Moon Phase Clock for Adafruit Matrix Portal

Created by Phillip Burgess

https://learn.adafruit.com/moon-phase-clock-for-adafruit-matrixportal

Last updated on 2023-10-18 12:11:17 PM EDT
# Table of Contents

## Overview
- Parts

## Prep the MatrixPortal
- Power Prep
- Power Terminals
- Panel Power
- Dual Matrix Setup
- Board Connection

## Install CircuitPython
- Set up CircuitPython Quick Start!
- Further Information

## Install Code and Graphics
- Installing Project Code

## Configure Network and Location

## Clock Usage
- Clock Operation Basics
- Customizing the Clock Display
- Timekeeping

## Moon Facts
- Other Moon Factoids
Overview

It’s hard work being a werewolf, and I don’t just mean cleaning blood stains out of the carpet or dealing with fleas in the house.

It’s this messy business of the full Moon. Most wall calendars and news sources, if they even mention Moon phases, there’s about a 50/50 chance that they’re off by a full day (explained on the “Moon Facts” page). You might be out at a fancy event wearing a rental tuxedo when things turn weird. There goes your deposit. So awkward.

The only way to get a correct Moon phase for your time and location is with math and science. Our Matrix Portal Moon phase clock lets you know where things stand right here and now, portraying the Moon’s phase more accurately than a printed calendar, and whether it’s currently in the sky or has set.

So now you can prepare. No tux! Stay home and put on a comfortable flannel shirt like werewolves do.

This project requires:

- Adafruit Matrix Portal M4 () board
- Any of our 64x32 pixel “HUB75” (not NeoPixel) RGB LED matrices (other sizes are not supported by this code)
- USB C cable
- USB power supply with output of 2 Amps or more
- WiFi internet access
- An Adafruit IO () account (the free basic plan will suffice, we’re just using this to set the time)
- An insatiable hunger for human flesh
This guide will get the software running on the bare Matrix Portal hardware. Mounting or supporting the clock in an enclosure or frame is left as an exercise to the reader.

Next we'll connect the Matrix Portal board and LED matrix. Do not over-tighten the screws on the power terminals or they can shear off. This is best done in human form.

Parts

Adafruit Matrix Portal - CircuitPython Powered Internet Display
Folks love our wide selection of RGB matrices and accessories, for making custom colorful LED displays... and our RGB Matrix Shields...
https://www.adafruit.com/product/4745

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

You can use a USB C power supply or a USB micro B with a micro B to C adapter ()

Official Raspberry Pi Power Supply 5.1V 3A with USB C
The official Raspberry Pi USB-C power supply is here! And of course, we have 'em in classic Adafruit black! Superfast with just the right amount of cable length to get your Pi 4...
https://www.adafruit.com/product/4298
5V 2.5A Switching Power Supply with 20AWG MicroUSB Cable
Our all-in-one 5V 2.5 Amp + MicroUSB cable power adapter is the perfect choice for powering single-board computers like Raspberry Pi, BeagleBone, or anything else that's...

Micro B USB to USB C Adapter
As technology changes and adapts, so does Adafruit, and speaking of adapting, this adapter has a Micro B USB jack and a USB C...
https://www.adafruit.com/product/4299

If you'd like your LEDs diffused, some acrylic may help

Black LED Diffusion Acrylic Panel 12" x 12" - 0.1" / 2.6mm thick
A nice whoppin' slab of some lovely black acrylic to add some extra diffusion to your LED Matrix project. This material is 2.6mm (0.1") thick and is made of special cast...
https://www.adafruit.com/product/4594

Adafruit carries a number of 64x32 RGB LED Matrices, varying between the space between LEDs (pitch) and whether rigid or flexible. Choose your favorite - larger pitch means the display is larger, width and height-wise but with the same number of pixels, and larger may be easier to read further away. Smaller for near your desk, for example.
64x32 RGB LED Matrix - 3mm pitch
Bring a little bit of Times Square into your home with this sweet 64 x 32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them...
https://www.adafruit.com/product/2279

64x32 RGB LED Matrix - 4mm pitch
Bring a little bit of Times Square into your home with this sweet 64 x 32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them...
https://www.adafruit.com/product/2278

64x32 RGB LED Matrix - 5mm pitch
Bring a little bit of Times Square into your home with this sweet 64x32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them on...
https://www.adafruit.com/product/2277

64x32 RGB LED Matrix - 6mm pitch
Bring a little bit of Times Square into your home with this sweet 64x32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them on...
https://www.adafruit.com/product/2276
If you've played with multiplexed RGB matrices, you may have wondered "hey, could we possibly manufacture these on a thin enough PCB, so it's flexible?" and the...

https://www.adafruit.com/product/3826

If you've played with multiplexed RGB matrices, you may have wondered "hey, could we possibly manufacture these on a thin enough PCB so it's flexible?" and the answer...

https://www.adafruit.com/product/3803
Prep the MatrixPortal

Power Prep
The MatrixPortal supplies power to the matrix display panel via two standoffs. These come with protective tape applied (part of our manufacturing process) which MUST BE REMOVED!

