PyGamer Thermal Camera with AMG8833

Created by Jan Goolsbey

Last updated on 2020-01-29 10:01:29 PM UTC

Overview

This pocket-sized, portable thermal camera project combines an AMG8833 IR Thermal Camera FeatherWing with a PyGamer to provide a full-featured thermal imaging camera. CircuitPython will be in charge of reading and displaying the thermal image as well as interacting with operator controls.

A thermal camera can be very useful for finding home heating leaks, looking for electrical circuit hot spots, troubleshooting printed circuit board components, and for knowing when your tea is just right for sipping. The initial reason I built one was to watch the rate of heat buildup along a length of clothes dryer exhaust duct.

The camera displays a thermal image or histogram and sports a shutter button to freeze the image. The focus feature fine-tunes the display's temperature range to match the current images maximum and maximum measurements. A settable alarm flashes lights and beeps when the camera sees a temperature at or above the threshold. The setup function is used to set the temperature display range and the alarm threshold. An editable configuration file contains the camera's power-up settings.

The heart of the camera is a thermal imaging sensor with an 8 by 8 thermopile array that reads temperatures from 32°F to 176°F (0°C to 80°C) with an absolute accuracy of +- 4.5°F (2.5°C) and resolution of 0.9°F (0.5°C). The 64 elements in the array are too few to see a lot of detail, but it is possible to recognize general thermal zones and shapes.

This version of the camera displays numeric temperature values as degrees Fahrenheit. Converting the displayed values to Celsius is possible but is left as an exercise.

CAUTION: The AMG8833 sensor used in this project is not accurate or stable enough to be used for health or safety purposes.

Parts

Your browser does not support the video tag.

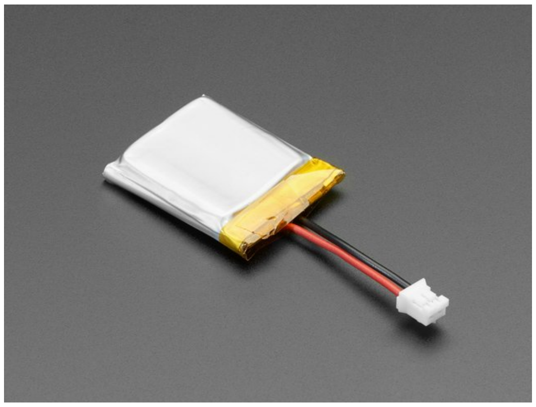
Adafruit AMG8833 IR Thermal Camera FeatherWing

$39.95
IN STOCK
Add To Cart

Adafruit PyGamer for MakeCode Arcade, CircuitPython or Arduino

$39.95
IN STOCK

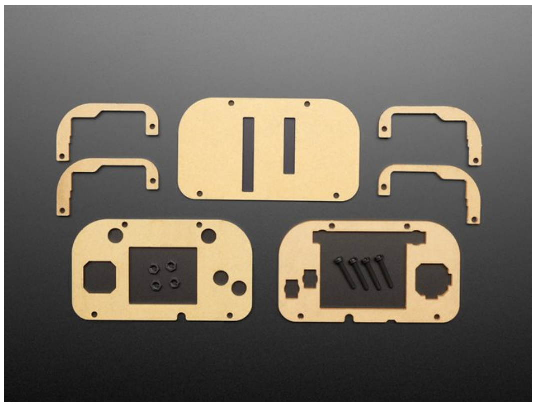
Add To Cart



Lithium Ion Polymer Battery with Short Cable - 3.7V 350mAh

$5.95
IN STOCK

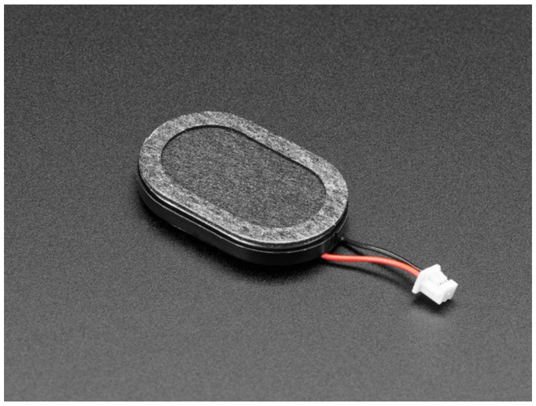
Add To Cart



Adafruit PyGamer Acrylic Enclosure Kit

$12.50
IN STOCK

Add To Cart



Mini Oval Speaker with Short Wires - 8 Ohm 1 Watt

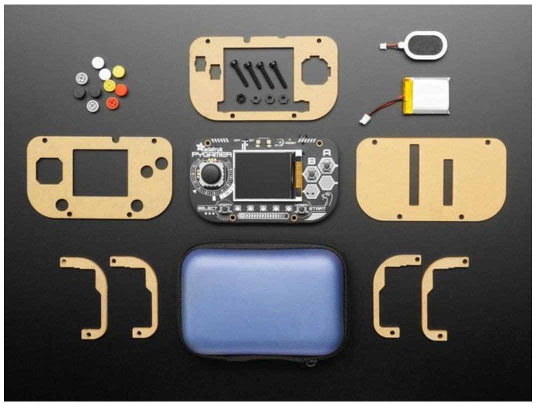
OUT OF STOCK

Out Of Stock

Plastic Button Caps For Square Top (10-pack) - 8mm Diameter

$0.95
IN STOCK
Add To Cart

The kit below contains the products used here also.



