MacroPad Remote Procedure Calls over USB to Control Home Assistant

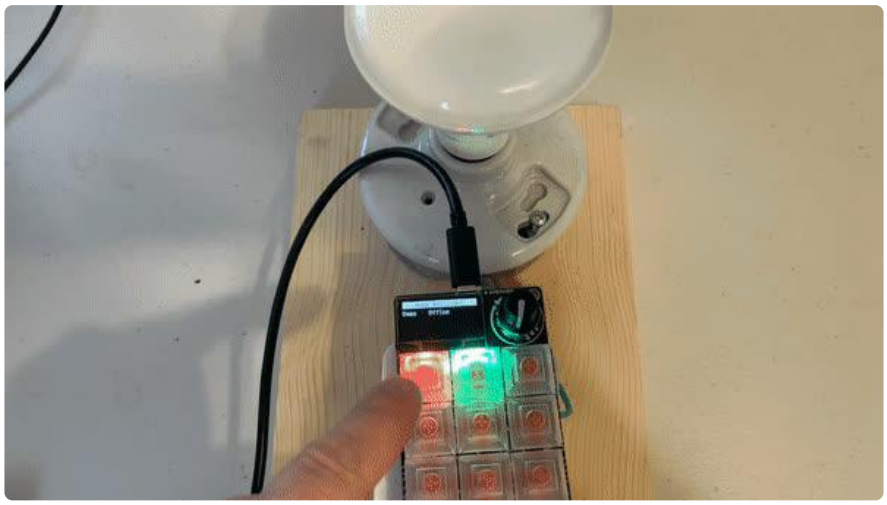
Created by Melissa LeBlanc-Williams

https://learn.adafruit.com/macropad-remote-procedure-calls-over-usb-to-control-home-assistant

Last updated on 2022-12-01 04:06:33 PM EST

Table of Contents

Overview

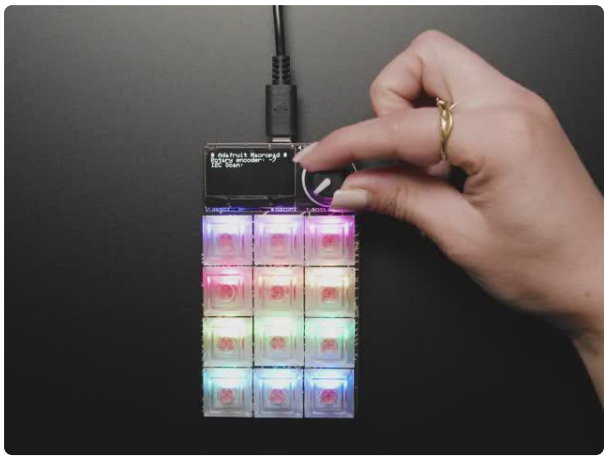


Ok, let's be honest. The Adafruit MacroPad is just awesome! But it could be better if you could control things over the Internet or local network. Fear not, for there is a way. With the magic of Remote Procedure Calls, you can have your computer to do all of the networking stuff and return the results by telling it what you want over USB Serial and receiving the results back.

This project connects to Home Assistant to control lights using automations. To display the current state of the lights, the MacroPad will be publishing its control changes, and it will have the host computer subscribe to the MQTT topics of the devices to grab their state and change the color of the keys accordingly.

Since the MacroPad has a rotary encoder, this project uses it to control device dimming. Since the state of the lights and the brightness of the light can usually be controlled by the switch itself and the controls in Home Assistant, in addition to the MacroPad, it made sense to only send the MacroPad state changes, such as the rotary encoder changes, rather than trying to keep track of the state on the MacroPad itself.

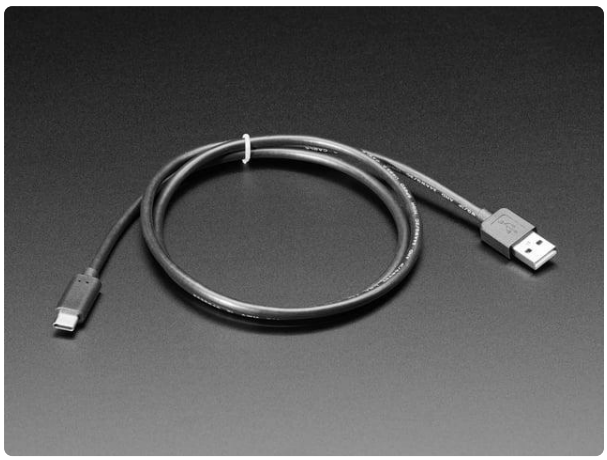
Parts



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



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

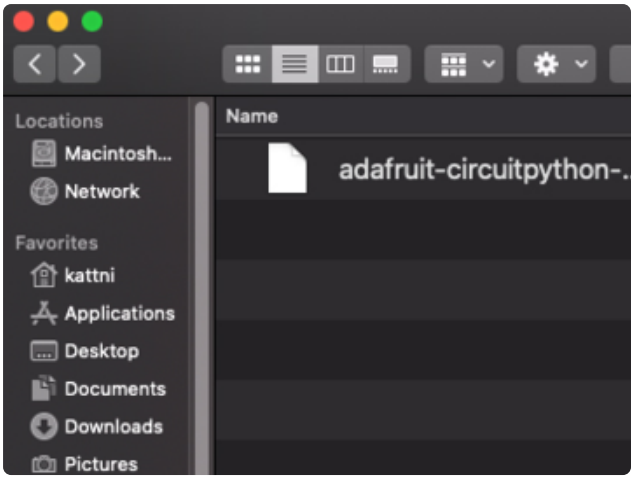
CircuitPython

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

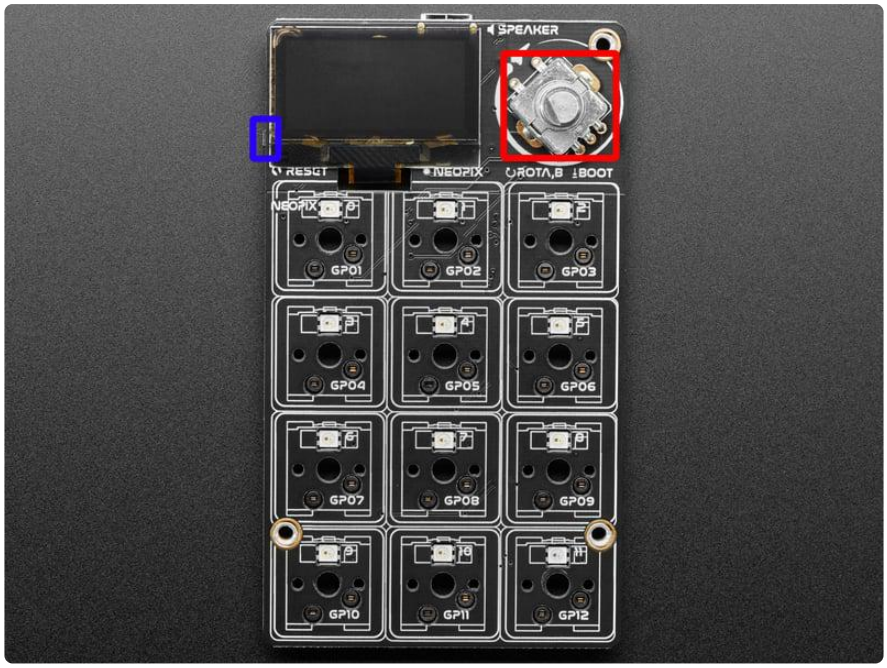
Follow this step-by-step to quickly get CircuitPython running on your board.

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.



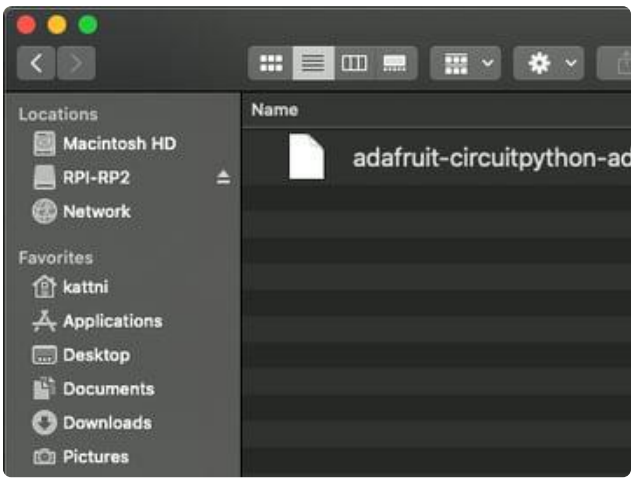
The BOOT button is the button switch in the rotary encoder! To engage the BOOT button, simply press down on the rotary encoder.