Use some tweezers or a fingernail to remove the two amber circles.
Power Terminals

Next, screw in the spade connectors to the corresponding standoff.

red wire goes to +5V
black wire goes to GND
Panel Power
Plug either one of the four-conductor power plugs into the power connector pins on the panel. The plug can only go in one way, and that way is marked on the board's silkscreen.

Dual Matrix Setup

If you're planning to use a 64x64 matrix, follow these instructions on soldering the Address E Line jumper.
Board Connection

Now, plug the board into the left side shrouded 8x2 connector as shown. The orientation matters, so take a moment to confirm that the white indicator arrow on the matrix panel is oriented pointing up and right as seen here and the MatrixPortal overhangs the edge of the panel when connected. This allows you to use the edge buttons from the front side.

Check nothing is impeding the board from plugging in firmly. If there’s a plastic nub on the matrix that’s keeping the Portal from sitting flat, cut it off with diagonal cutters.
Install 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.

Set up CircuitPython Quick Start!

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

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

Further Information

For more detailed info on installing CircuitPython, check out Installing CircuitPython ().
Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).

Plug your MatrixPortal M4 into your computer using a known-good USB cable.

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

Double-click the Reset button (indicated by the green arrow) on your board, and you will see the NeoPixel RGB LED (indicated by the magenta arrow) turn green. If it turns red, check the USB cable, try another USB port, etc.

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!
You will see a new disk drive appear called MATRIXBOOT.

Drag the adafruit_circuitpython_etc.uf2 file to MATRIXBOOT.

The LED will flash. Then, the MATRIXBOOT drive will disappear and a new disk drive called CIRCUITPY will appear.

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

Install Code and Graphics

Back up any existing code or files you want to keep from your Matrix Portal CIRCUITPY drive.
Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your CIRCUITPY drive. Then you need to update code.py with the example script.

Thankfully, we can do this in one go. In the example below, click the Download Project Bundle button below to download the necessary libraries and the code.py file in a zip file. Extract the contents of the zip file, open the directory Matrix_Portal_Moon_Clock/ and then click on the directory that matches the version of CircuitPython you’re using and copy the contents of that directory to your CIRCUITPY drive.

Your CIRCUITPY drive should now look similar to the following image:

```
# SPDX-FileCopyrightText: 2020 Phillip Burgess for Adafruit Industries
# # SPDX-License-Identifier: MIT

MOON PHASE CLOCK for Adafruit Matrix Portal: displays current time, lunar phase and time of next moonrise or moonset. Requires WiFi internet access and Adafruit IO user account (basic account is free, just needs registration).

Written by Phil 'PaintYourDragon' Burgess for Adafruit Industries.
MIT license, all text above must be included in any redistribution.

BDF fonts from the X.Org project. Startup 'splash' images should not be
```
import gc
import time
import math
import board
import busio
import displayio
from rtc import RTC
from adafruit_matrixportal.network import Network
from adafruit_matrixportal.matrix import Matrix
from adafruit_bitmap_font import bitmap_font
import adafruit_display_text.label
import adafruit_lis3dh

try:
    from secrets import secrets
except ImportError:
    print('WiFi secrets are kept in secrets.py, please add them there!')
    raise

# CONFIGURABLE SETTINGS
TWELVE_HOUR = True  # If set, use 12-hour time vs 24-hour (e.g. 3:00 vs 15:00)
COUNTDOWN = False  # If set, show time to (vs time of) next rise/set event
MONTH_DAY = True   # If set, use MM/DD vs DD/MM (e.g. 31/12 vs 12/31)
BITPLANES = 6      # Ideally 6, but can set lower if RAM is tight

# SOME UTILITY FUNCTIONS AND CLASSES

# Notes to Future Self on timekeeping: times are expressed in so many
# formats throughout this code, a variable naming system is used: local
# times (i.e. in clock’s present geographic time zone) will have _local
# in their variable name, while UTC times (aka Greenwich or Zulu time)
# will have _utc. Types are also explicitly stated: strings (e.g.
# "2023-07-20T08:37-07:00") will have _string in the variable name,
# struct_time objects will have _struct, and integer "UNIX time" epoch
# (seconds will have _seconds. Conversions (offset is signed, e.g. -700):
# 1. Convert UTC to local time: add UTC offset;
# 1. Convert local to UTC time: subtract UTC offset; utc = local - offset

def update_system_time():
    # Update system clock date/time from Adafruit IO. Credentials and time
    # zone are in secrets.py. See http://worldtimeapi.org/api/timezone for
    # list of time zones. If missing, will attempt using IP geolocation.
    # Returns present local (not UTC) time as a struct_time and UTC offset
    # as string "sHH:MM". This may throw an exception on get_local_time(),
    # it is NOT CAUGHT HERE, should be handled in the calling code because
    # different behaviors may be needed for some situations (e.g.
    # reschedule later).
    local_time_string = NETWORK.get_local_time()  # Sets RTC() time, but also
    # elements = local_time_string.split(" ")  # returns server response
    utc_offset = int(elements[-2])  # Format shHMM, e.g. -700 = -7 hr, 0 min
    # Pad UTC format shHMM to shHMM as needed for moon API 3.0
    utc_offset_string = "{:+03d}:{:02d}".format(utc_offset // 100,  # Hours
        abs(utc_offset) % 100)  # Mins
    return RTC().datetime, utc_offset_string

