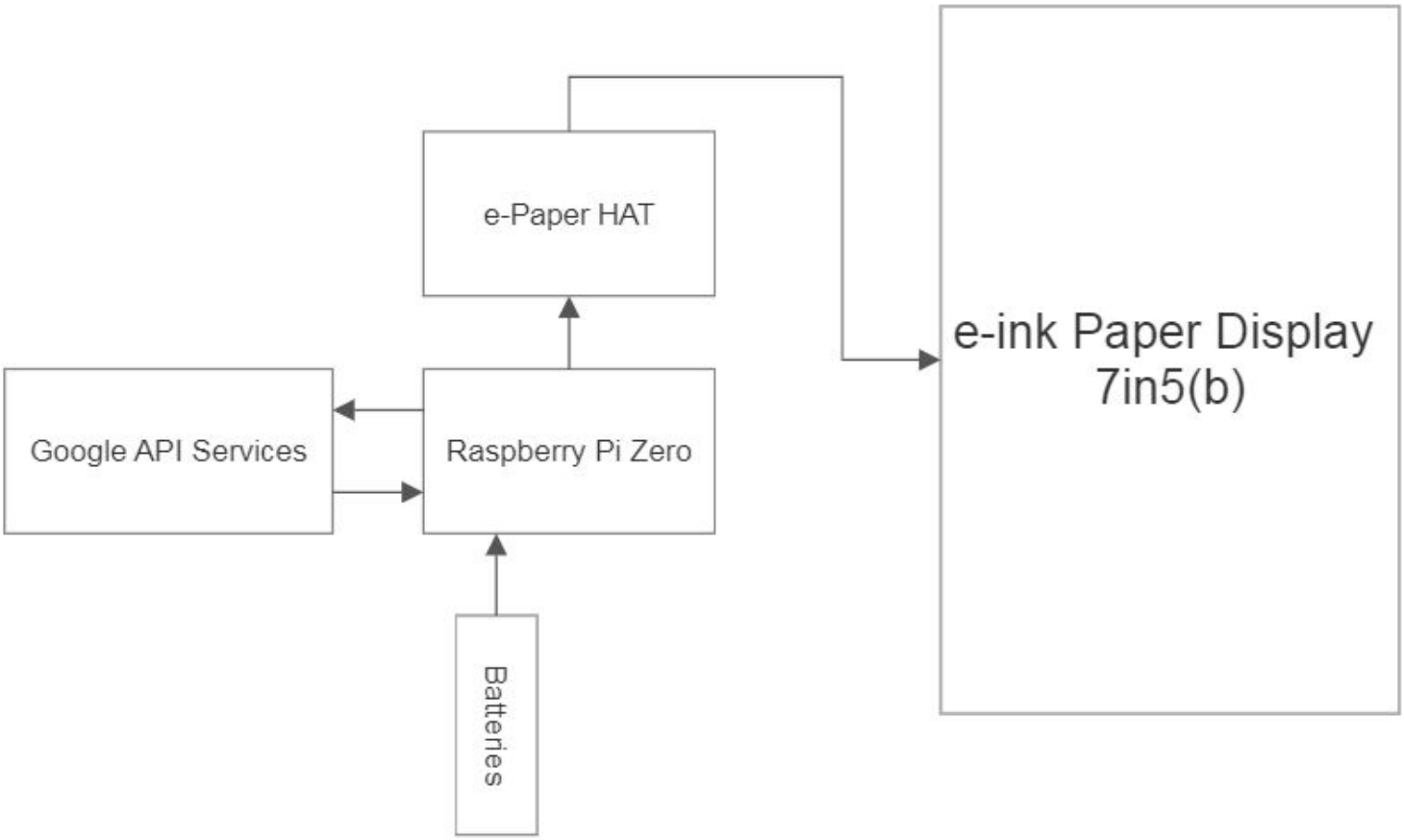
Author: Carson Pepe

E-ink Paper Display Calendar w/ Raspberry Pi Zero

The goal of this project was to create a visual display of scheduled conference room meetings from Google Calendar that can be hung up on the wall. The purpose of this was to make scheduling meetings for that conference room an easier task due to quicker access of information on the activity of that particular conference room that day. To do this, I programmed a RaspberryPi Zero with Python3 to connect to an e-Ink Paper Display, which is very low powered. This way the calendar can be running and updating itself 24/7 if needed while consuming a substantially small amount of energy. Below is a top-level block diagram of this project.