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!

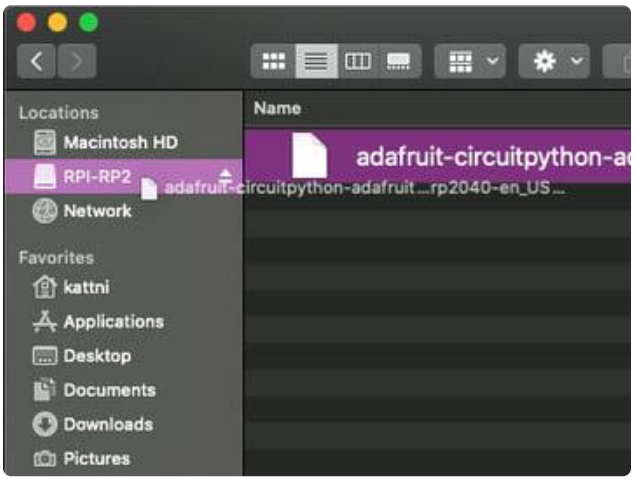
If the drive does not appear, release all the buttons, and then repeat the process above.

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.

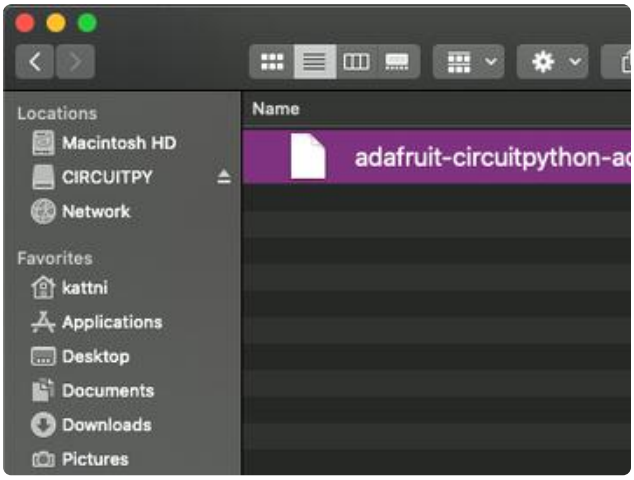
A lot of people end up using charge-only USB cables and it is very frustrating! Make sure you have a USB cable you know is good for data sync.



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 in CircuitPython 6.x

This section explains entering safe mode on CircuitPython 6.x.

To enter safe mode when using CircuitPython 6.x, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 700ms. On some boards, the onboard status LED (highlighted in green above) will turn solid yellow during this time. If you press reset during that 700ms, 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.)

Entering Safe Mode in CircuitPython 7.x

This section explains entering safe mode on CircuitPython 7.x.

To enter safe mode when using CircuitPython 7.x, 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

Once you've entered safe mode successfully in CircuitPython 6.x, the LED will pulse yellow.

If you successfully enter safe mode on CircuitPython 7.x, 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

MacroPad Setup

Once you have CircuitPython installed, the first thing you will need to do is to enable the CDC Data device. CDC stands for "Communications Device Class" and is a USB term. When a CircuitPython device is booted up, by default it normally comes with a CDC Console Device enabled, which is awesome for debugging, but it can introduce special characters into the stream. The Data device is an additional serial device that can be enabled that overcomes this issue. You can read more about it in our Customizing USB Devices in CircuitPython () guide.

To enable the CDC Data device, you just need to add the following into a boot.py file on the root level of the CIRCUITPY drive:

```
import usb_cdc

usb_cdc.enable(console=True, data=True)
```

Secrets File

Like other network enabled devices, such as the Adafruit PyPortal or MagTag, this project uses a secrets file. This will contain the MQTT connection information that will be used to connect to your Home Assistant MQTT server. If you have done any of the other Adafruit HomeAssistant projects, you should just be able to copy over an existing secrets.py file. If you haven't you can just create a secrets.py file at the root level of your CIRCUITPY drive with the following content:

```
# This file is where you keep secret settings, passwords, and tokens!
# If you put them in the code you risk committing that info or sharing it

secrets = {
    'ssid' : 'home_wifi_network',
    'password' : 'wifi_password',
    'aio_username' : 'my_adafruit_io_username',
    'aio_key' : 'my_adafruit_io_key',
    'timezone' : "America/New_York", # http://worldtimeapi.org/timezones
    'mqtt_broker': "192.168.1.1",
    'mqtt_port': 1883,
    'mqtt_username': 'my_mqtt_username',
    'mqtt_password': 'my_mqtt_password',
}
```

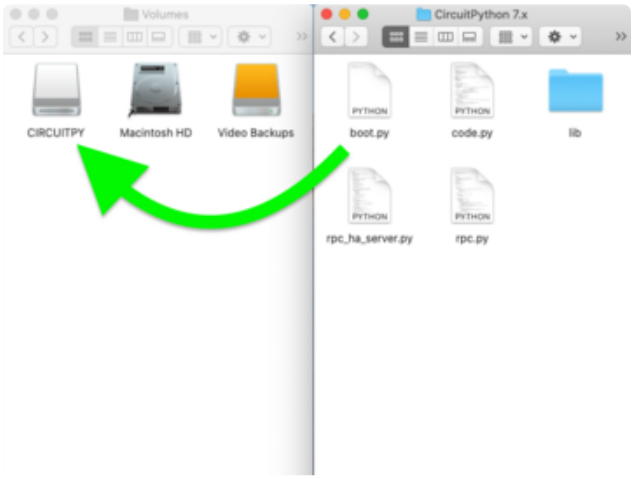
The only items you really need to change the values on are the MQTT parameters.

If your MQTT Server only allows connecting to port 8883, it won't currently work since the Mini MQTT library doesn't currently support SSL over CPython.

Download the Project Bundle

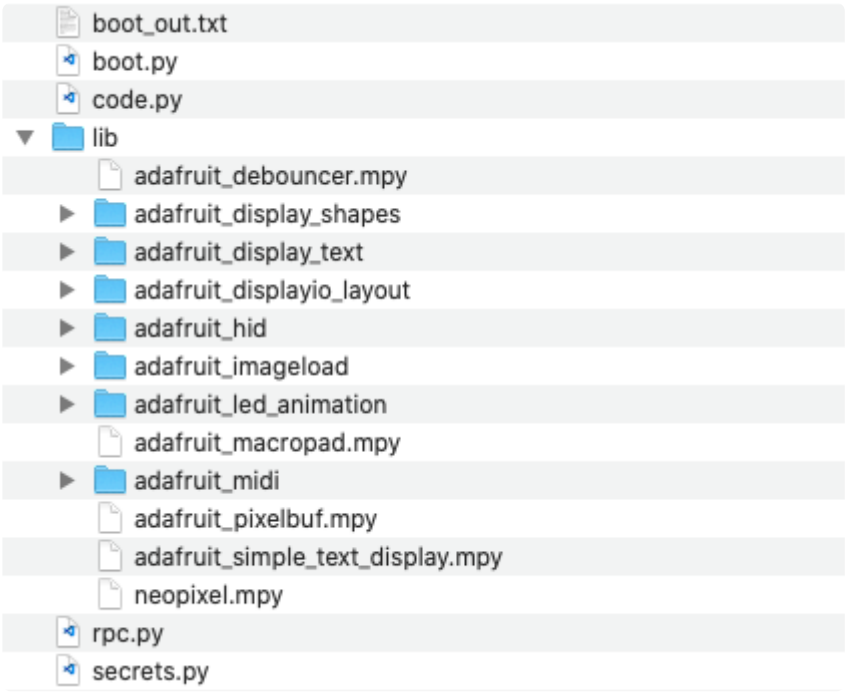
Your project will use a specific set of CircuitPython libraries as well as the code.py and rpc.py files. In order to get the libraries you need, click on the Download Project Bundle link below, and decompress the .zip file.

You can skip copying rpc_ha_server.py since that will be used on your host computer in the next step.



Next, drag the contents of the CircuitPython 7.x folder in the uncompressed bundle directory onto you microcontroller board's CIRCUITPY drive, replacing any existing files or directories with the same names, and adding any new ones that are necessary.