def hh_mm(time_struct):
    # Used for clock display elements, not for delta-time calculations.
Given a struct_time, return a string as H:MM or HH:MM, either 12- or 24-hour style depending on global TWELVE_HOUR setting. This is ONLY for 'clock time,' NOT for countdown time, which is handled separately in the one spot where it's needed.

```python
if TWELVE_HOUR:
    if time_struct.tm_hour > 12:
        hour_string = str(time_struct.tm_hour - 12) # 13-23 -> 1-11 (pm)
    elif time_struct.tm_hour > 0:
        hour_string = str(time_struct.tm_hour) # 1-12
    else:
        hour_string = '12' # 0 -> 12 (am)
else:
    hour_string = '{0:0>2}'.format(time_struct.tm_hour)
return hour_string + ':' + '{0:0>2}'.format(time_struct.tm_min)
```

```python
def parse_time_to_utc_seconds(time_local_string):
    # Given a string of YYYY-MM-DDTHH:MMsHH:MM or YYYY-MM-DDTHH:MM:SSZ
    # return equivalent UTC epoch seconds.
    # This could be UTC or local time, don't know yet, so no tag in var name
    # include no seconds value. Thus, only first two values are referenced:
    # HH:MM as encoded in string,
    # still could be UTC or local...
    # If not "Zulu time" (UTC), is local, so:
    # convert local to UTC
    return time.mktime(time.struct_time((int(date_str[0]),
                                           int(date_str[1]),
                                           int(date_str[2]),
                                           hour,
                                           minute,
                                           0, -1, -1, False)))
```

```python
class MoonData:
    # Class holding lunar data for a given 24-hour period. App uses two
    # one for the current day, and one for the following day,
    # then some interpolations and such can be made. Elements include:
    # Moon phase 'age' at start of period, expressed
    # from 0.0 (new moon) through 0.5 (full moon) to
    # 1.0 (next new moon).
    age               : Moon phase 'age' at start of period, expressed
    start_utc_seconds : Epoch time at start of period, UTC
    end_utc_seconds   : Epoch time at end of period,
    rise_utc_seconds  : Epoch time of moon rise within this 24-hour period
    set_utc_seconds   : Epoch time of moon set within this 24-hour period

    def __init__(self, datetime_local_struct, days_ahead, utc_offset_string):
        # Initialize MoonData elements (see above) given a struct_time,
        # days to skip ahead (typically 0 or 1), and a UTC offset (as a
        # string) and a query to the MET Norway Sunrise API (also provides
        # lunar data), documented at:
        # https://docs.api.met.no/doc/sunrise/celestial.html

        if days_ahead > 0:
            # Can't change attributes in struct_time, need to create a new
            # one which will roll the date ahead as needed. Convert to local
            # epoch seconds and back for the offset to work. :
            datetime_local_struct = time.localtime(
                time.mktime(time.struct_time((
                    datetime_local_struct.tm_year,
                    datetime_local_struct.tm_mon,
                    datetime_local_struct.tm_mday + days_ahead,
                    datetime_local_struct.tm_hour,
                    datetime_local_struct.tm_min,
                    datetime_local_struct.tm_wday,
                    datetime_local_struct.tm_yday,
                    datetime_local_struct.tm_isleap))))
```
datetime_local_struct.tm_hour,
datetime_local_struct.tm_min,
datetime_local_struct.tm_sec,
(-1, -1, -1))

# URL does not contain local or UTC time, only date. strftime() is
# not available in CircuitPython, manual conversion to time string
# is needed. Response is moon data for a 24-hour period, based on
# longitude and requested date. Some values within are UTC time,
# others are local. Anything we parse out of this will be converted
# to UTC epoch seconds, period.
url = ('https://api.met.no/weatherapi/sunrise/3.0/moon?lat=' +
str(LATITUDE) + '&lon=' + str(LONGITUDE) +
'&date=' + str(datetime_local_struct.tm_year) + '-' +
'{0:0>2}'.format(datetime_local_struct.tm_mon) + '-' +
'{0:0>2}'.format(datetime_local_struct.tm_mday) +
'&offset=' + utc_offset_string)
print('Fetching moon data via', url)

try:
    moon_data = NETWORK.fetch_data(url,
        json_path=[],
        headers = HEADERS)
    properties = moon_data['properties']
    # 0 = new moon, 90 = Q1, 180 = full moon, 270 = LQ
    self.age = float(properties['moonphase']) / 360
    interval = moon_data['when']['interval']
    self.start_utc_seconds = parse_time_to_utc_seconds(interval[0])
    self.end_utc_seconds = parse_time_to_utc_seconds(interval[1])
    # Thx user sandorcourane for the properties fixes!
    if properties['moonrise']['time'] is not None:
        self.rise_utc_seconds = parse_time_to_utc_seconds(
            properties['moonrise']['time'])
    else:
        self.rise_utc_seconds = None
    if properties['moonset']['time'] is not None:
        self.set_utc_seconds = parse_time_to_utc_seconds(
            properties['moonset']['time'])
    else:
        self.set_utc_seconds = None
    return # Success!
except:
    # Moon server error (maybe), try again after 15 seconds.
    # (Might be a memory error, that should be handled different)
    time.sleep(15)

