Humidity and Temperature Monitor with E-Ink Display
Created by Dave Astels

Last updated on 2019-04-12 10:19:02 PM UTC
Overview

Depending on your climate (and HVAC system) dried herbs can go moldy. A big cause of mold is too high of humidity. That's the reason for all those silica gel packets in items you get. If you store a lot of dried herbs or anything that requires a controlled humidity (and/or temperature) environment, this project can help by monitoring the temperature and humidity in a container and warn you if they go out of the desired range.

In a previous guide: Storage Humidity and Temperature Monitor (https://adafruit.it/BIS), we built a simple monitoring project. Specifically, for a small jar. It was really simple, using a Trinket M0 to drive it, a SI7021 breakout to take measurements, and a TPL5111 breakout to manage power. The onboard DotStar and a piezo buzzer were used to indicate status. The code was written in C.

That was built in late 2017. Fast forward (at least it seems fast) to February 2019 (a bit over a year later) and it was time to revisit the design.

The SI7021 is a reliable, robust sensor that the author has used in various projects. All we require is temperature and humidity measurements, and the I2C interface is simple to work with. We’ll keep that.

In place of a Trinket M0 we’ll use a Feather M4 Express. That's a big jump in power, but this board is reasonably priced and is the author's go-to board for most projects. A huge advantage is the onboard battery charging circuit. This will allow the battery to be charged without having to disconnect and remove it. The Feather also has far more I/O pins than the Trinket. This makes possible the next change.

With the Feather, we can do something better for status indication. It was decided to use an eInk display (which uses SPI and 5 digital I/O lines). The 1.54" square 3-color display was selected. An eInk display is a huge benefit on a device like this. It wakes up approximately every hour to take readings, update the display, and possibly raise an alarm. It's not powered the rest of the time. Using eInk provides a persistent display that remains "on" and informative all the time without requiring power. The downside of eInk is that screen refresh is a slow process. But when you only need to update every hour, that's not a problem.

The previous project used the TPL5110 breakout which directly controls power to the circuit. Since the Feather has an enable pin to control power, the TPL5111 breakout is used in this version.
We keep the piezo buzzer as an out-of-range signal, and add a pushbutton switch to provide input for cancelling the alarm as well as entering and exiting edit mode (we’ll talk about that later).

Finally, since the Feather M4 was selected, it was made a project goal to write the software in CircuitPython rather than C.

Required Parts

- **Adafruit Feather M4 Express - Featuring ATSAMD51**
  - $22.95
  - IN STOCK
  - [ADD TO CART](#)

- **Adafruit 1.54” Tri-Color eInk / ePaper Display with SRAM**
  - $22.50
  - IN STOCK
  - [ADD TO CART](#)

- **Adafruit TPL5111 Low Power Timer Breakout**
  - $5.95
  - IN STOCK
  - [ADD TO CART](#)
You'll need your choice of SPST momentary pushbutton switch. The one below is a lovely option for breadboarding.
1 x **Resistor**  
120K 1/4w resistor  

BUY NOW

1 x **Capacitor**  
100nF (aka 0.1uF) capacitor  

ADD TO CART

1 x **Prototyping perf-board**  
An assortment of perfboard for prototyping  

BUY NOW
Hardware

Above is the wiring diagram. The schematic is at the bottom of this page and makes a little more sense in some ways. Let's look at the pieces of the circuit around the Feather, one piece at a time.

Power

The circuit is powered by a LiPo battery. A 1200mAh model was selected based on size and charge lifetime.

Power is controlled by a TPL5111 breakout which controls the Feather's Enable input. While it can be used as is, with the onboard potentiometer (to set the sleep duration) is hard to get exact. The alternative is to use an external resistor which is what we do. To do this, we need to cut the Trim trace under the TPL5111 breakout. The other (LED) trace can be cut as well to save power and since the "on" LED won't be of much use.