The files on your MacroPad should look like this:



```
# SPDX-FileCopyrightText: Copyright (c) 2021 Melissa LeBlanc-Williams for Adafruit
Industries
#
# SPDX-License-Identifier: Unlicense
"""
Home Assistant Remote Procedure Call for MacroPad.
"""
import time
import displayio
import terminalio
from adafruit_display_shapes.rect import Rect
from rpc import RpcClient, RpcError
from adafruit_display_text import label
from adafruit_macropad import MacroPad
from secrets import secrets

macropad = MacroPad()
rpc = RpcClient()

COMMAND_TOPIC = "macropad/peripheral"
SUBSCRIBE_TOPICS = ("stat/demoswitch/POWER", "stat/office-light/POWER")
ENCODER_ITEM = 0
KEY_LABELS = ("Demo", "Office")
UPDATE_DELAY = 0.25
NEOPIXEL_COLORS = {
    "OFF": 0xff0000,
    "ON": 0x00ff00,
}

class MqttError(Exception):
    """For MQTT Specific Errors"""
    pass
# 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=(str(KEY_LABELS[key_index]) if
key_index < len(KEY_LABELS) else ''), 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='Home Assistant', color=0x000000,
                         anchored_position=(macropad.display.width//2, -2),
                         anchor_point=(0.5, 0.0)))
macropad.display.show(group)

def rpc_call(function, *args, **kwargs):
    response = rpc.call(function, *args, **kwargs)
    if response["error"]:
        if response["error_type"] == "mqtt":
            raise MqttError(response["message"])
        raise RpcError(response["message"])
    return response["return_val"]

def mqtt_init():
    rpc_call("mqtt_init", secrets["mqtt_broker"], username=secrets["mqtt_username"],
password=secrets["mqtt_password"], port=secrets["mqtt_port"])
    rpc_call("mqtt_connect")

def update_key(key_number):
    if key_number < len(SUBSCRIBE_TOPICS):
        switch_state = rpc_call("mqtt_get_last_value", SUBSCRIBE_TOPICS[key_number])
        if switch_state is not None:
            macropad.pixels[key_number] = NEOPIXEL_COLORS[switch_state]
```

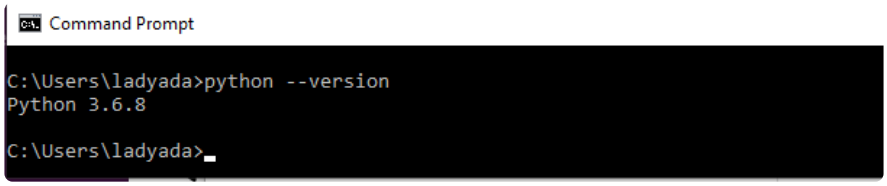
```
        else:
            macropad.pixels[key_number] = 0

server_is_running = False
print("Waiting for server...")
while not server_is_running:
    try:
        server_is_running = rpc_call("is_running")
        print("Connected")
    except RpcError:
        pass

mqtt_init()
last_macropad_encoder_value = macropad.encoder

for key_number, topic in enumerate(SUBSCRIBE_TOPICS):
    rpc_call("mqtt_subscribe", topic)
    update_key(key_number)

while True:
    output = {}

    key_event = macropad.keys.events.get()
    if key_event and key_event.pressed:
        output["key_number"] = key_event.key_number

    if macropad.encoder != last_macropad_encoder_value:
        output["encoder"] = macropad.encoder - last_macropad_encoder_value
        last_macropad_encoder_value = macropad.encoder

    macropad.encoder_switch_debounced.update()
    if macropad.encoder_switch_debounced.pressed and "key_number" not in output and
ENCODER_ITEM is not None:
        output["key_number"] = ENCODER_ITEM

    if output:
        try:
            rpc_call("mqtt_publish", COMMAND_TOPIC, output)
            if "key_number" in output:
                time.sleep(UPDATE_DELAY)
                update_key(output["key_number"])
            elif ENCODER_ITEM is not None:
                update_key(ENCODER_ITEM)
        except MqttError:
            mqtt_init()
        except RpcError as err_msg:
            print(err_msg)
```

Host Computer Setup

Have Python 3 Installed

We assume you already have Python 3 installed on your computer. Note we do not support Python 2 - it's deprecated and no longer supported!

At your command line prompt of choice, check your Python version with `python --version`



Install Required Libraries

You will need to have a few libraries installed before the script will run on your computer.

Install Adafruit_Board_Toolkit:

```
pip3 install adafruit-board-toolkit
```

Install PySerial next:

```
pip3 install pyserial
```

Install The CircuitPython Mini MQTT Library:

```
pip3 install adafruit-circuitpython-minimqtt
```

Copy rpc_ha_server.py and rpc.py onto the computer. You can either copy them out of the bundle that you downloaded in the MacroPad Setup step or if you have a Mac or Linux computer, you can use `wget` to copy them right off the web into your current folder:

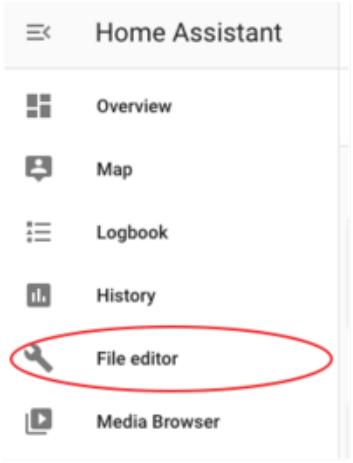
```
wget https://github.com/adafruit/Adafruit_Learning_System_Guides/raw/main/
MacroPad_RPC_Home_Assistant/rpc_ha_server.py
wget https://github.com/adafruit/Adafruit_Learning_System_Guides/raw/main/
MacroPad_RPC_Home_Assistant/rpc.py
```

Home Assistant Add-Ons

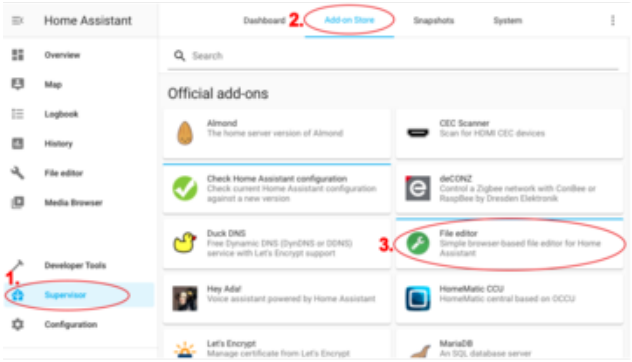
This guide assumes you already have a working and running Home Assistant server. If you don't, be sure to visit our Set up Home Assistant with a Raspberry Pi () guide first.

Check Your Add-Ons

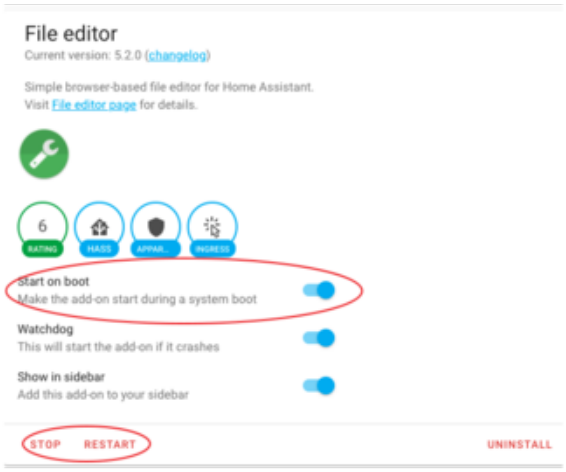
Start out by logging in and opening up your Home Assistant dashboard and checking that the File editor is installed.



As part of the setup, you should have an add-on either called configurator or File editor with a wrench icon next to it. Go ahead and select it.



If you don't see it, it may not be installed. You can find it under Supervisor → Add-on Store → File editor and go through the installation procedure.



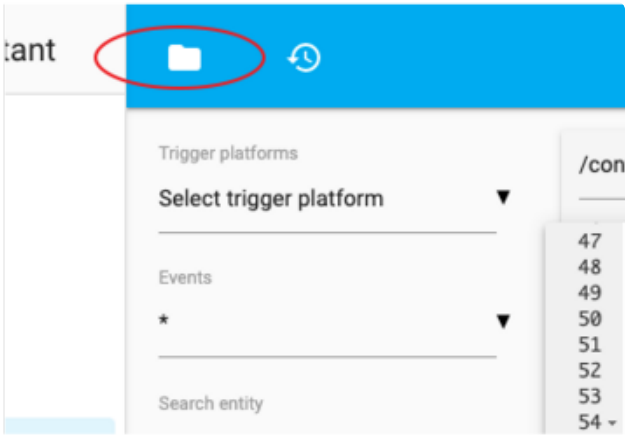
If you already have it, but it's just not showing up, be sure it is started and the option to show in the sidebar is selected.