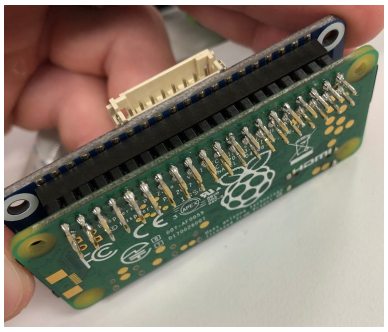


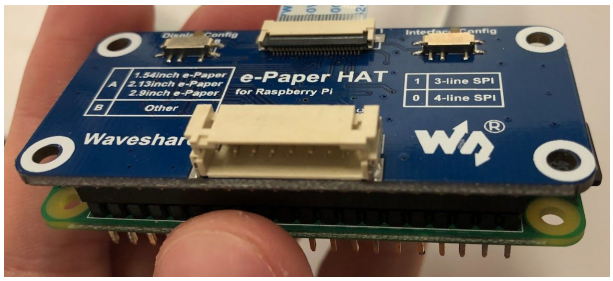
Materials:

- Waveshare 7in5(b) E-ink Paper Display
- RaspberryPi Zero
- Monitor (HDMI cable to connect)
- Keyboard
- Mouse (Optional but recommended)
- 4 Lithium-Ion batteries (3.7 Volts)
- Quad battery holder for the 4 Lithium-Ion batteries
- Voltage regulator with female micro USB plugin
- Spare male micro USB to strip & solder to output of voltage regulator

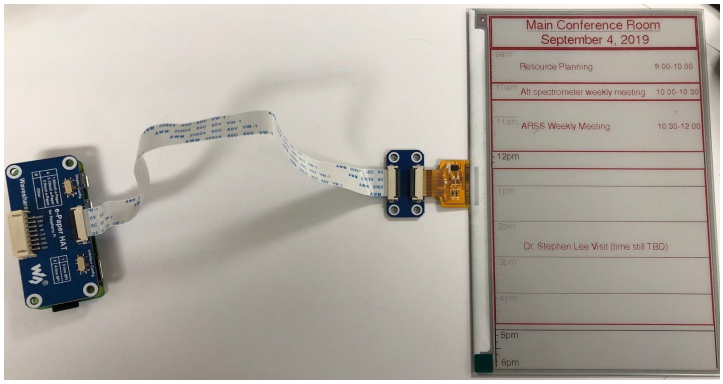
Hardware Setup:

- Solder blue e-Paper HAT chip to raspberry pi with a 2x20 header (raspberry pi logo facing out)

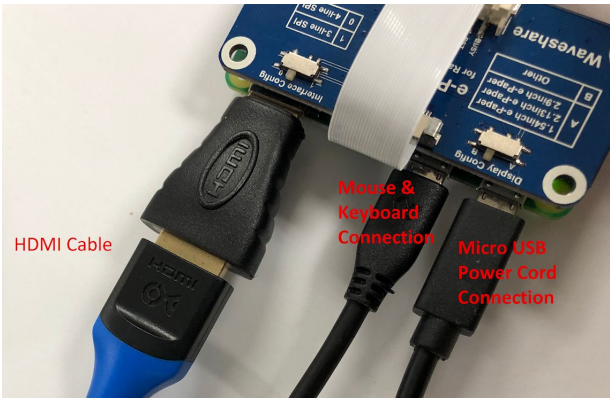




- On e-Paper HAT, ‘Display Config’ switch should be on B, ‘Interface Config’ switch on 0
- Connect RaspberryPi to E-ink Paper



- Connect RaspberryPi to monitor with HDMI cord (adapter should come with the Pi)
- Connect mouse & keyboard to Pi
- Power up the Pi with a micro usb cord (see pictures on next page)



Step 1: Setup

- Need a micro SD card with python and linux on it *See Alex Gries
- Download Raspbian onto the micro SD card (choose “Raspbian Buster with desktop and recommended software”)
- Download etcher on your laptop/desktop. You’ll use this to flash files onto the micro SD
- Flash these files onto the micro SD card (choose __pycache_ folder and any file involving ‘e-Paper Display…’). You can also get them from the nanoserver under inside the cpepe folder.
- Alternatively, you can skip the last step and add the files in through SSH once you get to step 4.

Step 2: Pi Setup

- Insert micro SD card in the RaspberryPi and power it up (make sure the HDMI cord is plugged in BEFORE powering it up, else it wont show up on screen)
- Follow the prompts on the screen selecting USA for country, American English for language, etc. (the settings are pretty obvious)
- Change the password to something easy (doesn’t matter what it is in the end, just write it down to be safe - won’t end up needing it soon)
- Configure wifi
- Say yes to updating the device, this part will take several minutes so find something to do in the meantime

Step 3: System Setup & Update/Upgrades

- Once it’s done updating, open up the terminal
- Type the command below and then press enter

```
pi@raspberryPiCalendar:~ $ sudo raspi-config
```

1. Network Options → Wi-fi → enter in the Wifi name and password
2. Boot Options → Desktop/CLI → Desktop Autologin (last option)

3. Interfacing Options → SSH → yes
   → VNC → yes
   → SPI → yes
   → Remote GPIO → yes
4. Go back to main menu and select Finish. Say yes to rebooting.