Adafruit PyGamer Starter Kit

OUT OF STOCK
Out Of Stock

Acknowledgements

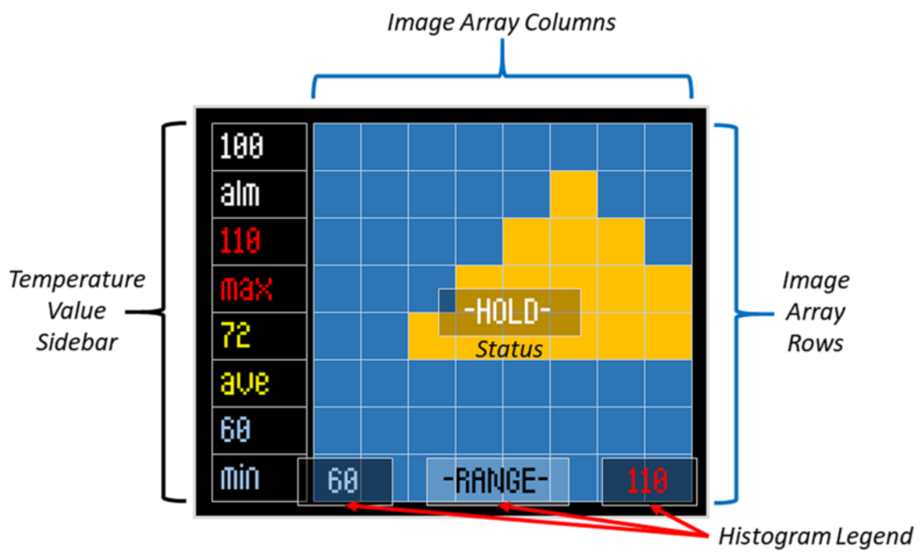
Thank you to the CircuitPython community for unfailing support and encouragement. Thanks to Anne Barela for insightful advice and help with the overall guide content/readability.

Special thanks to Melissa LeBlanc-Williams for her expert Displayio tutorial -- it was the catalyst for my initial breakthrough in understanding and applying Displayio. Her techniques also shaped some of the display principles used in this learning guide.

Features and Operation

The Thermal Camera's controls are used to switch display modes, take snapshots, automatically increase or decrease image temperature gradient detail, and facilitate setting alarm and maximum/minimum display range parameters. The display shows the image or histogram and the currently measured maximum, minimum, and average temperature values in Fahrenheit.

Display Layout



The camera's display is divided into four zones. The temperature value sidebar is used to display the alarm (**alm**) threshold setting, the measured maximum temperature (**max**), the average temperature calculation (**ave**), and the measured minimum temperature (**min**). The sidebar continuously displays measured values during normal operation. When in the **Setup** mode, the sidebar indicates the current alarm threshold, the maximum display range, and the minimum display range.

The image array area consists of 64 blocks in an 8 column by 8 row array. The image array is used to display a thermal sensor image or histogram.

Superimposed over the display grid are the status message area (centered in the image array area) and the histogram legend area (near the bottom of the image array area). The status message area indicates various operational states including **Hold**, **Focus**, and the **Setup** mode. The histogram legend area shows the current minimum and maximum display range settings when viewing a histogram