Now let's talk about the startup sequence of the system. The first thing the code does at the earliest opportunity is set the DONE signal to the TPL5111 low so as to not turn off the power and start the timing sequence. The problem is that the SAMD51 output floats high. That along with the fact that the TPL5111 starts up faster than CircuitPython can configure and set the DONE output low means it will shut down before getting started (because it sees a high DONE signal). To avoid that we need to delay that DONE signal registering as True/high. For that we turn to our friend the capacitor.

By using a small capacitor between the DONE pin and ground, the pin won't be reach a voltage that registers as True for a short length of time: the amount of time that the capacitor takes to charge. A little trial and error showed that 100nF did the job. It delays DONE long enough for the code to configure the output and set it to False/low. This lets the TPL5111 stay in on mode until the DONE line is explicitly set to True/high by the code.
Measurement

The Si7021 is simple: power, ground and the pair of I2C signals. Nothing unusual about the connection. Actual wiring is a bit more involved. The sensor breakout has to be inside the container (a jar in the target build), preferably in an airtight way since a sealed environment is desired.

Display

The display is connected simply as well: power, ground, the SPI clock and data lines, as well as various control and status lines as shown. Note that we’re not using the SD card in this project.

Miscellaneous

The remaining pieces are a push button switch and a piezo. Note that this is a buzzer that is driven by a square wave (via a PWM) and not one that makes sound when power is applied.
Using a SAMD51 based board gives plenty of memory and speed for writing in CircuitPython. Because of that it was decided from the start to write the code in CircuitPython.

Getting Familiar

CircuitPython is a programming language based on Python, one of the fastest growing programming languages in the world. It is specifically designed to simplify experimenting and learning to code on low-cost microcontroller boards. Here are some guides which cover the basics:

- Welcome to CircuitPython! (https://adafru.it/cpy-welcome)
- Adafruit Feather M4 Express (https://adafru.it/CJN)

Be sure you have the latest CircuitPython loaded onto your board per the second guide.

CircuitPython is easiest to use within the Mu Editor. If you haven't previously used Mu, this guide will get you started (https://adafru.it/ANO).

Download Library Files

Plug your Feather M4 Express board into your computer via a USB cable. Please be sure the cable is a good power+data cable so the computer can talk to the Feather board.

A new disk should appear in your computer's file explorer/finder called CIRCUITPY. This is the place we'll copy the code and code library. If you can only get a drive named CPLAYBOOT, load CircuitPython per the guide above.

Create a new directory on the CIRCUITPY drive named lib.

Download the latest CircuitPython driver package to your computer using the green button below. Match the library you get to the version of CircuitPython you are using. Save to your computer's hard drive where you can find it.

https://adafru.it/zB-
https://adafru.it/zB-

With your file explorer/finder, browse to the bundle and open it up. Copy the following folder from the library bundle to your CIRCUITPY lib directory you made earlier:
All of the other necessary code is baked into CircuitPython!

Your CIRCUITPY drive should look like the snapshot below.

Download Code

Below is the code for this project. Select download project zip below and save it to your computer's hard drive where you can find it.
**Code Operation**

**Startup**

As mentioned earlier, we want to configure and set the DONE signal low as soon as possible after the runtime hands control to our code.

```python
import digitalio
import board

done = digitalio.DigitalInOut(board.A4)
done.direction = digitalio.Direction.OUTPUT
done.value = False
```

Once that's done, the usual imports and setup can be done. There are no surprises here: display, sensor, buzzer, and switch are set up as you would expect. The display setup is covered in its tutorial guide (https://adafruit.it/DQ0). Likewise, see the guide for the sensor (https://adafruit.it/DQ1).

```python
import time
import pulseio
import busio
from adafruit_epd.epd import Adafruit_EPD
from adafruit_epd.il0373 import Adafruit_IL0373
import adafruit_si7021
import font

spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
ces = digitalio.DigitalInOut(board.D11)
dc = digitalio.DigitalInOut(board.D10)
srcs = digitalio.DigitalInOut(board.D9)
rst = digitalio.DigitalInOut(board.D6)
busy = digitalio.DigitalInOut(board.D12)
display = Adafruit_IL0373(152, 152, rst, dc, busy, srcs, ecs, spi)

i2c = busio.I2C(board.SCL, board.SDA)
sensor = adafruit_si7021.SI7021(i2c)

ON = 2**15
OFF = 0

buzzer = pulseio.PWMOut(board.D5, variable_frequency=True)
buzzer.duty_cycle = OFF

silence_button = digitalio.DigitalInOut(board.A5)
silence_button.direction = digitalio.Direction.INPUT
silence_button.pull = digitalio.Pull.UP
```