Home Assistant Configuration

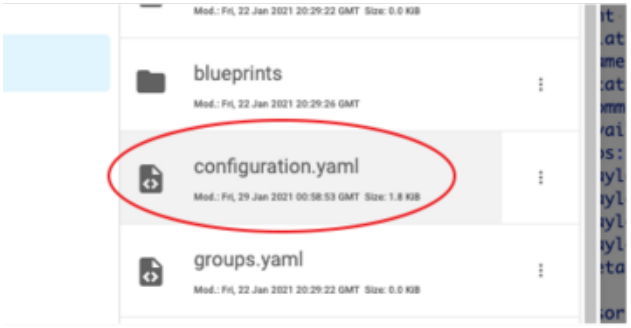
Set up your Automations

Automations are going to be highly dependent on your specific setup. In this example, we will be using a couple of devices called `office_light`, which is a switch and `test_dimmer`, which is a dimmable light. You will want to change these values to suit your specific setup. The code below provides 3 different automations to attach the events from the MacroPad to different actions and will go over those in a bit of detail.

To begin, you'll want to open up the File Editor and add some automations.



Click on the Folder Icon at the top and select configuration.yaml, then click on an area to the right of the file list to close it.



Light Toggle Automation

The first automation simply toggles the dimmer light on and off whenever it receives a keypress on key number 0 of the MacroPad, which is the upper right button.

```
automation macropad_button_0:
  alias: "Demo Light Toggle"
```

```
  trigger:
    - platform: mqtt
      topic: "macropad/peripheral"
      payload: "0"
      value_template: "{{ value_json.key_number }}"
  condition: "{{ trigger.payload_json.key_number is defined }}"
  action:
    service: light.toggle
    entity_id: light.test_dimmer
```

The `alias` is just a friendly name to display in the control panel.

Under the `trigger` section the code is set up to look for the `macropad/peripheral` topic on the `mqtt` server and trigger when it sees a value of 0. The `value_template` tells the automation where the payload value is in the JSON that is published by the MacroPad.

Under the `condition` section, The code triggers only if `key_number` is defined. This is important because when the encoder is used, there is no `key_number` defined, and it can cause some warnings in Home Assistant. Also, you may note the use of `payload_json` instead of `value_json` and that's just one of the quirks of home assistant.

Under the `action` section, it is just telling the `test_dimmer`, which is a `light`, to trigger the `light.toggle` event.

Switch Toggle Automation

The second automation simply toggles the office light on and off whenever it receives a keypress on key number 1 of the MacroPad, which is the upper center button. This is nearly identical to the light automation, so only the differences are covered.

```
automation macropad_button_1:
  alias: "Office Light Toggle"
  trigger:
    - platform: mqtt
      topic: "macropad/peripheral"
      payload: "1"
      value_template: "{{ value_json.key_number }}"
  condition: "{{ trigger.payload_json.key_number is defined }}"
  action:
    service: switch.toggle
    entity_id: switch.office_light
```

The main differences here are making use of the `switch` type of device instead of a light and the payload value waited for is 1.

Dimmer Automation

This one is the trickiest because the MacroPad is only sending the changes in the rotation. This allows other controls such Home Assistant itself to also adjust the dimmer. However, by only sending the changes, you don't need to worry grabbing the current brightness setting, modifying it, and then sending the new absolute value back. However, that would have been another way to do it.

```
automation macropad_dimmer:
  alias: "Demo Light Dimmer"
  trigger:
    - platform: mqtt
      topic: "macropad/peripheral"
      value_template: "{{ value_json.encoder }}"
  condition: "{{ trigger.payload_json.encoder is defined }}"
  action:
    service: light.turn_on
    data_template:
      entity_id: light.test_dimmer
      brightness: >
        {% set current = state_attr('light.test_dimmer', 'brightness') %}
        {% set delta = trigger.payload_json.encoder|int * 10 %}
        {{ current + delta }}
```

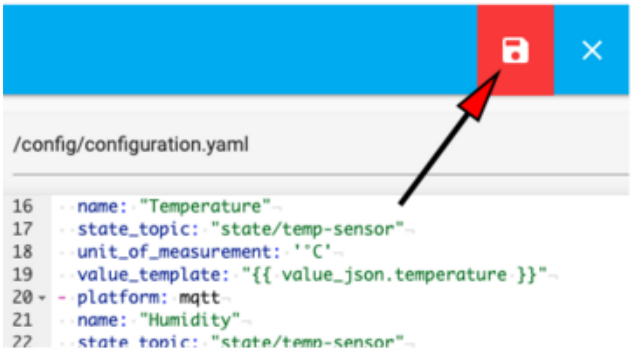
Just like before, the `alias`, `trigger`, and `condition` sections are similar, but this time there is not a specific `payload` value defined to trigger on. It will trigger based on the condition alone, which is that there is an `encoder` value defined.

Under the `action` section is where you will notice most of the differences. To adjust the brightness of the bulb, you need to make use of the `light.turn_on` service this time. In order to calculate the new brightness, we make use of templates. Templating is powered by the Jinja2 () templating engine.

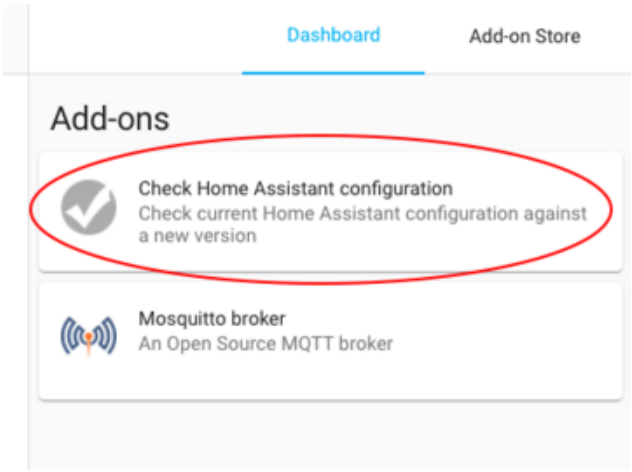
Under the `data_template`, the code tells which entity to adjust and the brightness value that it should be set to. This is calculated by taking the current value which is between 0-255, taking the delta, or change in the encoder knob, and multiplying by 10 so you don't need to crank the knob 20-30 times to get it to go from full dimness to full brightness. Then the final brightness value that it should adjust to is output.

Save Your Config

Once you're done with adding the automations to your configuration.yaml file, you'll want to restart your Home Assistant service.



Click the save button at the top.



If you have the Check Home Assistant configuration tool installed, now would be a good time to run it. It takes several minutes to run and you can check the log tab to see the results.

Once you have restarted, try pressing the top buttons on your MacroPad. They should toggle your lights. If you have a dimmer, try turning the encoder and it should dim your lights.

Troubleshooting

If you see the icons, but there is no data, it is easiest to start by checking the MQTT messages. There is a guide on how to use Desktop MQTT Client for Adafruit.io (), which can be used for the Home Assistant MQTT server as well.

Go ahead and configure a username and password to match your MQTT server and connect. Under subscribe, you can subscribe to the `#` topic to get all messages.

If you are seeing messages from the sensor, you may want to double check your Home Assistant configuration.

If you don't see any messages, you will want to follow the debugging section on the C ode the Sensor page.

Running the Code

To use everything, you will want to make sure your Home Assistant instance is up and running. Next you will want to make sure your MacroPad is running its code. You can use a serial console to see that it is "Waiting for Server".

You can then start the server by going to the folder containing the file rpc_ha_server.py and typing the following:

```
python3 rpc_ha_server.py
```

The MacroPad should connect and, if everything is configured correctly, you should be able to control your lights.

Code Walkthrough

The code is broken down into three main pieces. Because the code is a bit complex, it is separated into a separate page.

Shared RPC Library

The code walkthrough starts with the shared RPC library, because this library is the foundation that the rest of the code relies on. This library was written in such a way that it can be used with both CPython and CircuitPython, but the Server component

relies on a CPython specific library and the Client library is expecting the CDC data device to be enabled, so to make them truly work on either would require additional code.