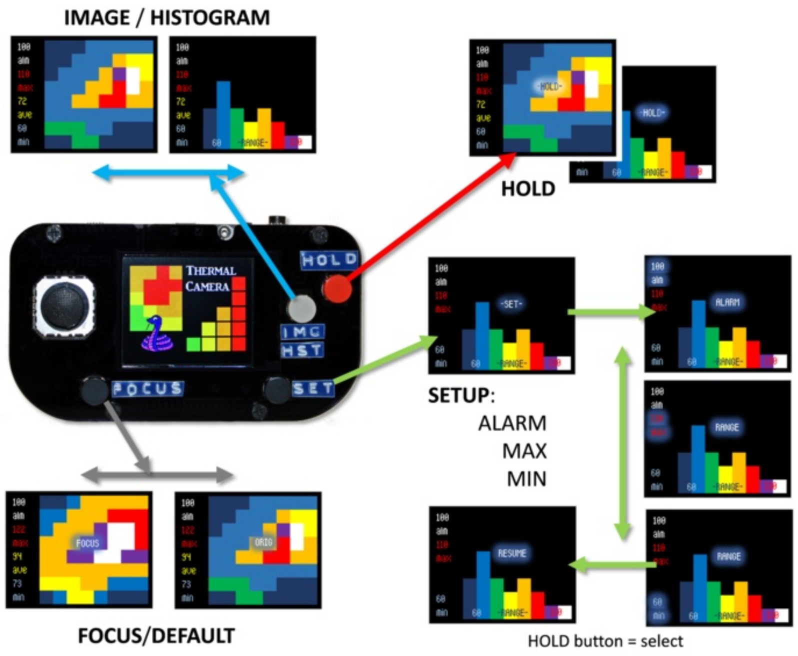


Image / Histogram Mode

The **IMG-HST** button (PyGamer button B) is used to toggle between a temperature gradient image and a temperature distribution representation of the thermopile sensor's measurements. The **IMG-HST** button is operational when in **Hold** mode to allow analysis of held measurements.

Hold Mode

The **HOLD** button (PyGamer button A) freezes and releases the image or histogram display contents. Press the button once to hold the display; press it again to resume normal operation. The **IMG-HST** button continues to operate normally regardless of whether or not the display is held.

Focus Range / Default Range

The **FOCUS** button (PyGamer Select button) automatically changes the minimum and maximum display range values to provide increased or decreased detail based on the currently measured maximum and minimum temperatures. Press **FOCUS** once to change the current display range from the current setting to a range that matches the measured minimum and maximum values. Press it again to return to the original display range settings. **Focus** mode is useful when looking for increased temperature gradient detail or when the temperature of the object is outside of the default or re-set display range.

Setup Function

Pressing the **SET** button (PyGamer Start button) will stop normal operation and enter the**Setup** mode where the alarm threshold and maximum/minimum display range are adjustable. Use the joystick or the PyBadge D-Pad buttons to highlight the parameter to change, then press the **HOLD** button to select. Use the joystick to increase or decrease the parameter value. Press the **HOLD** button to select the new value. To exit the **Setup** mode, press the **SET** button.

The newly selected values will go into effect when exiting **Setup** mode, but will not be preserved if the camera's power is turned off. To change power-on parameter values, edit the **thermal_cam_config.py** file with Mu or your favorite text editor.

Build the Camera

It's time to get the PyGamer ready by installing CircuitPython and its libraries, plug in the speaker and battery, and put it into an elegant enclosure. Once the enclosure is in place, we'll attach the AMG8833 FeatherWing and load the Thermal Camera code.

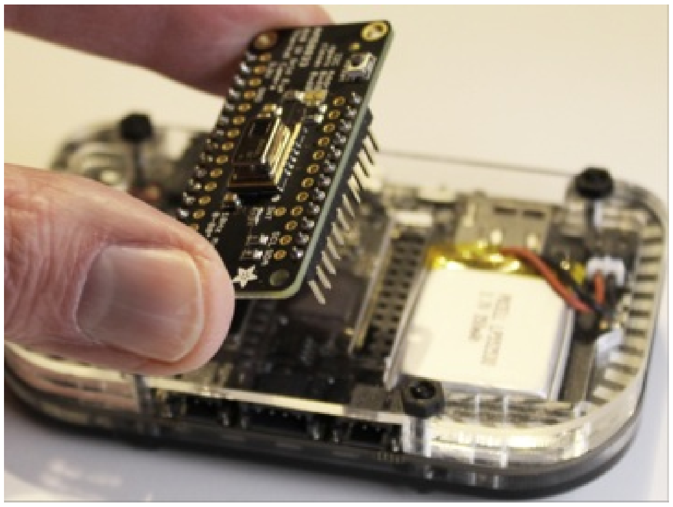
You can build the Thermal Camera from individual components or from the PyGamer Starter Kit (https://adafru.it/IAh). Add the AMG8833 FeatherWing (https://adafru.it/IAi) and you'll be ready to go.

Assembling the PyGamer

The PyGamer Introduction (https://adafru.it/pygamer) will guide you through the process of setting up the PyGamer to include the case, battery, and speaker.

You may follow the Starter Kit enclosure instructions (https://adafru.it/IAj) for installing the speaker and battery, even if you don't plan to use the enclosure.

Prepare the FeatherWing



Solder the included male headers onto the AMG8833 FeatherWing and attach it through the acrylic back panel into the PyGamer's Feather connector. Refer to the soldering guide (https://adafru.it/dxy) if this is your first time with a soldering iron.

Preparing the PyGamer with CircuitPython, Libraries, and Accessories

The PyGamer Introduction (https://adafru.it/pygamer) guide also has the information needed to install CircuitPython (https://adafru.it/FoA) and its libraries (https://adafru.it/IAk).