**Settings**

Various aspects of the system's behavior are controlled by a collection of settings. These are read from the settings.txt file on the CIRCUITPY drive. Once read they are stored in a dictionary which is initialized with default values after the hardware is set up.
\begin{verbatim}
settings = {
    'temperature_range': (15, 30),
    'humidity_range': (60, 70),
    'title': 'Jar Minder',
    'alarm_frequency': 4000,
    'alarm_number_of_beeps': 3,
    'alarm_seconds_beep_on': 0.5,
    'alarm_seconds_between_beeps': 0.5,
    'alarm_seconds_between_alarms': 5.0,
    'alarm_timeout': 60.0
}
\end{verbatim}

\textbf{temperature_range} takes the low and high bounds of the acceptable temperature range: two integers separated by a dash.

\textbf{humidity_range} is similar for the humidity range.

\textbf{title} is a string (at most 19 characters with no surrounding quotes) that is shown at the top of the display.

\textbf{alarm_frequency} is the frequency of alarm beeps as an int.

\textbf{alarm_number_of_beeps} is the number of beeps in an alarm as an int.

\textbf{alarm_seconds_beep_on} the amount of time each alarm beep sounds (seconds, a float).

\textbf{alarm_seconds_between_beeps} the amount of time between alarm beeps (seconds, a float).

\textbf{alarm_seconds_between_alarms} the amount of time between groups of alarm beeps (seconds, a float).

\textbf{alarm_timeout} how long (ins seconds) to continue sounding the alarm (float) after which the device will go to sleep.

Here's an example of the setting file:

\begin{verbatim}
temperature_range:15-30
humidity_range:60-70
title:Jar Minder
alarm_frequency:4000
alarm_number_of_beeps:3
alarm_seconds_beep_on:0.5
alarm_seconds_between_beeps:0.5
alarm_timeout:60.0
\end{verbatim}

The code that reads the setting file is pretty simple:
Along with the settings file, there is a file of text named `description.txt` which contains informative text to be displayed on the screen. It is limited to 4 lines of 19 characters each. This can be used to identify the contents, when it was put in the jar, etc. It gets read and stored for display:

```python
with open('description.txt', 'r') as f:
    text = [line.strip() for line in f]
```

There are a few functions to display text on the display:

```python
def to_int_tuple(a):
    """Convert an array of strings to a tuple of ints.
    :param [int] a: array of strings to convert
    """
    return tuple([int(x.strip()) for x in a])
```

```python
with open('settings.txt', 'r') as f:
    for line in f:
        key, value = line.strip().split(':').strip()
        values = value.split('-')
        if key == 'temperature_range':
            setting = to_int_tuple(values)
        elif key == 'humidity_range':
            setting = to_int_tuple(values)
        elif key == 'title':
            setting = value
        elif key == 'alarm_frequency':
            setting = int(value)
        elif key == 'alarm_number_of_beeps':
            setting = int(value)
        elif key == 'alarm_seconds_beep_on':
            setting = float(value)
        elif key == 'alarm_seconds_between_beeps':
            setting = float(value)
        elif key == 'alarm_timeout':
            setting = float(value)
        settings[key] = setting
```
def render_character(x, y, ch, color=Adafruit_EPD.BLACK):
    """Render a character.
    :param int x: horizontal position of the left edge of the character
    :param int y: vertical position of the top edge of the character
    :param str ch: a single character string to be displayed
    :param Adafruit_EPD.* color: BLACK or RED, background is always white
    ""
    if x < 144 and y < 144:
        bitmap = font.bitmaps[ord(ch)]
        for row_num in range(8):
            row = bitmap[row_num]
            for column_num in range(8):
                if (row & 1) == 0:
                    display.draw_pixel(x + column_num, y + row_num, Adafruit_EPD.WHITE)
                else:
                    display.draw_pixel(x + column_num, y + row_num, color)
                row >>= 1