# ONE-TIME INITIALIZATION
--------------------------------------------------------------

MATRIX = Matrix(bit_depth=BITPLANES)
DISPLAY = MATRIX.display
ACCEL = adafruit_lis3dh.LIS3DH_I2C(busio.I2C(board.SCL, board.SDA),
    address=0x19)
    _ = ACCEL.acceleration # Dummy reading to blow out any startup residue
    time.sleep(0.1)
DISPLAY.rotation = (int(((math.atan2(-ACCEL.acceleration.y,
    -ACCEL.acceleration.x) + math.pi) /
    (math.pi * 2) + 0.875) * 4) % 4) * 90

LARGE_FONT = bitmap_font.load_font('/fonts/helvB12.bdf')
SMALL_FONT = bitmap_font.load_font('/fonts/helvR10.bdf')
SYMBOL_FONT = bitmap_font.load_font('/fonts/6x10.bdf')
LARGE_FONT.load_glyphs('0123456789:.
SMALL_FONT.load_glyphs('0123456789:/.%')
SYMBOL_FONT.load_glyphs('↥↧')

# Display group is set up once, then we just shuffle items around later.
# Order of creation here determines their stacking order.
GROUP = displayio.Group()
try:
    FILENAME = 'moon/splash-' + str(DISPLAY.rotation) + '.bmp'

    # CircuitPython 6 & 7 compatible
    BITMAP = displayio.OnDiskBitmap(open(FILENAME, 'rb'))
    TILE_GRID = displayio.TileGrid(
        BITMAP,
        pixel_shader=getattr(BITMAP, 'pixel_shader', displayio.ColorConverter())
    )

    # # CircuitPython 7+ compatible
    # BITMAP = displayio.OnDiskBitmap(FILENAME)
    # TILE_GRID = displayio.TileGrid(BITMAP, pixel_shader=BITMAP.pixel_shader)

    GROUP.append(TILE_GRID)
except:
    GROUP.append(adafruit_display_text.label.Label(SMALL_FONT, color=0xFF0000, text='AWOO'))
    GROUP[0].x = (DISPLAY.width - GROUP[0].bounding_box[2] + 1) // 2
    GROUP[0].y = DISPLAY.height // 2 - 1

# Elements 1-4 are an outline around the moon percentage -- text labels
# offset by 1 pixel up/down/left/right. Initial position is off the matrix,
# updated on first refresh. Initial text value must be long enough for
# longest anticipated string later.
for i in range(4):
    GROUP.append(adafruit_display_text.label.Label(SMALL_FONT, color=0, text='99.9%', y=-99))

# Element 5 is the moon percentage (on top of the outline labels)
GROUP.append(adafruit_display_text.label.Label(SMALL_FONT, color=0xFF0000, text='99.9%', y=-99))

# Element 6 is the current time
GROUP.append(adafruit_display_text.label.Label(LARGE_FONT, color=0x808080, text='12:00', y=-99))

# Element 7 is the current date
GROUP.append(adafruit_display_text.label.Label(SMALL_FONT, color=0x808080, text='12/31', y=-99))

# Element 8 is a symbol indicating next rise or set
GROUP.append(adafruit_display_text.label.Label(SYMBOL_FONT, color=0x00FF00, text='x', y=-99))

# Element 9 is the time of (or time to) next rise/set event
GROUP.append(adafruit_display_text.label.Label(SMALL_FONT, color=0x00FF00, text='12:00', y=-99))

DISPLAY.show(GROUP)

NETWORK = Network(status_neopixel=board.NEOPIXEL, debug=False)
NETWORK.connect()

# LATITUDE, LONGITUDE, TIMEZONE are set up once, constant over app lifetime

# Fetch latitude/longitude from secrets.py. If not present, use
# IP geolocation. This only needs to be done once, at startup!
try:
    LATITUDE = secrets['latitude']
    LONGITUDE = secrets['longitude']
    print('Using stored geolocation: ', LATITUDE, LONGITUDE)
except KeyError:
    LATITUDE, LONGITUDE = (
        NETWORK.fetch_data('http://www.geoplugin.net/json.gp',
        json_path=[['geoplugin_latitude'],
                    ['geoplugin_longitude']])
        print('Using IP geolocation: ', LATITUDE, LONGITUDE)

# Set initial clock time, also fetch initial UTC offset while
# here (NOT stored in secrets.py as it may change with DST).
# pylint: disable=bare-except
try:
    DATETIME_LOCAL_STRUCT, UTC_OFFSET_STRING = update_system_time()
except:
    DATETIME_LOCAL_STRUCT, UTC_OFFSET_STRING = time.localtime(), '+00:00'
LAST_SYNC_LOCAL_SECONDS = time.mktime(DATETIME_LOCAL_STRUCT)