Software Setup

This project uses CircuitPython, a user friendly version of Python for microcontrollers. The files are just text files and are copied over to the PyGamer to the flash drive **CIRCUITPY** which appears when the PyGamer is attached to a computer via a USB cable.

Preparing the PyGamer with CircuitPython and Software Libraries

The PyGamer Introduction (https://adafru.it/pygamer) guide also has the information needed to install CircuitPython (https://adafru.it/FoA) and its libraries (https://adafru.it/IAk).

See the following pages on how to perform these operations.

CircuitPython

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) 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** flash drive to iterate.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

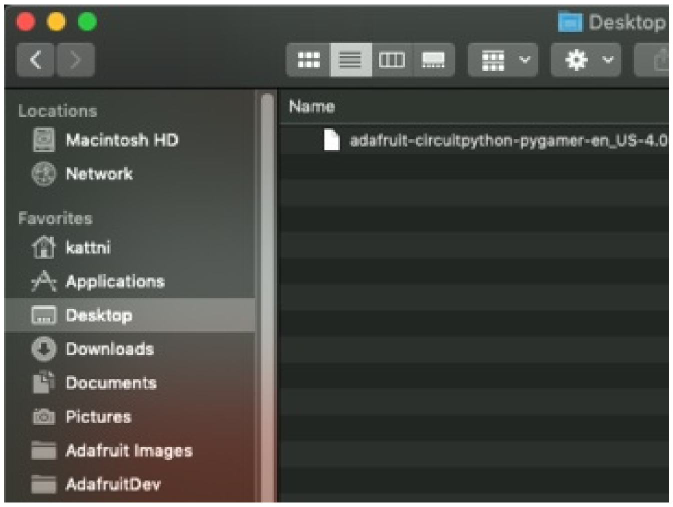
Follow this quick step-by-step for super-fast Python power :)

https://adafru.it/FxM

https://adafru.it/FxM

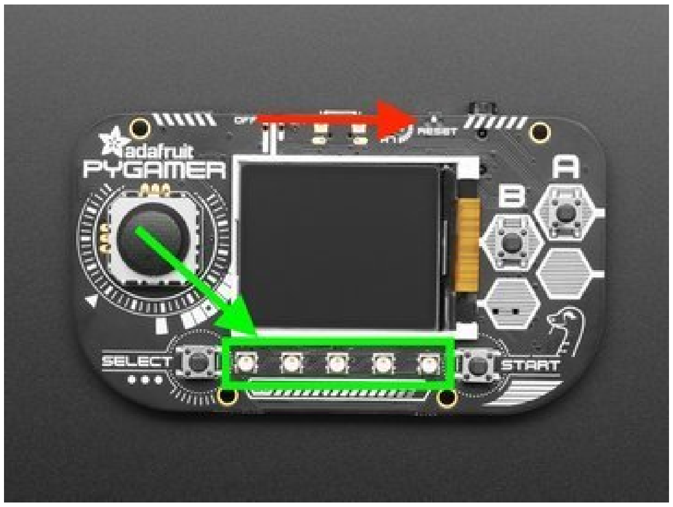
Further Information

For more detailed info on installing CircuitPython, check out Installing CircuitPython (https://adafru.it/Amd).



Click the link above and download the latest UF2 file.

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

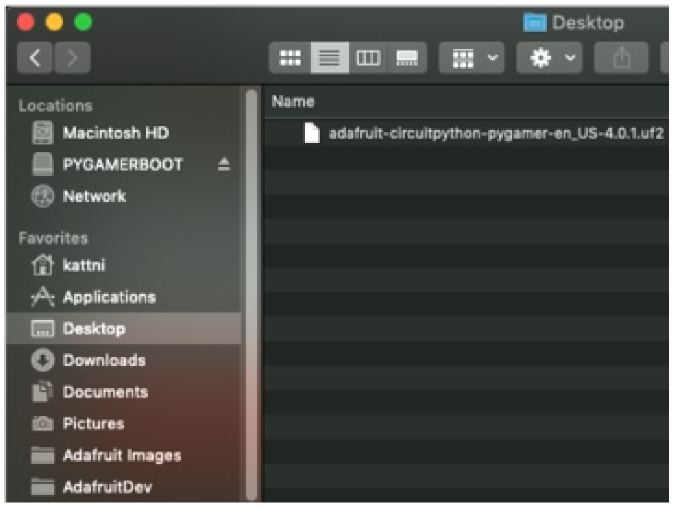


Plug your PyGamer into your computer using a known-good USB cable.