- Re-open the terminal (the following commands can take up to almost and hour long)
  - `pi@raspberryPiCalendar:~ $ sudo apt-get update`
  - `pi@raspberryPiCalendar:~ $ sudo apt-get upgrade`
  - Once both are done, `pi@raspberryPiCalendar:~ $ sudo reboot`

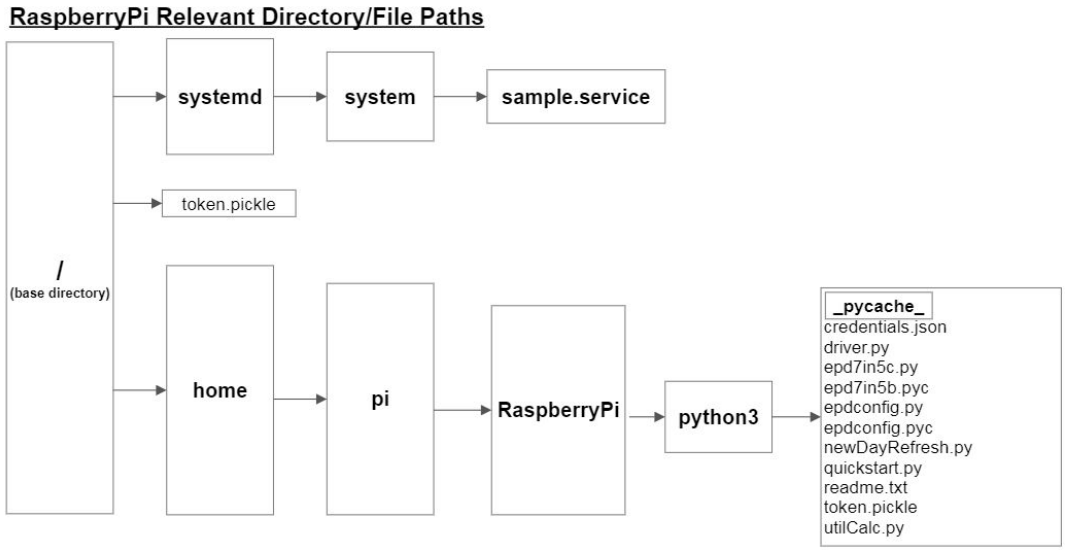
Step 4: Make sure files are in the correct folders (directories)

- ***IMPORTANT*** From here on out, unless you are editing any file using 'nano', I highly suggest you edit it through the desktop File Manager and open them with the default editor called *Thonny*. If the file is over a couple hundred lines long, this will end up being very slow/laggy, so instead ssh into your RaspberryPi from your laptop using MobaXTerm. Download the file you want to edit (left side in MobaXTerm you'll see you can do that) and edit it on your laptop. When done, save it back into the RaspberryPi through ssh.
  - To ssh into your Pi remotely you need the Pi's IP address. This IP address changes often (presumably everytime you re-login to it). To find the current IP address open up the terminal in the Pi and type 'ifconfig', then press enter. Under the 'wlan0' section your IP address is the 'xxx.xxx.xx.xx' number listed after 'inet'.
- Create a file called myscript.py `pi@raspberryPiCalendar:~ $ vi myscript.py`
- Type something random in it (so it can be saved), save & exit (esc → :x → enter), then edit it through one of the suggested methods
- Add these three lines to it then save & exit

```
#!/usr/bin/python
import os
os.system("sudo python /home/pi/RaspberryPi/python3/driver.py")
```

- Now, go into your python3 directory and check what files are in it
- `pi@raspberryPiCalendar:~ $ cd RaspberryPi/python3`
- `pi@raspberryPiCalendar:~RaspberryPi/python3 $ ls`
- Make sure it contains the following files:
  - driver.py
  - epd7in5b.py
  - epdconfig.py
  - newDayRefesh.py
  - __pycache__ (folder)
  - quickstart.py
  - utilCalc.py
- Below is a layout of relevant directories and files in the RaspberryPi. Your Pi should mimic this! You may also notice that there are two files both called 'token.pickle' in two

separate directories, as well as a file called ‘credentials.json’. Don’t worry about these files yet - we’ll get to these in Step 7. Also, ‘readme.txt’ isn’t needed.