# Poll server for moon data for current 24-hour period and +24 ahead
PERIOD = []
for DAY in range(2):  # Today, tomorrow
    PERIOD.append(MoonData(DATETIME_LOCAL_STRUCT, DAY, UTC_OFFSET_STRING))
# PERIOD[0] is a current 24-hour time period we're in. PERIOD[1] is the
# 24 hours following that. Start/end time thresholds vary by longitude.
# Any values within the object are expressed in UTC seconds. Data is
# shifted down and new data fetched as days expire. Thought we might need a
# PERIOD[2] for certain circumstances but it appears not, that's changed
# easily enough if needed.

# MAIN LOOP ##########################################################
while True:
    gc.collect()
    NOW_LOCAL_SECONDS = time.time()  # Current local epoch time in seconds

    # Sync with time server every ~3 hours
    if NOW_LOCAL_SECONDS - LAST_SYNC_LOCAL_SECONDS > 3 * 60 * 60:
        try:
            DATETIME_LOCAL_STRUCT, UTC_OFFSET_STRING = update_system_time()
            LAST_SYNC_LOCAL_SECONDS = time.mktime(DATETIME_LOCAL_STRUCT)
            continue  # Time may have changed; refresh NOW_LOCAL_SECONDS value
        except:
            # update_system_time() can throw an exception if time server doesn't
            # respond. That's OK, keep running with our current time, and
            # push sync time ahead to retry in 30 minutes (don't overwhelm
            # the server with repeated queries).
            LAST_SYNC_LOCAL_SECONDS += 30 * 60  # 30 minutes -> seconds

    # NOW_LOCAL_SECONDS and DATETIME_LOCAL_STRUCT are local time, while all
    # moon properties are UTC. Convert 'now' to UTC seconds...
    # UTC_OFFSET_STRING is a string, like +HH:MM. Convert to integer seconds:
    hhmm = UTC_OFFSET_STRING.split(':')
    utc_offset_seconds = ((int(hhmm[0]) * 60 + int(hhmm[1])) * 60)
    NOW_UTC_SECONDS = NOW_LOCAL_SECONDS - utc_offset_seconds

    # If PERIOD has expired, move data down and fetch new +24-hour data
    if NOW_UTC_SECONDS >= PERIOD[0].end_utc_seconds:
        PERIOD[0] = PERIOD[1]
        PERIOD[1] = MoonData(time.localtime(), 1, UTC_OFFSET_STRING)

    # Determine weighting of tomorrow's phase vs today's, using current time
    RATIO = ((NOW_UTC_SECONDS - PERIOD[0].start_utc_seconds) /  
              (PERIOD[1].start_utc_seconds - PERIOD[0].start_utc_seconds))

    # Determine moon phase 'age'
    # 0.0  = new moon
    # 0.25 = first quarter
    # 0.5  = full moon
    # 0.75 = last quarter
    # 1.0  = new moon
    if PERIOD[0].age < PERIOD[1].age:
        AGE = (PERIOD[0].age +  
               (PERIOD[1].age - PERIOD[0].age) * RATIO) % 1.0
    else:  # Handle age wraparound (1.0 -> 0.0)
        # If tomorrow's age is less than today's, it indicates a new moon
        # crossover. Add 1 to tomorrow's age when computing age delta.
        AGE = (PERIOD[0].age +  
               (PERIOD[1].age + 1 - PERIOD[0].age) * RATIO) % 1.0

    # AGE can be used for direct lookup to moon bitmap (0 to 99) -- these
    # images are pre-rendered for a linear timescale (solar terminator moves
# nonlinearly across sphere).
FRAME = int(AGE * 100) % 100 # Bitmap 0 to 99

# Then use some trig to get percentage lit
if AGE <= 0.5: # New -> first quarter -> full
    PERCENT = (1 - math.cos(AGE * 2 * math.pi)) * 50
else:          # Full -> last quarter -> new
    PERCENT = (1 + math.cos((AGE - 0.5) * 2 * math.pi)) * 50

# Find next rise/set event, complicated by the fact that some 24-hour
# periods might not have one or the other (but usually do) due to the
# Moon rising ~50 mins later each day. This uses a brute force approach,
# working through the time periods to locate rise/set events that
# A) exist in that 24-hour period (are not None), B) are still in
# the future, and C) are closer than the last guess. What’s left at the
# end is the next rise or set time, and a flag whether the moon’s
# currently risen or not.
NEXT_EVENT_UTC_SECONDS = NOW_UTC_SECONDS + 300000 # Way future
for DAY in PERIOD:
    if (DAY.rise_utc_seconds and
        NOW_UTC_SECONDS < DAY.rise_utc_seconds < NEXT_EVENT_UTC_SECONDS):
        NEXT_EVENT_UTC_SECONDS = DAY.rise_utc_seconds
        RISEN = False # Current moon state; next event is inverse
    if (DAY.set_utc_seconds and
        NOW_UTC_SECONDS < DAY.set_utc_seconds < NEXT_EVENT_UTC_SECONDS):
        NEXT_EVENT_UTC_SECONDS = DAY.set_utc_seconds
        RISEN = True