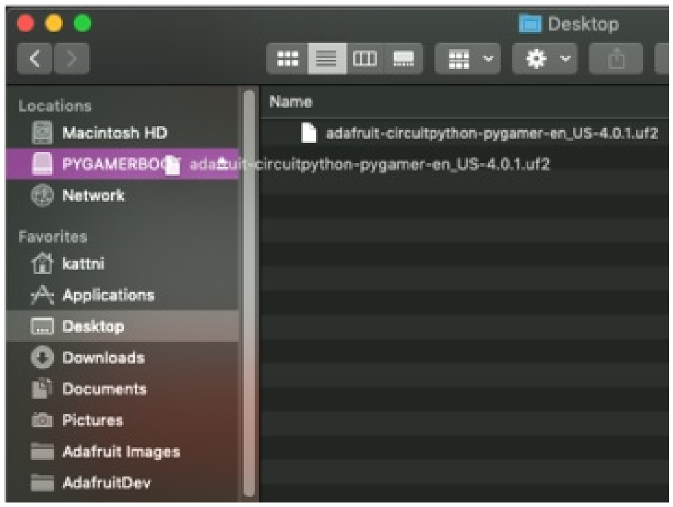
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the **Reset button on the top** of your board (indicated by the red arrow in the first image). You will see an image on the display instructing you to drag a UF2 file to your board, and **the row of NeoPixel RGB LEDs on the front will turn green** (indicated by the green arrow and square in the image). If they turn 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!



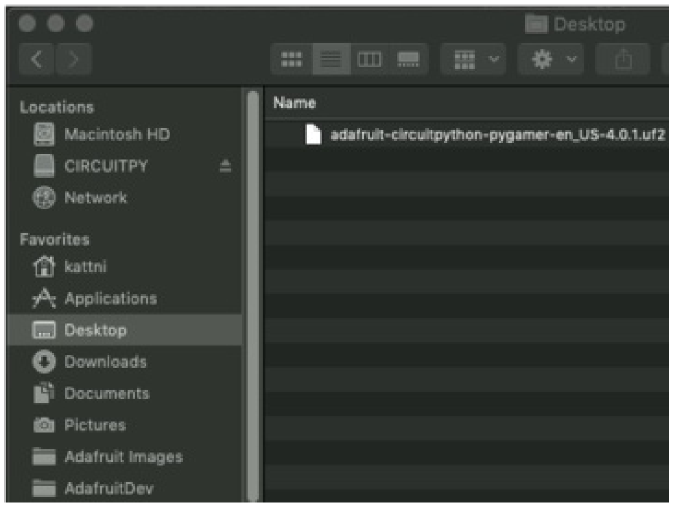
adafruit-circuitpython-pygamer-en_US-4.0.1.uf2



adafruit-circuitpython-pygamer-en_US-4.0.1.uf2

You will see a new disk drive appear called **PYGAMERBOOT**.

Drag the **adafruit_circuitpython_etc.uf2** file to **PYGAMERBOOT.**



The LEDs will flash. Then, the **PYGAMERBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

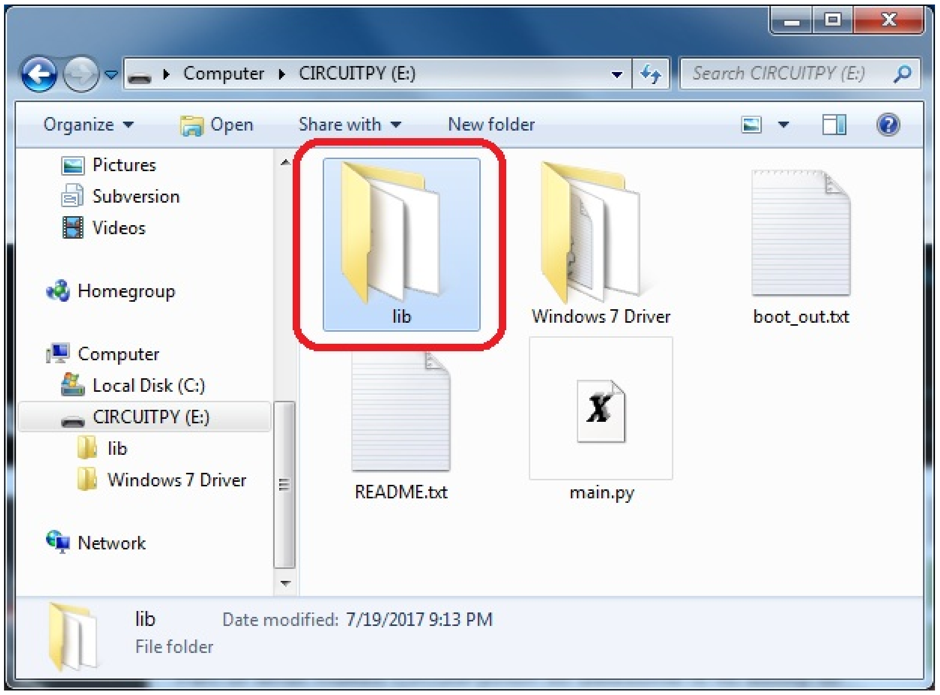
That's it, you're done! :)

CircuitPython Libraries

As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called *libraries*. Some of them are built into CircuitPython. Others are stored on your **CIRCUITPY** drive in a folder called **lib**. Part of what makes CircuitPython so awesome is its ability to store code separately from the firmware itself. Storing code separately from the firmware makes it easier to update both the code you write and the libraries you depend.