Step 5: Fill in quickstart.py

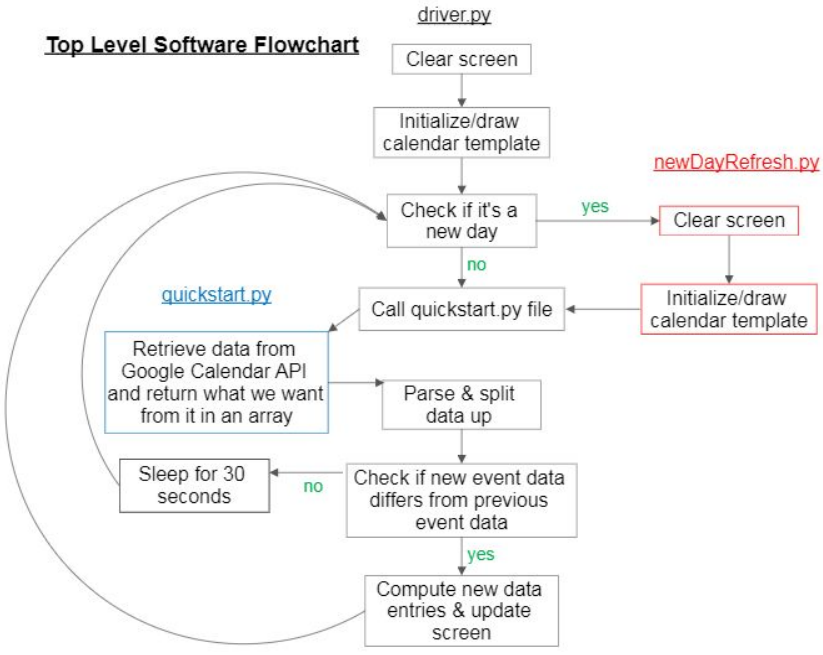
- Open up quickstart.py
- On line 11, you need to set the ‘HQ1_CONFERENCE_CAL_ID’ variable to contain the string id pertaining to which calendar you want to connect
- To find the unique string id:
    - Open up Google Calendar
    - Make sure the calendar you want to retrieve data from is added to your calendar (likely will be under “Other Calendars”)
    - On the bottom left of the screen, hover your mouse over the desired calendar and press the three vertical dots, then press settings
    - Scroll down to the bottom. Under the ‘Integrate Calendar’ section you’ll see the Calendar ID. This is the id you need to plug in as a string equal to the ‘HQ1_CONFERENCE_CAL_ID’’ variable

Step 6: Fill in driver.py

- Below the font variables on line 39, change the string passed to this function to be one pertaining to your calendar
- Save and run the script to estimate what coordinates you need to pass to line 40 in order for the underline of the text to look right (keep trying until it fits normally)
- Note: if something isn’t working properly, scroll down to the bottom of the file, comment out main() under the except statement, and add exit() to help debug and allow your program to exit on command. Otherwise the program will be stuck in an infinite loop, even when you ctl + c out of it from the command line because it takes that as an exception error

- Remember that this code is written for a 7in5(b) screen. The ‘7in5’ denotes the size of the screen and (b) denotes that it is tricolor (black, white and red).
    - If your e-Paper Display doesn’t have anything after the size denotation, this code will fail as it calls upon a variable several times to use the red color. First change all lines starting with ‘drawred’ to ‘drawblack’. Then go to the waveshare wiki, scroll down and on the right side of the page pick which version you have. Under “Demo Code” press GitHub and compare my epd7in5b.py and epdconfig.py files to the ones corresponding to your screen and make the appropriate changes. You will likely be just deleting lines off my version for the most part but make any changes where due.
    - If the screen you are working with is a different size, you’re going to need to change each pixel number to scale. To do this, individually take every x-coordinate in the code, divide it by 384 (unit pixels) and then multiply it by the height of your screen (in pixels). For the y-coordinates do the same but instead divide it by 640 and multiply it by the width of your screen. The files newDayRefresh.py and utilCalc.py both contain pixel coordinates as well, so the same action will need to be taken with those. In utilCalc.py, the ‘monthName(arg)’ and ‘time_y_coord(hour, minute)’ functions are the only two functions in the file containing pixel information (hint: they’re all y-coordinates).
        - *How can you tell which numbers in this file are ‘pixel information’ and which ones aren’t?*

          Any number assigned or compared to a variable with a ‘y’ or ‘x’ separated by an underscore from the rest of the variable name is pixel information. Any number in a line that starts out with ‘drawred’ or ‘drawblack’ is pixel information, EXCLUDING numbers set equal to a variable called ‘fill’.



Step 7: Google API Services

- Follow the steps in this link and **ignore Step 3** (use the provided quickstart.py instead)
- In Step 4 make sure you add 'sudo' to the front ("sudo python quickstart.py")
- After troubleshooting if it's not working at all, try creating an empty file called 'token.pickle' in the same directory as quickstart.py and run it again. If that doesn't work, create an empty file called 'credentials.json' and try again.
- If you're interested in how getting data from the Google API Services works, at the end of this instructable are two flowcharts outlining it. The first one shows how it uses an integrated service called OAuth 2.0 Protocol to make sure the access of data is secure. The second one shows how refreshing an expired access token is done. To understand this a little easier, try traversing the quickstart.py file while going through these flowcharts.