First, it tries to import some CPython specific libraries and uses that to determine the environment that the library is running in:

```
import time
import json
try:
    import serial
    import adafruit_board_toolkit.circuitpython_serial
    json_decode_exception = json.decoder.JSONDecodeError
except ImportError:
    import usb_cdc as serial
    json_decode_exception = ValueError
```

Next are a couple of adjustable parameters for timeout values. These values seemed to work well, but feel free to adjust them if it improves performance for you.

```
RESPONSE_TIMEOUT=5
DATA_TIMEOUT=0.5
```

Next a custom `RpcError` is defined to differentiate it from other Python errors that are specific to this library.

```
class RpcError(Exception):
    """For RPC Specific Errors"""
    pass
```

Next up is the code that is shared between the libraries which is called `_Rpc` and has the underscore because the base class is not meant to be directly instantiated.

```
class _Rpc:
    def __init__(self):
        self._serial = None
```

This code will create a response packet which makes it so the receiving component will know the structure of what to expect. By having it in a function, the code can pass just the minimum of what it needs to and get a full packet out.

```
@staticmethod
def create_response_packet(error=False, error_type="RPC", message=None,
return_val=None):
    return {
        "error": error,
        "error_type": error_type if error else None,
        "message": message,
```

```
        "return_val": return_val
    }
```

The other kind of packet is the request packet to request an RPC operation.

```
@staticmethod
def create_request_packet(function, args=[], kwargs={}):
    return {
        "function": function,
        "args": args,
        "kwargs": kwargs
    }
```

The `_wait_for_packet()` function behaves slightly differently depending on whether a timeout was given or not. If timeout is `None`, it will continue to wait indefinitely until a packet is received. Otherwise it will exit the function with an error response packet if it times out.

If it doesn't time out and a packet is received, the received packet will be returned to the calling function. One other thing that this function is responsible for is understanding the type of packet it is listening for and whether it has received the entire thing.

```
def _wait_for_packet(self, timeout=None):
    incoming_packet = b""
    if timeout is not None:
    response_start_time = time.monotonic()
    while True:
        if incoming_packet:
        data_start_time = time.monotonic()
        while not self._serial.in_waiting:
            if incoming_packet and (time.monotonic() - data_start_time) &gt;= 
DATA_TIMEOUT:
            incoming_packet = b""
            if not incoming_packet and timeout is not None:
                if (time.monotonic() - response_start_time) &gt;= timeout:
                return self.create_response_packet(error=True, message="Timed out 
waiting for response")
            time.sleep(0.001)
        data = self._serial.read(self._serial.in_waiting)
        if data:
            try:
                incoming_packet += data
                packet = json.loads(incoming_packet)
                # json can try to be clever with missing braces, so make sure we 
have everything
                if sorted(tuple(packet.keys())) == sorted(self._packet_format()):
                return packet
            except json_decode_exception:
            pass # Incomplete packet
```

The first kind of class that can be created from this library is the `RpcClient`. The `RpcClient` is the component that will make the calls and listen for responses from

the `RpcServer` and is fairly straightforward because it makes use of much of the shared code covered above and `call()` is really the only unique public function.

`_packet_format()` just helps the `_wait_for_packet()` function know what type of packet it is listening for.

```
class RpcClient(_Rpc):
    def __init__(self):
        super().__init__()
        self._serial = serial.data

    def _packet_format(self):
        return self.create_response_packet().keys()

    def call(self, function, *args, **kwargs):
        packet = self.create_request_packet(function, args, kwargs)
        self._serial.write(bytes(json.dumps(packet), "utf-8"))
        # Wait for response packet to indicate success
        return self._wait_for_packet(RESPONSE_TIMEOUT)
```

`RpcServer` is a bit more involved because it needs PySerial to handle initializing the serial connection whereas the `RpcClient`, which is intended to be run on a CircuitPython device, has already taken care of that. The `RpcServer` starts off with needing a `handler` function passed in, which is called whenever a packet is received. The reason for using this strategy is because of function scope. If the handler were built into the library, only the library functions would be accessible.

One of the nice things about the expected setup is that the `RpcServer` is expecting a CircuitPython device, so it makes use of the Adafruit_Board_Toolkit to automatically detect which port the MacroPad is connected to. It is also able to return only the CDC Data devices, further simplifying things.

The `loop()` function is intended to be called regularly to listen for and process request packets by sending then to the `handler` function specified when the library was instantiated.

```
class RpcServer(_Rpc):
    def __init__(self, handler, baudrate=9600):
        super().__init__()
        self._serial = self.init_serial(baudrate)
        self._handler = handler

    def _packet_format(self):
        return self.create_request_packet(None).keys()

    def init_serial(self, baudrate):
        port = self.detect_port()

        return serial.Serial(
            port,
            baudrate,
            parity='N',
            rtscts=False,
```

```
            xonxoff=False,
            exclusive=True,
        )

    def detect_port(self):
        """
        Detect the port automatically
        """
        comports = adafruit_board_toolkit.circuitpython_serial.data_comports()
        ports = [comport.device for comport in comports]
        if len(ports) >= 1:
            if len(ports) > 1:
                print("Multiple devices detected, using the first detected port.")
            return ports[0]
        raise RuntimeError("Unable to find any CircuitPython Devices with the CDC
Data port enabled.")

    def loop(self, timeout=None):
        packet = self._wait_for_packet(timeout)
        if "error" not in packet:
            response_packet = self._handler(packet)
            self._serial.write(bytes(json.dumps(response_packet), "utf-8"))

    def close_serial(self):
        if self._serial is not None:
            self._serial.close()
```

Full Code Listing

```
# SPDX-FileCopyrightText: Copyright (c) 2021 Melissa LeBlanc-Williams for Adafruit
Industries
#
# SPDX-License-Identifier: Unlicense
"""
USB CDC Remote Procedure Call class
"""

import time
import json
try:
    import serial
    import adafruit_board_toolkit.circuitpython_serial
    json_decode_exception = json.decoder.JSONDecodeError
except ImportError:
    import usb_cdc as serial
    json_decode_exception = ValueError

RESPONSE_TIMEOUT=5
DATA_TIMEOUT=0.5

class RpcError(Exception):
    """For RPC Specific Errors"""
    pass

class _Rpc:
    def __init__(self):
        self._serial = None

    @staticmethod
    def create_response_packet(error=False, error_type="RPC", message=None,
return_val=None):
        return {
            "error": error,
            "error_type": error_type if error else None,
            "message": message,
```

```
            "return_val": return_val
        }

    @staticmethod
    def create_request_packet(function, args=[], kwargs={}):
        return {
            "function": function,
            "args": args,
            "kwargs": kwargs
        }

    def _wait_for_packet(self, timeout=None):
        incoming_packet = b""
        if timeout is not None:
            response_start_time = time.monotonic()
        while True:
            if incoming_packet:
                data_start_time = time.monotonic()
            while not self._serial.in_waiting:
                if incoming_packet and (time.monotonic() - data_start_time) >=
DATA_TIMEOUT:
                    incoming_packet = b""
                if not incoming_packet and timeout is not None:
                    if (time.monotonic() - response_start_time) >= timeout:
                        return self.create_response_packet(error=True,
message="Timed out waiting for response")
                time.sleep(0.001)
            data = self._serial.read(self._serial.in_waiting)
            if data:
                try:
                    incoming_packet += data
                    packet = json.loads(incoming_packet)
                    # json can try to be clever with missing braces, so make sure
we have everything
                    if sorted(tuple(packet.keys())) ==
sorted(self._packet_format()):
                        return packet
                except json_decode_exception:
                    pass # Incomplete packet

class RpcClient(_Rpc):
    def __init__(self):
        super().__init__()
        self._serial = serial.data

    def _packet_format(self):
        return self.create_response_packet().keys()

    def call(self, function, *args, **kwargs):
        packet = self.create_request_packet(function, args, kwargs)
        self._serial.write(bytes(json.dumps(packet), "utf-8"))
        # Wait for response packet to indicate success
        return self._wait_for_packet(RESPONSE_TIMEOUT)

class RpcServer(_Rpc):
    def __init__(self, handler, baudrate=9600):
        super().__init__()
        self._serial = self.init_serial(baudrate)
        self._handler = handler

    def _packet_format(self):
        return self.create_request_packet(None).keys()

    def init_serial(self, baudrate):
        port = self.detect_port()

        return serial.Serial(
            port,
            baudrate,
```

```
            parity='N',
            rtscts=False,
            xonxoff=False,
            exclusive=True,
        )

    def detect_port(self):
        """
        Detect the port automatically
        """
        comports = adafruit_board_toolkit.circuitpython_serial.data_comports()
        ports = [comport.device for comport in comports]
        if len(ports) >= 1:
            if len(ports) > 1:
                print("Multiple devices detected, using the first detected port.")
            return ports[0]
        raise RuntimeError("Unable to find any CircuitPython Devices with the CDC
Data port enabled.")

    def loop(self, timeout=None):
        packet = self._wait_for_packet(timeout)
        if "error" not in packet:
            response_packet = self._handler(packet)
            self._serial.write(bytes(json.dumps(response_packet), "utf-8"))

    def close_serial(self):
        if self._serial is not None:
            self._serial.close()
```

MacroPad Code

First the code starts off by importing all of the libraries that will be used. Ones to take note of are the `rpc` library which is project specific and the `secrets` which should have been set up in an earlier step.

```
import time
import displayio
import terminalio
from adafruit_display_shapes.rect import Rect
from rpc import RpcClient, RpcError
from adafruit_display_text import label
from adafruit_macropad import MacroPad
from secrets import secrets
```

Now to initialize the MacroPad and RpcClient libraries.

```
macropad = MacroPad()
rpc = RpcClient()
```

Next are the configurable settings:

- `COMMAND_TOPIC` is what Home Assistant should listen to.