Your board may ship with a **lib** folder already, it's in the base directory of the drive. If not, simply create the folder yourself. When you first install CircuitPython, an empty **lib** directory will be created for you.



CircuitPython libraries work in the same way as regular Python modules so the Python docs (https://adafru.it/rar) are a great reference for how it all should work. In Python terms, we can place our library files in the **lib** directory because its part of the Python path by default.

One downside of this approach of separate libraries is that they are not built in. To use them, one needs to copy them to the **CIRCUITPY** drive before they can be used. Fortunately, we provide a bundle full of our libraries.

Our bundle and releases also feature optimized versions of the libraries with the **.mpy** file extension. These files take less space on the drive and have a smaller memory footprint as they are loaded.

Installing the CircuitPython Library Bundle

We're constantly updating and improving our libraries, so we don't (at this time) ship our CircuitPython boards with the full library bundle. Instead, you can find example code in the guides for your board that depends on external libraries. Some of these libraries may be available from us at Adafruit, some may be written by community members!

Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

You can grab the latest Adafruit CircuitPython Bundle release by clicking the button below.

Note: Match up the bundle version with the version of CircuitPython you are running - 3.x library for running any version of CircuitPython 3, 4.x for running any version of CircuitPython 4, etc. If you mix libraries with major CircuitPython versions, you will most likely get errors due to changes in library interfaces possible during major version changes.

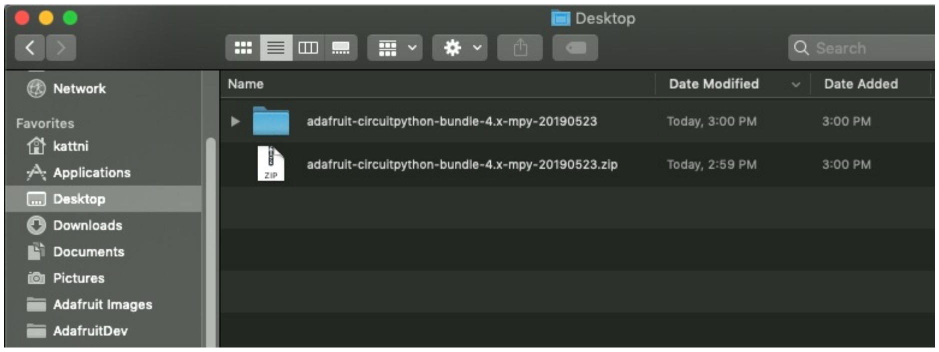
https://adafru.it/ENC

https://adafru.it/ENC

If you need another version, you can also visit the bundle release page (https://adafru.it/Ayy) which will let you select exactly what version you're looking for, as well as information about changes.

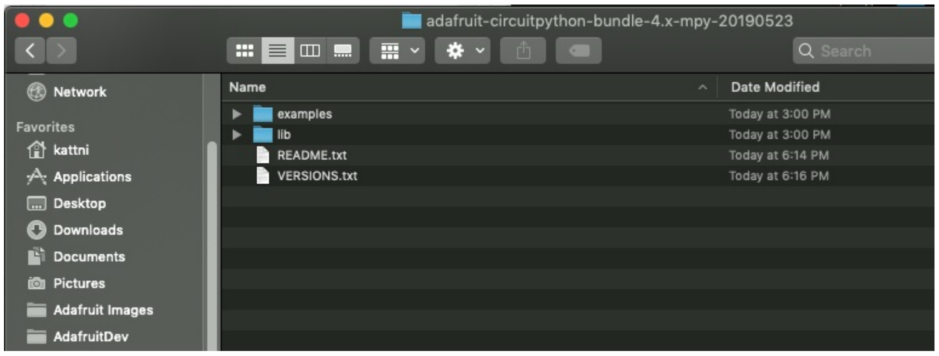
Either way, download the version that matches your CircuitPython firmware version. If you don't know the version, look at the initial prompt in the CircuitPython REPL, which reports the version. For example, if you're running v4.0.1, download the 4.x library bundle. There's also a **py** bundle which contains the uncompressed python files, you probably *don't* want that unless you are doing advanced work on libraries.