Step 8: Run on startup

- `pi@raspberryPiCalendar:~ $ sudo nano /lib/systemd/system/sample.service`
- Enter what's in the image below into the file but change the ExecStart line to run `/home/pi/myscript.py` instead of `/home/pi/sample.py`

```
[Unit]
Description=My Script Service
After=multi-user.target

[Service]
Type=idle
ExecStart=/usr/bin/python /home/pi/sample.py

[Install]
WantedBy=multi-user.target
```

- Save & exit this file once done editing by pressing ctl + o → enter → ctrl + x
- (In order to store the script's text output in a log file you can change ExecStart line to `ExecStart=/usr/bin/python /home/pi/myscript.py > /home/pi/sample.log 2>&1`
- Note: The systemd folder contains what initializations need to be run and in what order at bootup. Using this method creates our own service it needs to run. "After=multi-user.target" specifies the system to run this service once the RaspberryPi is ready to use in a multi-user environment. "Type=idle" means the actual execution of this service will be delayed until ALL jobs are finished. These specifications are important because we need ours to run when our Pi is completely ready so that it can run normally, just as when we manually run our script through the command line.
- Note: `sudo` in all of the following commands are needed for it to work
- Next, the permission on the unit file needs to be set to 644 :
  `sudo chmod 644 /lib/systemd/system/sample.service`

- Now run `sudo systemctl daemon-reload`
  then `sudo systemctl enable sample.service`

- Now, go to the base directory on the Pi. To make sure you are, keep typing “cd ..” into the command line and pressing enter. After a few times, type ‘pwd’ and press enter. If it returns “/’ you’re there. This is where the Pi on startup will run your script from.
- Because it’s running your script from here, you need to have the files ‘token.pickle’ and also (initially) ‘credentials.json’ in this directory

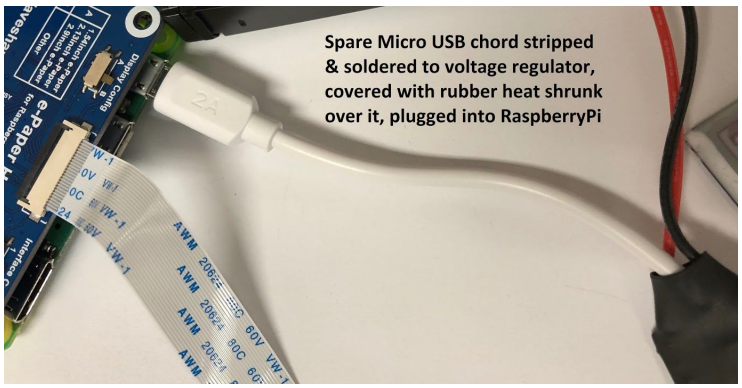
  ```
  sudo cp /home/pi/RaspberryPi/python3/token.pickle token.pickle
  sudo cp /home/pi/RaspberryPi/python3/credentials.json credentials.json
  ```
- You can test it out by using these commands:

  `$ sudo systemctl start sample.service`
  - Starts your service
  - Use this as a simulation to make sure your program is being ran correctly

  `$ sudo systemctl stop sample.service`
  - Stops your service
  - Use this to stop the current service when something isn’t going right
  - Not necessary if your service stops by itself

  `$ systemctl status sample.service` (no need to put sudo)
  - Checks the status of your service
  - Use this after you start your service to check if it is running or why it stopped running
- Debugging: Any print statements or errors in the code won’t show up on the command line when running it from systemd. If there are problems, go into the files being used and write to a file at different ‘checkpoints’ in the code to see where it is being stopped.
- Once it’s running correctly (takes a minute to a minute and a half for it to actually start displaying the calendar) reboot the Pi `sudo reboot` and it should run (give it a minute or two)

Step 9: Battery Powered
*See Dan Mitchell



Improvements/Recommendations
There are still several improvements that can be made to this project and its code. Here are a few things I suggest be changed and/or fixed. I also mentioned what I've already tried, why I tried it, and what I learned about the e-Paper hardware & software from these attempts. I figure if you want to tackle these issues it may be helpful to know what I learned from it so you don't find yourself retracing my steps entirely and ending up in the same rabbit holes I did. However, I advise you to not look at these attempts as a 'rule out' of possible solutions. You never know, I may have already taken the right route but had one small thing missing or incorrect. Note: Some sections of this reference the code in some of my files, making it difficult to follow along without the code in front of you. You'll get the most out of this if you have the code up in an editor while reading this.

a) **Problem 1(biggest one by far):** Cancelled events that appear as crossed out in the Google Calendar still somehow make it through to the data set of events we receive from the Google Calendar API. This not only portrays inaccurate information but also can

cause for apparent meeting overlaps on the e-Paper Display, which looks very confusing and doesn't make any sense when looking at it. **This problem matters the most and should be fixed before any other problem, as the failure to filter out cancelled events often compromises the general purpose of the e-Paper Calendar.**

