and/or strings in a set of square brackets [ ... ], like so:

```
(0x000040, 'Brush', 'B'),
(0x004000, 'Undo', [Keycode.CONTROL, 'z'])),
```

Along the way, it was decided that timed pauses were sometimes needed, and (as shown on the prior page) these could be inserted in that bracketed list as floating-point numbers. Keycode values are really just integers, and CircuitPython can distinguish the two data types:

```
[Keycode.CONTROL, 0.5, 'a', 1.0, 'b']
```
So...to handle different controls, we just extended this types-in-list distinction to more types.

**Media Control (“Consumer Control” codes)**

For media functions like play/pause, volume or brightness your macro file should first import the ConsumerControlCode values. The example macro file media.py demonstrates this.

```python
from adafruit_hid.consumer_control_code import ConsumerControlCode
```

Like HID Keycodes, ConsumerControl codes are also just integers. But there are a few ranges where they may overlap. So, to distinguish between the two, Consumer Control codes go in a `[]` list nested inside the Keycode list, like this:

```python
[[ConsumerControlCode.SCAN_NEXT_TRACK]]
```

(Notice the double brackets: `[[ ... ]]`)

If you need to get fancy, it’s possible to mix Keycodes, strings and/or Consumer Control codes in the same list (delays, too).

```python
[Keycode.CONTROL, [ConsumerControlCode.SCAN_NEXT_TRACK], -Keycode.CONTROL, 1.0, 'foo']
```

(Press Control key, activate “next track,” release Control key, pause one second, type the string “foo”.)

For a complete list of Consumer Control codes, you can type in the CircuitPython REPL:

```python
>>> from adafruit_hid.consumer_control_code import ConsumerControlCode
>>> print(dir(ConsumerControlCode))
['__class__', '__module__', '__name__', '__qualname__', '__bases__', 'RECORD', 'FAST_FORWARD', 'REWIND', 'SCAN_NEXT_TRACK', 'SCAN_PREVIOUS_TRACK', 'STOP', 'EJECT', 'PLAY_PAUSE', 'MUTE', 'VOLUME_DECREMENT', 'VOLUME_INCREMENT', 'BRIGHTNESS_DECREMENT', 'BRIGHTNESS_INCREMENT']
```

**Mouse Control**

For mouse-related actions like movement and button clicks, your macro file should first import the HID Mouse module. The example macro file mouse.py demonstrates this.
from adafruit_hid.mouse import Mouse

A mouse is a more complex thing, with spatial axes in addition to buttons. A bracketed list won’t do…we have to step this up to a full Python dictionary, in curly braces {}. But this appears inside the original outermost Keycode [] list.

`{'buttons':Mouse.LEFT_BUTTON}`

(Notice the bracket-brace combo: [[ ... ]])

The dictionary key (the quoted bit of text before the colon) can be one of four values: 'x', 'y', 'wheel' and 'buttons', controlling cursor movement, scroll wheel position, and button states, respectively.

The dictionary value (after the colon) depends on what's being controlled. For 'x', 'y', 'wheel', it's a signed integer for relative motion. For example, 10 to move the cursor right or down by 10 pixels, or -10 to move up or left. For 'buttons', it can be Mouse.LEFT_BUTTON, Mouse.MIDDLE_BUTTON or Mouse.RIGHT_BUTTON (you can combine multiple buttons with +) and, like Keycode, can be negative if you want to indicate a button release in the middle of a longer sequence.

And again, like with Consumer Control codes, this can be mixed with Keycodes and delays as part of a complex sequence.

Experimental Stuff

The use of Python dictionaries for mouse control really opened this up for just about anything. As an experiment, the ability to play tones through MACROPAD’s onboard speaker was added. The example macro file tones.py demonstrates this. Not very useful on its own, but when used in combination with keyboard and mouse actions this could provide some audio feedback to each macro.

This plays a 262 Hz tone (roughly middle-C-ish) for as long as the corresponding key is held:

`{'tone':262}`

Only one tone can play at a time. These can be mixed in the [] macro list with other types. A value of 0 (or releasing the key) stops any currently-playing tone. For example, this will play middle-C for one half second:

`{'tone':262}, 0.5, {'tone':0}`
Going Further

Once it’s all started up, the code has little work to do...just watch for key press events and encoder movement, taking some action when necessary.

If one were so inclined, the hardware and code could be extended for extra functionality. MACROPAD features a Stemma QT connector on the side, which might accommodate some interesting sensors or a NeoKey 1x4 QT () for four extra key switches. The code as written does not handle any of this automatically... the additions would be a software project of your own.

Our hotkeys code uses MACROPAD’s encoder wheel to select among different application settings. It’s usually fine, but if you’re a big multitasker it can be awkward when the wrong app is selected (which is why color-coding with the LEDs is recommended, for at-a-glance familiarity).

Could there be some way to automatically switch based on the current application in use? CircuitPython can receive serial messages while also emulating a keyboard, so there’s ways to send information to MACROPAD. The host-side implementation though, that gets complex, and would vary with all the myriad system types and their particular scripting or development options, which is why it’s not done here. Food for thought!