After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



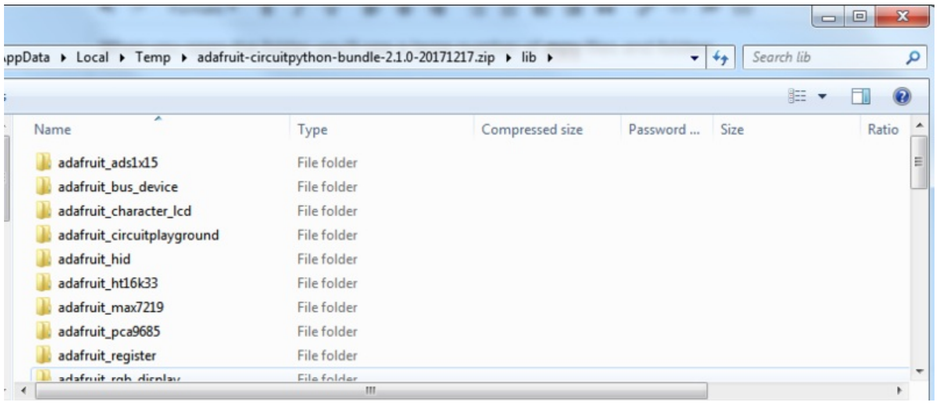
Name	Date Modified	Date Added
adafruit-circuitpython-bundle-4.x-mpy-20190523	Today, 3:00 PM	3:00 PM
adafruit-circuitpython-bundle-4.x-mpy-20190523.zip	Today, 2:59 PM	3:00 PM

Open the bundle folder. Inside you'll find two information files, and two folders. One folder is the lib bundle, and the other folder is the examples bundle.



Name	Date Modified
examples	Today at 3:00 PM
lib	Today at 3:00 PM
README.txt	Today at 6:14 PM
VERSIONS.txt	Today at 6:16 PM

Now open the lib folder. When you open the folder, you'll see a large number of**mpy** files and folders

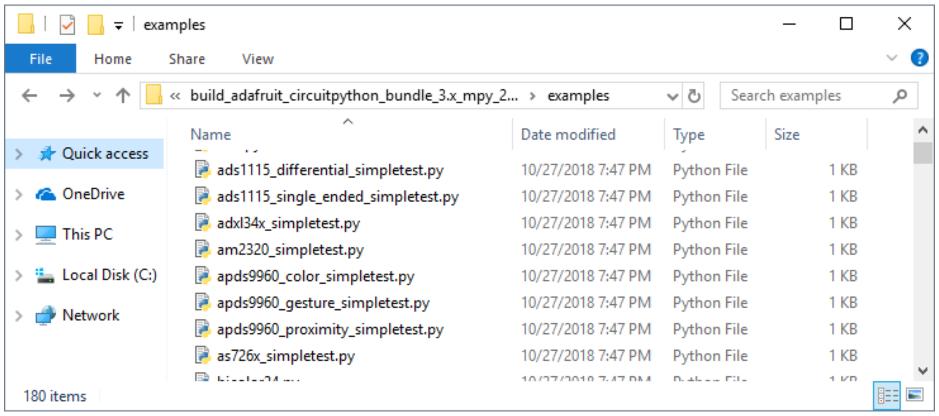

ppData ▸ Local ▸ Temp ▸ adafruit-circuitpython-bundle-2.1.0-20171217.zip ▸ lib ▸

Name	Type	Compressed size	Password ...	Size	Ratio
adafruit_ads1x15	File folder				
adafruit_bus_device	File folder				
adafruit_character_lcd	File folder				
adafruit_circuitplayground	File folder				
adafruit_hid	File folder				
adafruit_ht16k33	File folder				
adafruit_max7219	File folder				
adafruit_pca9685	File folder				
adafruit_register	File folder				

Example Files

All example files from each library are now included in the bundles, as well as an examples-only bundle. These are included for two main reasons:

- Allow for quick testing of devices.
- Provide an example base of code, that is easily built upon for individualized purposes.


build_adafruit_circuitpython_bundle_3.x_mpy_2... › examples

Name	Date modified	Type	Size
ads1115_differential_simpletest.py	10/27/2018 7:47 PM	Python File	1 KB
ads1115_single_ended_simpletest.py	10/27/2018 7:47 PM	Python File	1 KB
adxl34x_simpletest.py	10/27/2018 7:47 PM	Python File	1 KB
am2320_simpletest.py	10/27/2018 7:47 PM	Python File	1 KB
apds9960_color_simpletest.py	10/27/2018 7:47 PM	Python File	1 KB
apds9960_gesture_simpletest.py	10/27/2018 7:47 PM	Python File	1 KB
apds9960_proximity_simpletest.py	10/27/2018 7:47 PM	Python File	1 KB
as726x_simpletest.py	10/27/2018 7:47 PM	Python File	1 KB

180 items

Copying Libraries to Your Board

First you'll want to create a **lib** folder on your **CIRCUITPY** drive. Open the drive, right click, choose the option to create a new folder, and call it **lib**. Then, open the **lib** folder you extracted from the downloaded zip. Inside you'll find a number of folders and **.mpy** files. Find the library you'd like to use, and copy it to the lib folder on**CIRCUITPY**.