What to know:

- Within the quickstart.py file, line 50 {events_result = service.events().list(calendarId=...)} is where we actually are able to grab the 'protected resource', or data we want from the Google Calendar API. What's in the parentheses after .list contains the parameters of what exact data we want as well as what to filter out and what order to give it to us in. For more information on what the parameters each can control as well as other possible ones to try to include, click here. Also note that the events_result variable gets returned a hash of hashes, or in Python terms, a dictionary of dictionaries.
- You must call "epd.display(epd.getbuffer(HBlackimage), epd.getbuffer(HRedimage))" to actually start writing to the screen. Everything before that is simply adding to the picture what you want to be in it. It's also important to note that when re-writing to the screen due to an update of events, if you want an already existing line to disappear you have to draw another line over that same spot in white (same color as background). I personally don't like this method if the screen has a drastic change in it (for instance, when it changes to a new day) because overtime it would cause the variables holding the data for your picture to get overcrowded with writes, virtual erases, re-writes etc.. You can get around this by clearing the screen anytime an update that requires for something on the previous picture to go away, re-initializing the epd variable (which is basically your studio), the HBlackimage & HRedimage variables, and the drawblack & drawred variables (which are basically your portrait). This way not only is the screen cleared but the set of data you will write to the screen is a clean slate.

Attempted Solution(s):

1. I first tried to take care of this problem internally, or within this line (50) of the code. After studying the documentation on .list() and the possible parameters that can be passed to it, I added the 'showDeleted' variable into the code and set it equal to False. The documentation does say that this variable is False by default, but I tried it anyways and it still allowed cancelled events to come through.
2. I also tried to take care of this problem externally, orafter of this line of the code was run. I took the events_result variable after it was made and wrote out every bit of data it contained in each internal dictionary to a .txt file. In doing this, I tried to find a variable within each event that corresponded to the status of an event in terms of cancellation. According to the link above, there *is* one called 'status' and it will in fact be equalled to 'cancelled' if this is the case. However, I wasn't getting this result when I was writing out to the file. Furthermore, I tried finding any other incidental similarities between non-cancelled events that a cancelled event did

not have (because cancelled event data was much more rare to retrieve than non-cancelled events), but I was unable to find any consistent trend.

3. **If you are successful in filtering out cancelled events**, it is likely that you may need to do a screen clear and rewrite to the screen, as described in the last bullet point of the 'What to know' section. For an example of how to get this started, see lines 180 & 181 as well as the file newDayRefresh.py. I do this when a new day is detected to start each day on a clean slate. I do this because to my understanding, it is not good to overcrowd the board with data with writing and then writing over the same points.

b) **Problem 2:** The refresh rate of the screen is relatively slow to what I'm aware its hardware is capable of. The screen clear which I call at both startup and refreshing the screen for a new day is also slow, but the refresh rate holds higher importance since screen refreshes are more frequent.

What to know:

- It might be worth having an understanding of how the e-Paper Display is able to show different colors on the screen for this issue. If you want to have this knowledge before diving into this problem, watch 4:14 through 8:36 of this video. Essentially, each hexagonal pixel receives some sort of 'clever wave function' that either defines or provides a negative or positive voltage across the anode/cathode of that pixel. The video also explains how the only way a pixel can turn red is by turning that pixel black first, then following it with a 'clever' series of voltage fluctuations over a certain amount of time frames to that pixel so that the red particles eventually make it to the transparent surface of that pixel. I found this information in particular important to know because in order to change the frequency at which the screen refreshes, you need to somehow change that series of time frames; likely decrease it.
- When in driver.py 'epd.display' is called, the Pi will finally start to compile the picture that the code prior to it created for the screen to display. Everything coming before it either parses & filters data, is computational, or is creating different lines and/or text to add to the next picture. The parameters passed to this function call a separate function 'getbuffer' which is ran twice (once for each call) as its return value of the buffered list is needed to be passed to the display function.

Attempted Solution(s):

1. I first started looking into the epd7in5b.py and epdconfig.py files and studied them (I got these two files from the waveshare wiki). I found that epdconfig.py mainly contains functions for epd7in5b.py to call. Focusing on the getbuffer(self, image) function inside epd7in5b.py, I had it write to one file the values of pixels[x,y] and to another file the computation of the next line: buf[(newx + newy*self.width) // 8] &= ~(0x80 >> (y % 8)). After further studying the documentations on several of the imported methods called within this function, my attempt to find any patterns between the output of each file was still inconclusive. It seemed to me that if pixels[x,y] was equal to 0, it indicated that

that pixel needed to be updated/changed. I ended up having this function write the final return value 'buf' which is a list of integer values in hex (length 30,720) to a file so I would at least have that to go off of. I figured that if I wanted to be able to speed up the refresh rate, I'd need to at least have a deep understanding of this function and what it was creating.