if DISPLAY.rotation in (0, 180): # Horizontal 'landscape' orientation
    CENTER_X = 48      # Text along right
    MOON_Y = 0         # Moon at left
    TIME_Y = 6         # Time at top right
    EVENT_Y = 26       # Rise/set at bottom right
else:
    CENTER_X = 16      # Text down center
    if RISEN:
        MOON_Y = 0     # Moon at top
        EVENT_Y = 38   # Rise/set in middle
        TIME_Y = 49    # Time/date at bottom
    else:
        TIME_Y = 6     # Time/date at top
        EVENT_Y = 26   # Rise/set in middle
        MOON_Y = 32    # Moon at bottom

print()

# Update moon image (GROUP[0])
FILENAME = 'moon/moon' + '{0:0>2}'.format(FRAME) + '.bmp'

# CircuitPython 6 & 7 compatible
# BITMAP = displayio.OnDiskBitmap(open(FILENAME, 'rb'))
# TILE_GRID = displayio.TileGrid(
#     BITMAP,
#     pixel_shader=getattr(BITMAP, 'pixel_shader',
#                         displayio.ColorConverter())
# )

# CircuitPython 7+ compatible
BITMAP = displayio.OnDiskBitmap(FILENAME)
TILE_GRID = displayio.TileGrid(BITMAP, pixel_shader=BITMAP.pixel_shader)

TILE_GRID.x = 0
TILE_GRID.y = MOON_Y
GROUP[0] = TILE_GRID

if PERCENT >= 99.95:
    STRING = '100%
else:
STRING = '{:.1f}'.format(PERCENT + 0.05) + '%

print(NOW_UTC_SECONDS, STRING, 'full')

# Set element 5 first, use its size and position for setting others
GROUP[5].text = STRING
GROUP[5].y = MOON_Y + 16
for _ in range(1, 5):
    GROUP[_].text = GROUP[5].text
    GROUP[_].x, GROUP[_].y = GROUP[5].x, GROUP[5].y - 1  # Up 1 pixel
GROUP[2].x, GROUP[2].y = GROUP[5].x - 1, GROUP[5].y  # Left
GROUP[3].x, GROUP[3].y = GROUP[5].x + 1, GROUP[5].y  # Right
GROUP[4].x, GROUP[4].y = GROUP[5].x, GROUP[5].y + 1  # Down

# Update next-event time (GROUP[8] and [9])
NEXT_EVENT_LOCAL_STRUCT = time.localtime(NEXT_EVENT_UTC_SECONDS + utc_offset_seconds)  # Need later
if COUNTDOWN:  # Show NEXT_EVENT_UTC_SECONDS as countdown to event
    MINUTES = (NEXT_EVENT_UTC_SECONDS - NOW_UTC_SECONDS) // 60
    STRING = str(MINUTES // 60) + ':' + '{0:0>2}'.format(MINUTES % 60)
else:  # Show NEXT_EVENT_UTC_SECONDS in clock time
    STRING = hh_mm(NEXT_EVENT_LOCAL_STRUCT)
GROUP[9].text = STRING
XPOS = CENTER_X - (GROUP[9].bounding_box[2] + 6) // 2
GROUP[8].x = XPOS

if RISEN:  # Next event is SET
    GROUP[8].text = '\u21A7'  # Downwards arrow from bar
    GROUP[8].y = EVENT_Y - 2
    print('Sets:', STRING)
else:  # Next event is RISE
    GROUP[8].text = '\u21A5'  # Upwards arrow from bar
    GROUP[8].y = EVENT_Y - 1
    print('Rises:', STRING)
GROUP[9].x = XPOS + 6
GROUP[9].y = EVENT_Y

GROUP[8].color = GROUP[9].color = (0x00FF00 if
    NEXT_EVENT_LOCAL_STRUCT.tm_hour < 12
else 0xC04000)

# Update time (GROUP[6]) and date (GROUP[7])
NOW_LOCAL_STRUCT = time.localtime()
STRING = hh_mm(NOW_LOCAL_STRUCT)
GROUP[6].text = STRING
GROUP[6].y = TIME_Y
if MONTH_DAY:
    STRING = (str(NOW_LOCAL_STRUCT.tm_mon) + '/' +
        str(NOW_LOCAL_STRUCT.tm_mday))
else:
    STRING = (str(NOW_LOCAL_STRUCT.tm_mday) + '/' +
        str(NOW_LOCAL_STRUCT.tm_mon))
GROUP[7].text = STRING
GROUP[7].x = CENTER_X - GROUP[7].bounding_box[2] // 2
GROUP[7].y = TIME_Y + 10
DISPLAY.refresh()  # Force full repaint (splash screen sometimes sticks)
time.sleep(5)

One additional file you'll create or edit yourself — secrets.py — is explained on the next page...
Configure Network and Location

secrets.py on the CIRCUITPY drive holds your WiFi network and Adafruit IO credentials and other info. This file can be created or edited with any simple text editor you prefer. It resides in the root directory of the CIRCUITPY drive, not inside a folder.