- `SUBSCRIBE_TOPICS` are the MQTT topics that the code should subscribe to in order to get the current status of the lights. It is highly likely that you will need to change this in order to match your specific setup.
- `ENCODER_ITEM` refers to the `key_number` that should be sent when pressing the encoder knob. If you don't want it to respond, change the value to `None`.
- `KEY_LABELS` are just the labels that are displayed that correspond to the buttons.
- `UPDATE_DELAY` is the amount of time in seconds that the code should wait after sending a command before checking the status of the light. If it often seems to be the wrong status, you may want to increase the value, but it will seem less snappy.
- `NEOPIXEL_COLORS` refer to the value that the NeoPixels should light up corresponding to the value of the possible answers in `SUBSCRIBE_TOPICS`.

```
COMMAND_TOPIC = "macropad/peripheral"
SUBSCRIBE_TOPICS = ("stat/demoswitch/POWER", "stat/office-light/POWER")
ENCODER_ITEM = 0
KEY_LABELS = ("Demo", "Office")
UPDATE_DELAY = 0.25
NEOPIXEL_COLORS = {
    "OFF": 0xff0000,
    "ON": 0x00ff00,
}
```

Next to define a custom `MqttError` to differentiate it from other Python errors.

```
class MqttError(Exception):
    """For MQTT Specific Errors"""
    pass
```

The next bit of code will draw the labels that display what the buttons do and was borrowed from the MACROPAD Hotkeys () guide because of the nice aesthetic.

```
group = displayio.Group()
for key_index in range(12):
    x = key_index % 3
    y = key_index // 3
    group.append(label.Label(terminalio.FONT, text=(str(KEY_LABELS[key_index]) if
key_index < len(KEY_LABELS) else ''), 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='Home Assistant', color=0x000000,
                         anchored_position=(macropad.display.width//2, -2),
                         anchor_point=(0.5, 0.0)))
macropad.display.show(group)
```

This next function is simple, but makes things much easier. It allows you to specify the function you would like to call remotely and pass in the parameters in the same way as you would pass them into the remote function. It also handles raising the appropriate kind of error or returning the Return Value if it was successful.

```
def rpc_call(function, *args, **kwargs):
    response = rpc.call(function, *args, **kwargs)
    if response["error"]:
        if response["error_type"] == "mqtt":
            raise MqttError(response["message"])
        raise RpcError(response["message"])
    return response["return_val"]
```

The next couple of functions use the `rpc_call()` function to connect to MQTT and update the key colors.

```
def mqtt_init():
    rpc_call("mqtt_init", secrets["mqtt_broker"],
username=secrets["mqtt_username"], password=secrets["mqtt_password"],
port=secrets["mqtt_port"])
    rpc_call("mqtt_connect")

def update_key(key_number):
    switch_state = rpc_call("mqtt_get_last_value", SUBSCRIBE_TOPICS[key_number])
    if switch_state is not None:
        macropad.pixels[key_number] = NEOPIXEL_COLORS[switch_state]
    else:
        macropad.pixels[key_number] = 0
```

This bit of code waits for the server to start running by attempting to call a simple function and checking if an `RpcError` is being returned.

```
server_is_running = False
print("Waiting for server...")
while not server_is_running:
    try:
        server_is_running = rpc_call("is_running")
        print("Connected")
    except RpcError:
        pass
```

Once it is all connected, one last bit of code is run before entering the main loop. It just connects to MQTT and then subscribes to all of the `SUBSCRIBE_TOPICS`.

```
mqtt_init()
last_macropad_encoder_value = macropad.encoder

for key_number, topic in enumerate(SUBSCRIBE_TOPICS):
    rpc_call("mqtt_subscribe", topic)
    update_key(key_number)
```

The main loop just listens to the MacroPad library for button presses and encoder changes and if it detects them it will publish that change to MQTT.

```
while True:
    output = {}

    key_event = macropad.keys.events.get()
    if key_event and key_event.pressed:
        output["key_number"] = key_event.key_number

    if macropad.encoder != last_macropad_encoder_value:
        output["encoder"] = macropad.encoder - last_macropad_encoder_value
        last_macropad_encoder_value = macropad.encoder

    macropad.encoder_switch_debounced.update()
    if macropad.encoder_switch_debounced.pressed and "key_number" not in output and
ENCODER_ITEM is not None:
        output["key_number"] = ENCODER_ITEM

    if output:
        try:
            rpc_call("mqtt_publish", COMMAND_TOPIC, output)
            if "key_number" in output:
                time.sleep(UPDATE_DELAY)
                update_key(output["key_number"])
            elif ENCODER_ITEM is not None:
                update_key(ENCODER_ITEM)
        except MqttError:
            mqtt_init()
        except RpcError as err_msg:
            print(err_msg)
```

Full Code Listing

```
# SPDX-FileCopyrightText: Copyright (c) 2021 Melissa LeBlanc-Williams for Adafruit
Industries
#
# SPDX-License-Identifier: Unlicense
"""
Home Assistant Remote Procedure Call for MacroPad.
"""
import time
import displayio
import terminalio
from adafruit_display_shapes.rect import Rect
from rpc import RpcClient, RpcError
from adafruit_display_text import label
from adafruit_macropad import MacroPad
from secrets import secrets

macropad = MacroPad()
rpc = RpcClient()

COMMAND_TOPIC = "macropad/peripheral"
SUBSCRIBE_TOPICS = ("stat/demoswitch/POWER", "stat/office-light/POWER")
ENCODER_ITEM = 0
KEY_LABELS = ("Demo", "Office")
UPDATE_DELAY = 0.25
NEOPIXEL_COLORS = {
    "OFF": 0xff0000,
    "ON": 0x00ff00,
}
```

```
class MqttError(Exception):
    """For MQTT Specific Errors"""
    pass
# 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=(str(KEY_LABELS[key_index]) if
key_index < len(KEY_LABELS) else ''), 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='Home Assistant', color=0x000000,
                         anchored_position=(macropad.display.width//2, -2),
                         anchor_point=(0.5, 0.0)))
macropad.display.show(group)

def rpc_call(function, *args, **kwargs):
    response = rpc.call(function, *args, **kwargs)
    if response["error"]:
        if response["error_type"] == "mqtt":
            raise MqttError(response["message"])
        raise RpcError(response["message"])
    return response["return_val"]

def mqtt_init():
    rpc_call("mqtt_init", secrets["mqtt_broker"], username=secrets["mqtt_username"],
password=secrets["mqtt_password"], port=secrets["mqtt_port"])
    rpc_call("mqtt_connect")

def update_key(key_number):
    if key_number < len(SUBSCRIBE_TOPICS):
        switch_state = rpc_call("mqtt_get_last_value", SUBSCRIBE_TOPICS[key_number])
        if switch_state is not None:
            macropad.pixels[key_number] = NEOPIXEL_COLORS[switch_state]
        else:
            macropad.pixels[key_number] = 0

server_is_running = False
print("Waiting for server...")
while not server_is_running:
    try:
        server_is_running = rpc_call("is_running")
        print("Connected")
    except RpcError:
        pass

mqtt_init()
last_macropad_encoder_value = macropad.encoder

for key_number, topic in enumerate(SUBSCRIBE_TOPICS):
    rpc_call("mqtt_subscribe", topic)
    update_key(key_number)

while True:
    output = {}

    key_event = macropad.keys.events.get()
    if key_event and key_event.pressed:
        output["key_number"] = key_event.key_number

    if macropad.encoder != last_macropad_encoder_value:
        output["encoder"] = macropad.encoder - last_macropad_encoder_value
        last_macropad_encoder_value = macropad.encoder
```

```
    macropad.encoder_switch_debounced.update()
    if macropad.encoder_switch_debounced.pressed and "key_number" not in output and
ENCODER_ITEM is not None:
        output["key_number"] = ENCODER_ITEM

    if output:
        try:
            rpc_call("mqtt_publish", COMMAND_TOPIC, output)
            if "key_number" in output:
                time.sleep(UPDATE_DELAY)
                update_key(output["key_number"])
            elif ENCODER_ITEM is not None:
                update_key(ENCODER_ITEM)
        except MqttError:
            mqtt_init()
        except RpcError as err_msg:
            print(err_msg)
```