def render_string(x, y, s, color=Adafruit_EPD.BLACK):
    """Render a string.
    :param int x: horizontal position of the left edge of the string
    :param int y: vertical position of the top edge of the string
    :param str ch: a string to be displayed
    :param Adafruit_EPD.* color: BLACK or RED, background is always white
    ""
    x_pos = x
    for ch in s:
        render_character(x_pos, y, ch, color)
        x_pos += 8

def centered(s):
    """Compute the X position to center a string.
    :param str s: the string to center
    ""
    return 75 - (4 * len(s))

The **centered** function determines where to start a line to center it horizontally. **render_string** iterates through a string, rendering each character by using **render_character**, moving across the screen as it goes. A character is rendered by setting each pixel in its 8x8 grid to white (the background, represented by a 0 bit in the font data) or the foreground color that is passed in (black or red).

Where do these character bitmaps come from? At the start we imported **font.py** which contains the bitmaps for the 127 ASCII characters. It's done as Python code that defines an array of arrays. Each inner array contains 8 bytes: each bit is a pixel in the character bitmap. A small section is shown below, and is based on **font8x8_basic.h** from [https://github.com/dhepper/font8x8](https://github.com/dhepper/font8x8) (which is in the public domain).
Other Support Functions

There are three other functions.

First is \texttt{check\_for\_push} which monitors the pushbutton for a specified period of time. Every 100mS during that time the button is checked. If it is pushed, \texttt{True} is immediately returned. If the time expires \texttt{False} is returned.

```python
def check_for_push(button, duration):
    """Wait for a time, regularly checking for a button push.
: param DigitalInOut button: the button input to check
: param float duration: seconds to wait
Return True if the button is pushed, False if the time passes
"""
    stop_at = time.monotonic() + duration
    while time.monotonic() < stop_at:
        if not button.value:
            return True
        time.sleep(0.1)
    return False
```

Next is \texttt{sound\_alarm} which is responsible for generating a series of alarm beeps as specified in the settings. The \texttt{check\_for\_push} function is called during each time of the beeps and the time between beeps. As soon as it returns \texttt{True}, the alarm sequence is terminated and \texttt{sound\_alarm} returns \texttt{True}.

```python
def sound_alarm():
    """Sound the alarm based on the settings."""

    buzzer.frequency = settings['alarm_frequency']
    for _ in range(settings['alarm_number_of_beeps']):
        buzzer.duty_cycle = ON
        if check_for_push(silence_button, settings['alarm_seconds_bEEP_on']):
            buzzer.duty_cycle = OFF
            return True
        buzzer.duty_cycle = OFF
        if check_for_push(silence_button, settings['alarm_seconds_between_bEEPs']):
            return True
    return False
```

Finally, there is \texttt{out\_of\_range} which takes temperature and humidity readings and checks them against the appropriate range. \texttt{True} is returned if either reading is out of range, \texttt{False} otherwise.
def out_of_range(t, h):
    """Check if either temperature and humidity is out of range.
    :param float t: temperature reading
    :param float h: humidity reading
    """
    if t < settings['temperature_range'][0]:
        return True
    if t > settings['temperature_range'][1]:
        return True
    if h < settings['humidity_range'][0]:
        return True
    if h > settings['humidity_range'][1]:
        return True
    return False

Edit Mode
When the system starts up the switch is checked. If it is pressed edit mode is entered. This is indicated by a low beep and the message EDIT MODE on the display. The system stays in edit mode until the switch is pressed again. Another low beep is sounded and the normal mode is entered. The purpose of this is to keep the power on while the user edits the setting and description files/

if not silence_button.value:
    buzzer.frequency = 440
    buzzer.duty_cycle = ON
    time.sleep(0.5)
    buzzer.duty_cycle = OFF
    display.clear_buffer()
    render_string(39, 64, 'EDIT MODE')
    display.display()
    while not silence_button.value:       # wait for button to be released
        pass
    while silence_button.value:           # wait for button to be pressed
        pass
    buzzer.duty_cycle = ON
    time.sleep(0.5)
    buzzer.duty_cycle = OFF