2. My next step was to study the 'display(self, imageblack, imagered)' function. It appeared that this is where the 'clever wave functions' were created from an algorithm using low level logic (and's [&], or's [|], bitshifts [<<]). I figured this because for each iteration it sent data to the board eight times. This would mean that the wave function would have eight 'peaks' of different amplitudes. I couldn't figure out what exactly this algorithm was doing or even how to reverse engineer it. What I did find even more confusing from this algorithm after writing its results to another file was that the wave function it would create for a pixel that was supposed to be black could be entirely different from the wave function it would create for a different pixel that was ALSO supposed to be black (same thing would happen for white and red pixels, this wasn't exclusive to the color black). These findings led me to realize that the refresh rate may also be marginally affected by the amount of time it took for these functions to finish sending data to the screen.
3. I decided I could finally start to tamper with the functions since I had an idea of where making changes in them could have an effect. I started off timing each function (including the screen Clear(self, color) function) and I measured:

   ~1.9-2.1 seconds for epd.getbuffer(self, image) to finish
   ~25.9-26.1 seconds for epd.display(self, imageblack, imagered) to finish
   ~51.8-52.0 seconds for epd.Clear(0xFF) to finish

   Refresh Analysis*:* In conclusion, epd.display is what was taking up most of the time during refresh. However, the only line I could modify to speed this up was sleep(100) line. I changed this to 50 and it completely screwed up the data being sent to the e-Paper. Therefore I ruled this to be untouchable.

   Clear Screen Analysis**:** Further investigating this time-taking function, I timed separate parts of it and found that the four 'self.send_data(0x33)' lines all take about 5.0 seconds each to complete and all correspond to one fourth of the screen so none of them could be taken out. The only other line of code that took any relatively large amount of time was 'self.wait_until_idle()' which took about 31.85 seconds. This function call simply reads the 'busy_pin' value off the e-Paper Display board and if it has a value of zero, it waits another 100 milliseconds before reading the value again. It stops once busy_pin is a nonzero value. Therefore I ruled this function to also be untouchable.

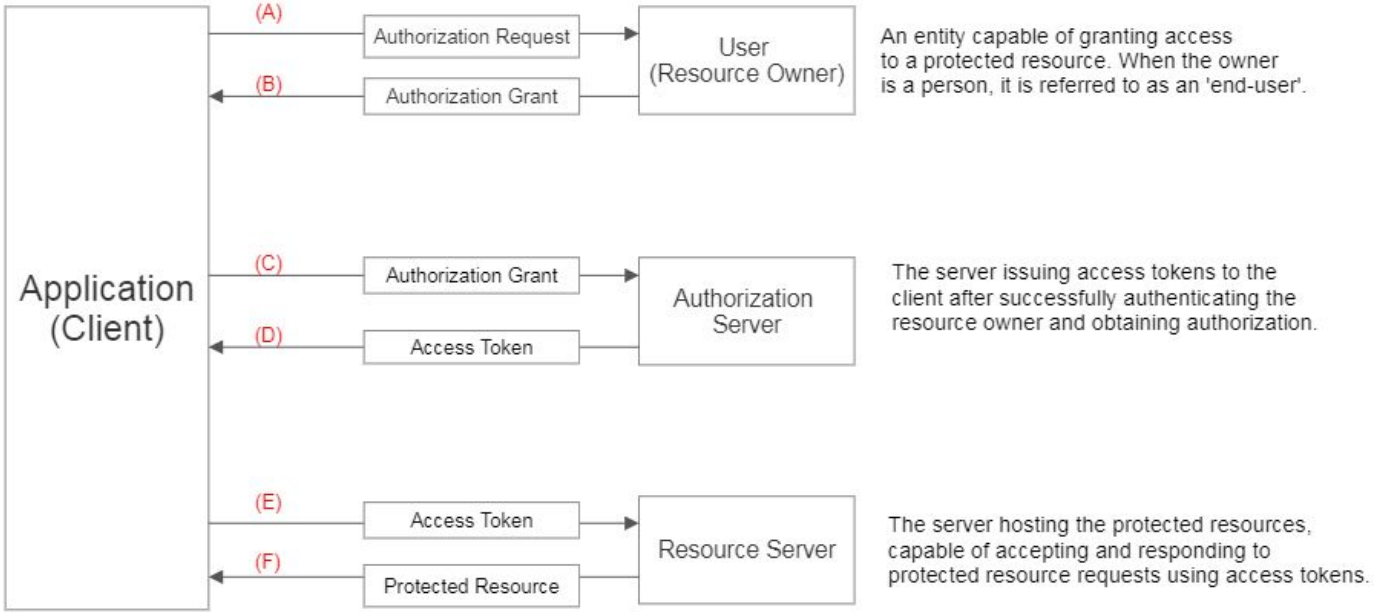
c) **Improvement:** Make the code more universal so that it's easier to modify for implementation on a different model of an e-Ink Paper Display. (i.e. find a way to make all pixel's x and y-axis numbers be computed by some general algorithm that is based off the ratio of itself divided by the maximum length of its corresponding axis). EPD_WIDTH and EPD_HEIGHT are variables any user could input their e-Paper's dimensions in. It