Host Computer Code

Finally there is code that runs on the host computer and acts as a server. First are the imported libraries:

```
import time
import json
import ssl
import socket
import adafruit_minimqtt.adafruit_minimqtt as MQTT
from rpc import RpcServer
```

Next are a few variables to keep track of the state of things:

```
mqtt_client = None
mqtt_connected = False
last_mqtt_messages = {}
```

Next is a list of protected functions. The purpose of this list is to prevent calling these function to avoid memory loops or other situations that would likely crash Python or may result in some difficult to debug situations.

```
# For program flow purposes, we do not want these functions to be called remotely
PROTECTED_FUNCTIONS = ["main", "handle_rpc"]
```

These functions are to keep track of our connection and the status of the topics that are being watched. These are used as callbacks when MQTT is initialized.

```
def connect(mqtt_client, userdata, flags, rc):
    global mqtt_connected
    mqtt_connected = True

def disconnect(mqtt_client, userdata, rc):
    global mqtt_connected
```

```
    mqtt_connected = False

def message(client, topic, message):
    last_mqtt_messages[topic] = message
```

Next tp define a custom `MqttError` like was done in the MacroPad code.

```
class MqttError(Exception):
    """For MQTT Specific Errors"""
    pass
```

Next there are all of the functions that are called by RPC and are just standard MQTT connection functions as used in the library examples with a few exceptions.

First in `mqtt_publish()`, if the connection has been dropped, it will attempt to reconnect automatically. This seemed to make the code overall more reliable.

`mqtt_get_last_value()` just returns a corresponding value from one of the topics it was watching if available, otherwise it just returns None.

Finally is the `is_running()` function which is simply used to check that there is an RPC connection when the MacroPad is waiting for the server.

```
# Default to 1883 as SSL on CPython is not currently supported
def mqtt_init(broker, port=1883, username=None, password=None):
    global mqtt_client, mqtt_connect_info
    mqtt_client = MQTT.MQTT(
        broker=broker,
        port=port,
        username=username,
        password=password,
        socket_pool=socket,
        ssl_context=ssl.create_default_context(),
    )

    mqtt_client.on_connect = connect
    mqtt_client.on_disconnect = disconnect
    mqtt_client.on_message = message

def mqtt_connect():
    mqtt_client.connect()

def mqtt_publish(topic, payload):
    if mqtt_client is None:
        raise MqttError("MQTT is not initialized")
    try:
        return_val = mqtt_client.publish(topic, json.dumps(payload))
    except BrokenPipeError:
        time.sleep(0.5)
        mqtt_client.connect()
        return_val = mqtt_client.publish(topic, json.dumps(payload))
    return return_val

def mqtt_subscribe(topic):
    if mqtt_client is None:
        raise MqttError("MQTT is not initialized")
    return mqtt_client.subscribe(topic)
```

```
def mqtt_get_last_value(topic):
    """Return the last value we have received regarding a topic"""
    if topic in last_mqtt_messages.keys():
        return last_mqtt_messages[topic]
    return None

def is_running():
    return True
```

This is the handler function and where all the magic happens. It starts by making sure the called function isn't in the protected functions list. Then it checks to make sure the function is in the `globals()` list just to make sure something like `the_function_that_doesn't_really_exist()` was called.

Assuming it gets this far, it will just call the function with all of the arguments and let Python handle any mismatched arguments. If everything happened like it was supposed to, there may be a return value. A response packet is created and returned. If not, an error response packet is created and returned.

```
def handle_rpc(packet):
    """This function will verify good data in packet,
    call the method with parameters, and generate a response
    packet as the return value"""
    print("Received packet")
    func_name = packet['function']
    if func_name in PROTECTED_FUNCTIONS:
        return rpc.create_response_packet(error=True, message=f"{func_name}'() is a
protected function and can not be called.")
    if func_name not in globals():
        return rpc.create_response_packet(error=True, message=f"Function {func_name}
() not found")
    try:
        return_val = globals()[func_name](*packet['args'], **packet['kwargs'])
    except MqttError as err:
        return rpc.create_response_packet(error=True, error_type="MQTT",
message=str(err))

    packet = rpc.create_response_packet(return_val=return_val)
    return packet
```

Here is the main function that really just keeps calling the RpcServer `loop()` function and if MQTT is connected, it calls the MQTT `loop()` function.

```
def main():
    """Command line, entry point"""
    global mqtt_connected
    while True:
        rpc.loop(0.25)
        if mqtt_connected and mqtt_client is not None:
            try:
                mqtt_client.loop(0.5)
            except AttributeError:
                mqtt_connected = False
```

Finally is the code that serves as the entry and exit points to the script.

```
if __name__ == '__main__':
    rpc = RpcServer(handle_rpc)
    try:
        print(f"Listening for RPC Calls, to stop press \"CTRL+C\"")
        main()
    except KeyboardInterrupt:
        print("")
        print(f"Caught interrupt, exiting...")
    rpc.close_serial()
```

Full Code Listing

```
# SPDX-FileCopyrightText: 2021 Melissa LeBlanc-Williams for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import json
import ssl
import socket
import adafruit_minimqtt.adafruit_minimqtt as MQTT
from rpc import RpcServer

mqtt_client = None
mqtt_connected = False
last_mqtt_messages = {}

# For program flow purposes, we do not want these functions to be called remotely
PROTECTED_FUNCTIONS = ["main", "handle_rpc"]

def connect(mqtt_client, userdata, flags, rc):
    global mqtt_connected
    mqtt_connected = True

def disconnect(mqtt_client, userdata, rc):
    global mqtt_connected
    mqtt_connected = False

def message(client, topic, message):
    last_mqtt_messages[topic] = message

class MqttError(Exception):
    """For MQTT Specific Errors"""
    pass

# Default to 1883 as SSL on CPython is not currently supported
def mqtt_init(broker, port=1883, username=None, password=None):
    global mqtt_client, mqtt_connect_info
    mqtt_client = MQTT.MQTT(
        broker=broker,
        port=port,
        username=username,
        password=password,
        socket_pool=socket,
        ssl_context=ssl.create_default_context(),
    )

    mqtt_client.on_connect = connect
    mqtt_client.on_disconnect = disconnect
    mqtt_client.on_message = message
```

```
def mqtt_connect():
    mqtt_client.connect()

def mqtt_publish(topic, payload):
    if mqtt_client is None:
        raise MqttError("MQTT is not initialized")
    try:
        return_val = mqtt_client.publish(topic, json.dumps(payload))
    except BrokenPipeError:
        time.sleep(0.5)
        mqtt_client.connect()
        return_val = mqtt_client.publish(topic, json.dumps(payload))
    return return_val

def mqtt_subscribe(topic):
    if mqtt_client is None:
        raise MqttError("MQTT is not initialized")
    return mqtt_client.subscribe(topic)

def mqtt_get_last_value(topic):
    """Return the last value we have received regarding a topic"""
    if topic in last_mqtt_messages.keys():
        return last_mqtt_messages[topic]
    return None

def is_running():
    return True

def handle_rpc(packet):
    """This function will verify good data in packet,
    call the method with parameters, and generate a response
    packet as the return value"""
    print("Received packet")
    func_name = packet['function']
    if func_name in PROTECTED_FUNCTIONS:
        return rpc.create_response_packet(error=True, message=f"{func_name}'() is a
protected function and can not be called.")
    if func_name not in globals():
        return rpc.create_response_packet(error=True, message=f"Function {func_name}
() not found")
    try:
        return_val = globals()[func_name](*packet['args'], **packet['kwargs'])
    except MqttError as err:
        return rpc.create_response_packet(error=True, error_type="MQTT",
message=str(err))

    packet = rpc.create_response_packet(return_val=return_val)
    return packet

def main():
    """Command line, entry point"""
    global mqtt_connected
    while True:
        rpc.loop(0.25)
        if mqtt_connected and mqtt_client is not None:
            try:
                mqtt_client.loop(0.5)
            except AttributeError:
                mqtt_connected = False

if __name__ == '__main__':
    rpc = RpcServer(handle_rpc)
    try:
        print(f"Listening for RPC Calls, to stop press \"CTRL+C\"")
        main()
    except KeyboardInterrupt:
        print("")
```

```
        print(f"Caught interrupt, exiting...")
    rpc.close_serial()
```