If you already have this file on your Matrix Portal from prior projects...great!

If not, it should resemble what's below. Some items are strings (in single quotes), others are numbers (no quotes).

The format of this file is super persnickety, every space and comma counts! If creating it for the first time, best to copy-and-paste the text below exactly, then change any items of interest (preserving quotation marks and such).

ssid and password are your WiFi network name and password.

aio_username and aio_key can be found by logging into your Adafruit IO account and clicking the yellow key icon near the top-right. If you don’t yet have an account, set one up, the basic service is free! This is used by the software to set the clock.

timezone is needed to distinguish your local time from UTC (aka Greenwich time). Here’s a list of valid time zone strings; find a city in your zone and copy/paste it here (in quotes).

latitude and longitude are optional but recommended. The clock can do IP geolocation (estimating your location using IP address), but having it here is more accurate. These values are floating-point numbers (in degrees and decimal fractions thereof) and should not be in quotes. For latitude, positive values are north, negative is south. For longitude, negative is west, positive is east.

```python
secrets = {
    'ssid'         : 'WiFi-Network-Name',
    'password'     : 'WiFi-Network-Password',
    'latitude'     : 32.71193,
    'longitude'    : -117.16072,
    'aio_username' : 'Your-AIO-Username',
    'aio_key'      : 'Your-AIO-Key',
    'timezone'     : 'America/Los_Angeles' # http://worldtimeapi.org/timezones
}
```

You can easily get your decimal latitude and longitude from Google Maps. Zoom in on a location and right-click on the map. Information there includes latitude and
longitude in exactly the format we need…click the coordinates to copy them to the clipboard, then paste into the secrets file and reformat the text as shown above.

The code will relaunch any time there’s a change on the CIRCUITPY drive…so, after editing secrets.py, the clock should start up on its own and you’ll see a splash screen after a few seconds.

If the clock does NOT start up, or shows the splash screen but then crashes: most likely the WiFi credentials are incorrect, or something is wrong with the secrets.py file syntax…make sure every quote, comma and colon is there and in the right place.

Clock Usage

On power-up, the clock will display a splash screen while it connects to the WiFi network and makes a couple of initial server requests for time and Moon data.

Time is provided by Adafruit IO (), and astronomical data from the Norwegian Meteorological Institute (). These two sources were chosen as they are free and do not require a paid account for access.

The orientation of the clock at startup determines how information is displayed — in a vertical “portrait” format, or horizontal “landscape” format. There’s different artwork for all four directions!
Clock Operation Basics

With the clock set up horizontally (either hung or propped up with a stand or case), the current Moon phase (with percentage lit) is displayed on the left.

The current time and date are displayed on the right. Below this is an indication of the next moonrise or set...

A downward-pointing arrow (as shown above) indicates the next lunar event will be a moonset — that the Moon is currently above the horizon (in the sky) as seen from your position, regardless of phase.

An upward-pointing arrow indicates a moonrise is next — the Moon is currently below the horizon, not currently visible from your position, regardless of phase.

The time shown next to the arrow is when this next lunar rise or set event will occur. It can alternately be configured as a countdown, explained shortly.

The color of the time and arrow indicates whether this next event occurs in the a.m. hours (green) or p.m. (amber).

The rise and set times are for an “idealized” horizon. Hills or other local obstructions could have it rising a short time later or setting earlier.

All the same data is displayed when the clock’s in a vertical orientation, but the visibility of the Moon from your location is made more apparent.

When the Moon is above the local horizon (in the sky) the phase is shown at the top of the matrix.
And if the Moon is currently below the local horizon, the phase scoots to the bottom.

The Moon might still appear above the horizon for someone to the west of you, but the phase will be nearly the same. That's how this game of cosmic billiards works.

The percentage shown is the amount of the lunar “disc” that’s currently illuminated, not the “age” of the current lunar month. So this will go from 0% (new Moon) to 100%* (full Moon) and back down to 0% (next new Moon) over the course of the lunar month. The lunar terminator — the line between light and dark — always moves right to left, like turning the pages of a book (that’s why it doesn’t bother reporting if the phase is waxing or waning — it’s easy to visualize).

* It’s normal some months that you’ll see this only go to 99.8 or 99.9 percent...the Moon’s never quite perfectly full () from our position.

If the clock starts up in the wrong orientation...just tap the reset button and let it try again. Occasionally the accelerometer gets confused when grappling the clock to plug it in, or the “snap” of a USB cable being inserted.

**Customizing the Clock Display**

Some aspects of the clock display can be changed by editing the file code.py in your text editor of choice. Starting around line 35, look for this section of code:

```python
# CONFIGURABLE SETTINGS -------------------------------------
TWELVE_HOUR = True  # If set, use 12-hour time vs 24-hour
COUNTDOWN = False  # If set, show time to next rise/set event
```

Change the value of `TWELVE_HOUR` to `False` (capitalized — Python is case sensitive) to have the clock display times in 24-hour military time.