would be best if all coordinates relied on these two variables. The way I would first think to go about this is creating a helper function in a separate file that you would pass every number that is a coordinate to. This function would contain the information on what the width and height are of the e-Paper as well as the ratios for each number I use in my code that represents a location on the e-Paper and keep it all in some sort of dictionary. From there it should be a simple solve-for-x type of algorithm.

d) **Improvement:** Increase the battery life (current life is ~8 days) by putting structure into the code that only allows it to update during a specific time frame during the day, and sleep when it's likely that no one will be at the building to see the calendar.

Google Calendar API

OAuth 2.0 Protocol Flow



(A) Client requests authorization from resource owner.

(B) Client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of the four grant types** (or an extension grant type). The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

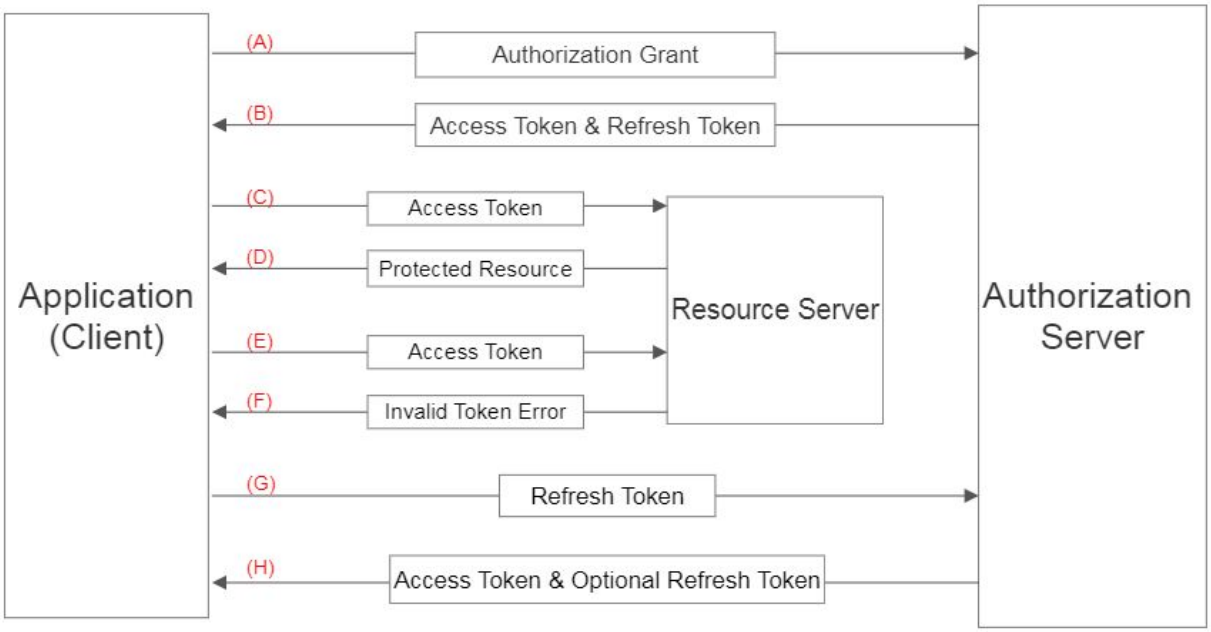
(C) Client requests an access token by authenticating with the authorization server and presenting the authorization grant.

(D) The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.

(E) Client requests the protected resource by presenting an access token.

(F) Resource server validates the given access token, if valid, it provides the client with the protected resource (the desired data).

Refreshing an Expired Access Token



(A) Client requests access token with authorization grant.

(B) If the authorization grant is confirmed valid, the authorization server issues an access token and refresh token.

(C) The client makes a protected resource request to the resource server by presenting the access token.

(D) If the access token is validated, it serves the request for the protected resource (data).

(E) Steps (C) and (D) repeat until the access token expires (typically takes about an hour.

(F) Once the client get an invalid token error, it skips out of this loop into step (G).

(G) The client requests a new access token by presenting the refresh token associated with the last access token, and authenticating it with the authorization server.

(H) The authorization server issues a new access token and refresh token (optionally, but for this project we will always need one).