Host Computer Code

Finally there is code that runs on the host computer and acts as a server. First are the imported libraries:

```
import time
import json
import ssl
import socket
import adafruit_minimqtt.adafruit_minimqtt as MQTT
from rpc import RpcServer
```

Next are a few variables to keep track of the state of things:

```
mqtt_client = None
mqtt_connected = False
last_mqtt_messages = {}
```

Next is a list of protected functions. The purpose of this list is to prevent calling these function to avoid memory loops or other situations that would likely crash Python or may result in some difficult to debug situations.

```
# For program flow purposes, we do not want these functions to be called remotely
PROTECTED_FUNCTIONS = ["main", "handle_rpc"]
```

These functions are to keep track of our connection and the statuses of the topics that are being watched. These are used as callbacks when MQTT is initialized.

```
def connect(mqtt_client, userdata, flags, rc):
    global mqtt_connected
    mqtt_connected = True

def disconnect(mqtt_client, userdata, rc):
    global mqtt_connected
    mqtt_connected = False

def message(client, topic, message):
    last_mqtt_messages[topic] = message
```

Next we define a custom `MqttError` like is done in the MacroPad code.

```
class MqttError(Exception):
    """For MQTT Specific Errors"""
    pass
```

Next there are all of the functions that are called by RPC and are just standard MQTT connection functions as used in the library examples with a few exceptions.

First in `mqtt_publish()`, if the connection has been dropped, it will attempt to reconnect automatically. This seemed to make the code overall more reliable.

`mqtt_get_last_value()` just returns a corresponding value from one of the topics it was watching if available, otherwise it just returns `None`.

Finally is the `is_running()` function which is simply used to check that there is an RPC connection when the MacroPad is waiting for the server.

```
# Default to 1883 as SSL on CPython is not currently supported
def mqtt_init(broker, port=1883, username=None, password=None):
    global mqtt_client, mqtt_connect_info
    mqtt_client = MQTT.MQTT(
        broker=broker,
        port=port,
        username=username,
        password=password,
        socket_pool=socket,
        ssl_context=ssl.create_default_context(),
    )

    mqtt_client.on_connect = connect
    mqtt_client.on_disconnect = disconnect
    mqtt_client.on_message = message

def mqtt_connect():
    mqtt_client.connect()

def mqtt_publish(topic, payload):
    if mqtt_client is None:
        raise MqttError("MQTT is not initialized")
    try:
        return_val = mqtt_client.publish(topic, json.dumps(payload))
    except BrokenPipeError:
        time.sleep(0.5)
        mqtt_client.connect()
        return_val = mqtt_client.publish(topic, json.dumps(payload))
    return return_val

def mqtt_subscribe(topic):
    if mqtt_client is None:
        raise MqttError("MQTT is not initialized")
    return mqtt_client.subscribe(topic)

def mqtt_get_last_value(topic):
    """Return the last value we have received regarding a topic"""
    if topic in last_mqtt_messages.keys():
        return last_mqtt_messages[topic]
    return None

def is_running():
    return True
```

This is the handler function and where all the magic happens. It starts by making sure the called function isn't in the protected functions list. Then it checks to make sure the

function is in the `globals()` list just to make sure something like `the_function_that_doesn't_really_exist()` was called.

Assuming it gets this far, it will just call the function with all of the arguments and let Python handle any mismatched arguments. If everything happened like it was supposed to, there may be a return value. A response packet is created and returned. If not, an error response packet is created and returned.

```
def handle_rpc(packet):
    """This function will verify good data in packet,
    call the method with parameters, and generate a response
    packet as the return value"""
    print("Received packet")
    func_name = packet['function']
    if func_name in PROTECTED_FUNCTIONS:
        return rpc.create_response_packet(error=True, message=f"{func_name}'() is a
protected function and can not be called.")
    if func_name not in globals():
        return rpc.create_response_packet(error=True, message=f"Function {func_name}
() not found")
    try:
        return_val = globals()[func_name](*packet['args'], **packet['kwargs'])
    except MqttError as err:
        return rpc.create_response_packet(error=True, error_type="MQTT",
message=str(err))

    packet = rpc.create_response_packet(return_val=return_val)
    return packet
```

Here is the main function that really just keeps calling the RpcServer `loop()` function and if MQTT is connected, it calls the MQTT `loop()` function.

```
def main():
    """Command line, entry point"""
    global mqtt_connected
    while True:
        rpc.loop(0.25)
        if mqtt_connected and mqtt_client is not None:
            try:
                mqtt_client.loop(0.5)
            except AttributeError:
                mqtt_connected = False
```

Finally is the code that serves as the entry and exit points to the script.

```
if __name__ == '__main__':
    rpc = RpcServer(handle_rpc)
    try:
        print(f"Listening for RPC Calls, to stop press \"CTRL+C\"")
        main()
    except KeyboardInterrupt:
        print("")
        print(f"Caught interrupt, exiting...")
    rpc.close_serial()
```

Full Code Listing

```
# SPDX-FileCopyrightText: 2021 Melissa LeBlanc-Williams for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import json
import ssl
import socket
import adafruit_minimqtt.adafruit_minimqtt as MQTT
from rpc import RpcServer

mqtt_client = None
mqtt_connected = False
last_mqtt_messages = {}

# For program flow purposes, we do not want these functions to be called remotely
PROTECTED_FUNCTIONS = ["main", "handle_rpc"]

def connect(mqtt_client, userdata, flags, rc):
    global mqtt_connected
    mqtt_connected = True

def disconnect(mqtt_client, userdata, rc):
    global mqtt_connected
    mqtt_connected = False

def message(client, topic, message):
    last_mqtt_messages[topic] = message

class MqttError(Exception):
    """For MQTT Specific Errors"""
    pass

# Default to 1883 as SSL on CPython is not currently supported
def mqtt_init(broker, port=1883, username=None, password=None):
    global mqtt_client, mqtt_connect_info
    mqtt_client = MQTT.MQTT(
        broker=broker,
        port=port,
        username=username,
        password=password,
        socket_pool=socket,
        ssl_context=ssl.create_default_context(),
    )

    mqtt_client.on_connect = connect
    mqtt_client.on_disconnect = disconnect
    mqtt_client.on_message = message

def mqtt_connect():
    mqtt_client.connect()

def mqtt_publish(topic, payload):
    if mqtt_client is None:
        raise MqttError("MQTT is not initialized")
    try:
        return_val = mqtt_client.publish(topic, json.dumps(payload))
    except BrokenPipeError:
        time.sleep(0.5)
        mqtt_client.connect()
        return_val = mqtt_client.publish(topic, json.dumps(payload))
    return return_val

def mqtt_subscribe(topic):
```

```
    if mqtt_client is None:
        raise MqttError("MQTT is not initialized")
    return mqtt_client.subscribe(topic)

def mqtt_get_last_value(topic):
    """Return the last value we have received regarding a topic"""
    if topic in last_mqtt_messages.keys():
        return last_mqtt_messages[topic]
    return None

def is_running():
    return True

def handle_rpc(packet):
    """This function will verify good data in packet,
    call the method with parameters, and generate a response
    packet as the return value"""
    print("Received packet")
    func_name = packet['function']
    if func_name in PROTECTED_FUNCTIONS:
        return rpc.create_response_packet(error=True, message=f"{func_name}'() is a
protected function and can not be called.")
    if func_name not in globals():
        return rpc.create_response_packet(error=True, message=f"Function {func_name}
() not found")
    try:
        return_val = globals()[func_name](*packet['args'], **packet['kwargs'])
    except MqttError as err:
        return rpc.create_response_packet(error=True, error_type="MQTT",
message=str(err))

    packet = rpc.create_response_packet(return_val=return_val)
    return packet

def main():
    """Command line, entry point"""
    global mqtt_connected
    while True:
        rpc.loop(0.25)
        if mqtt_connected and mqtt_client is not None:
            try:
                mqtt_client.loop(0.5)
            except AttributeError:
                mqtt_connected = False

if __name__ == '__main__':
    rpc = RpcServer(handle_rpc)
    try:
        print(f"Listening for RPC Calls, to stop press \"CTRL+C\"")
        main()
    except KeyboardInterrupt:
        print("")
        print(f"Caught interrupt, exiting...")
    rpc.close_serial()
```