Changing `COUNTDOWN` to `True` makes the clock show the time to the next rise/set event (hours and minutes), rather than the time of the event. Both modes are shown in the same HH:MM format, so COUNTDOWN mode may be confusing if you’re not familiar with the clock’s operation.
The startup splash screen images are inside the moon folder, named splash-0.bmp, splash-90.bmp, splash-180.bmp and splash-270.bmp — one image is selected on startup based on the Matrix Portal orientation. If you’ve settled on a position for installing your Moon clock but want the opposite splash image, you can just rename these files, e.g. swap the filenames on splash-0.bmp and splash-180.bmp (or 90 and 270). Or you can delete the splash images (they might be frightening to children) or substitute your own BMP files.

Matrix brightness is not currently adjustable. It’s just more than the microcontroller can handle.

**Timekeeping**

The clock will periodically contact a time server to keep itself in sync. But if you observe the time drifting by more than a minute or two, you can have it sync more often.

Look for these lines in code.py (in the “MAIN LOOP” section):

```python
# Sync with time server every ~3 hours
if NOW_LOCAL_SECONDS - LAST_SYNC_LOCAL_SECONDS > 3 * 60 * 60:
```

Change the 3 to a smaller value...perhaps 2, or 1 if the time drift is really pronounced. This is the period (in hours) between clock updates. To avoid over-taxing the server, provided as a free service, don’t set this too low.

Interestingly, the clock will keep better time if powered from a computer’s USB port (which provides a steady 1.000 KHz reference pulse) than from a passive USB “wall wart.” Not always practical, of course...but if you have a free USB port on a NAS or other nearby system that runs 24/7, it’s something to consider.
Moon Facts

So what's the big deal? What's wrong with the Moon phases on wall calendars? Surely those dates come from trusted sources! NASA, almanacs and stuff!

The problems are many...some technical, some semantic...each alone may be minor, but there's the potential for them to compound and really mix things up.

• Most importantly, unless they explain their data sources, static media can’t take into account different time zones. At any given moment, the Moon's phase appears nearly the same from anywhere on Earth (if it's above the local horizon), but the time of day...and even the day itself...may vary. If a calendar is basing their phases on UTC time but you’re in North America, they might be reporting a lunar phase on a Thursday while to you it’s still late Wednesday (this can happen in the other direction too).

• Each day's changeover at midnight doesn't align with our semantic concept of a “night” — we tend to lump early morning hours with the prior day. So, if a lunar phase crossing occurs in the early hours after midnight (for a given time zone), it might be reported as Thursday, while we’re inclined to think of it as late Wednesday night...but, seeing Thursday on the calendar, we think “Thursday night” and could easily end up celebrating a “full” Moon that’s past its ideal freshness date.

• What even is “full,” or any other phase of the Moon? It's erroneous to think there’s a single night that the Moon is locked in “full,” because this is an analog system with celestial bodies in continual motion, and phase can change by several percent in a single 24-hour period. To the unaided eye, anything more than about 98% illumination of the lunar disc is pretty much indistinguishable from “full,” and you’ll get two to three successive nights that would qualify. Those dates on the calendar typically refer to an “instantaneous” phase crossing
(and once again, for whatever time zone they’re using for reference). If at that moment the Moon is below the horizon (is on the opposite side of the planet from you, due to the Earth’s rotation), and if your definition of “full” is too narrow...have you really experienced a full Moon at all? Sometimes it’s okay to think of a phase as “ish.”

- Sometimes just sloppy reporting...the instantaneous time of a phase crossing is sometimes improperly reported as the Moon rise time instead, and other slip-ups.

As you can see, it’s A Huge Ordeal, and that was the inspiration for making this clock. No more “do they mean Wednesday night or Thursday morning?” This is the Moon as it is right now.

Other Moon Factoids

- We tend to think of the Moon as a nighttime phenomenon, but really it spends just as much time on the daylight side. During a new Moon, it rises and sets close to the same time as the Sun, the narrow crescent lost in the glare of the daylight sky.
- The cycle of phases — the “lunar month” — is about 29.5 days long. It varies a bit due to the elliptical nature of orbits and that the Earth-Moon system is in turn orbiting the Sun.
- If you really want to get lost in all the details, NASA’s Dial-A-Moon page is packed with information and nuance!
- A lunar eclipse can only happen during the full Moon, but not all full Moons experience a lunar eclipse ... the orbital planes of the Earth and Moon diverge by a few degrees. Correspondingly, a solar eclipse can only happen during the new Moon, but not all new Moons experience a solar eclipse. (WolfCop got this wrong, with a solar eclipse the day after a full moon, and I’m still bitter about it.)
- Some full moons have special names — a “blue Moon” is an infrequent second full Moon in the same month, the “pink Moon” is usually the full Moon in April — but these are only names and are not actually descriptive of the Moon’s color. Sometimes atmospheric phenomena (smoke, etc.) can cast the Moon in varying hues, but it’s entirely unrelated to these names.