Main Script
After settings and description are loaded as described above, the display is updated based on readings. First the title and description:

display.clear_buffer()
render_string(centered(settings['title']), 12, settings['title'])

# Display text
row_index = 64
for line in text:
    if row_index > 112:
        break
    render_string(centered(line), row_index, line)
    row_index += 10
Next the sensor is read and the values displayed.

```python
temperature = int(sensor.temperature)
humidity = int(sensor.relative_humidity)
render_string(8, 32, '{0:2d} C'.format(temperature))
render_string(112, 32, '{0:2d} %'.format(humidity))
```

Once we have the readings, they are checked against their respective ranges and explanatory text assigned. If a reading is in range, this will be an empty string.

```python
if temperature < settings['temperature_range'][0]:
    temperature_message = 'LOW TEMPERATURE'
elif temperature > settings['temperature_range'][1]:
    temperature_message = 'HIGH TEMPERATURE'
else:
    temperature_message = ''
if humidity < settings['humidity_range'][0]:
    humidity_message = 'LOW HUMIDITY'
elif humidity > settings['humidity_range'][1]:
    humidity_message = 'HIGH HUMIDITY'
else:
    humidity_message = ''
```

If a reading is out of range, the associated text is displayed in red and red bars are displayed at the top and bottom of the screen.

```python
if temperature_message:
    render_string(centered(temperature_message), 122, temperature_message, Adafruit_EPD.RED)
if humidity_message:
    render_string(centered(humidity_message), 132, humidity_message, Adafruit_EPD.RED)
if temperature_message or humidity_message:
    display.fill_rect(0, 0, 152, 10, Adafruit_EPD.RED)
    display.fill_rect(0, 142, 152, 10, Adafruit_EPD.RED)
    display.display()
```

Finally, we handle sounding the alarm. While either reading is out of range and the alarm timeout hasn't expired, the series of alarm beeps is generated each interval specified in the settings. Once the timeout expires or the switch is pressed, the alarm stops and the device powers down by setting the `DONE` signal `True`/high.

```python
timeout = time.monotonic() + settings['alarm_timeout']
while out_of_range(sensor.temperature, sensor.humidity) and time.monotonic() < timeout:
    if sound_alarm():
        break
    if check_for_push(silence_button, settings['alarm_seconds_between_alarms']):
        break
done.value = True
```
Prototype Construction

This version of the jar monitor is still in the prototype stage with a hand-wired version glued onto a 3D printed lid for a large mouth mason jar.

The Jar Minder prototype mounted on the printed jar lid.

The jar lid is based on https://www.thingiverse.com/thing:1323393/ (https://adafruit.it/DQ3) which is for a small sized mason jar. It was scaled on the X and Y axies to 123% to fit the larger size jars.
Closeups, including with the display removed.
For the unassembled prototype, nylon standoffs were superglued to the lid upon which the main board is mounted. Standoffs were bolted to the board, glue applied to their other ends, and the assembly positioned and left to set.

Point-to-point wiring was used, and everything was mounted to a piece of perf-board. The LED and potentiometer traces on the underside of the TPL5111 breakout were cut to disable the LED and allow the use of an external, fixed timing resistor which can be seen along with the DONE capacitor.

The wire below was used to connect the sensor to the Feather.
Wrap Up

This guide has revisited a previous project using more powerful and feature rich hardware. The switch to a Feather M4 Express has allowed the use of a feature-rich device coded in CircuitPython. The Feather's generous I/O capabilities has made possible the use of an eInk display which gives the ability to display readings as well as informative and warning text. The Feather also provides in-place battery charging which makes the device far more convenient.

Going Further

There are a few ways to expand the project. First is making a printed enclosure that extends or mounts on the jar lid. Next is making a custom PCB rather than using hand-wired perf-board construction. Another direction is to enhance the display. A larger font would help with legibility, as would some icons.