This also applies to example files. They are only supplied as raw **.py** files, so they may need to be converted to **.mpy** using the **mpy-cross** utility if you encounter `MemoryErrors`. This is discussed in the CircuitPython Essentials Guide (https://adafru.it/CTw). Usage is the same as described above in the Express Boards section. Note: If you do not place examples in a separate folder, you would remove the examples from the `import` statement.

Example: `ImportError` Due to Missing Library

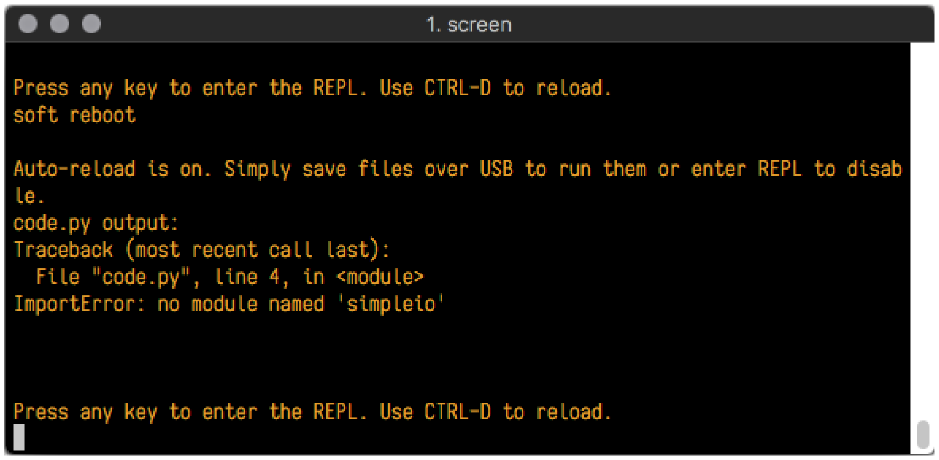
If you choose to load libraries as you need them, you may write up code that tries to use a library you haven't yet loaded. We're going to demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

This demonstration will only return an error if you do not have the required library loaded into the **lib** folder on your **CIRCUITPY** drive.

Let's use a modified version of the blinky example.

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.D13)

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.

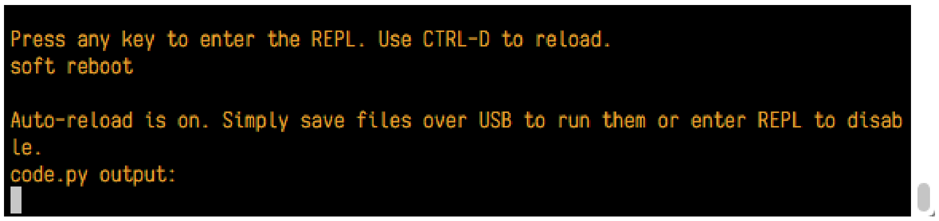


```
Press any key to enter the REPL. Use CTRL-D to reload.
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disab
le.
code.py output:
Traceback (most recent call last):
  File "code.py", line 4, in <module>
ImportError: no module named 'simpleio'



Press any key to enter the REPL. Use CTRL-D to reload.
```

We have an `ImportError`. It says there is `no module named 'simpleio'`. That's the one we just included in our code!

Click the link above to download the correct bundle. Extract the lib folder from the downloaded bundle file. Scroll down to find **simpleio.mpy**. This is the library file we're looking for! Follow the steps above to load an individual library file.

The LED starts blinking again! Let's check the serial console.



```
Press any key to enter the REPL. Use CTRL-D to reload.
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disab
le.
code.py output:
```

No errors! Excellent. You've successfully resolved an `ImportError`!

If you run into this error in the future, follow along with the steps above and choose the library that matches the one you're missing.

Library Install on Non-Express Boards

If you have a Trinket M0 or Gemma M0, you'll want to follow the same steps in the example above to install libraries as you need them. You don't always need to wait for an `ImportError` as you probably know what library you added to your code. Simply open the **lib** folder you downloaded, find the library you need, and drag it to the **lib** folder on your **CIRCUITPY** drive.

You may end up running out of space on your Trinket M0 or Gemma M0 even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find them in the Troubleshooting page in the Learn guides for your board.

Updating CircuitPython Libraries/Examples

Libraries and examples are updated from time to time, and it's important to update the files you have on your **CIRCUITPY** drive.

To update a single library or example, follow the same steps above. When you drag the library file to your lib folder, it will ask if you want to replace it. Say yes. That's it!

A new library bundle is released every time there's an update to a library. Updates include things like bug fixes and new features. It's important to check in every so often to see if the libraries you're using have been updated.

CircuitPython Code

CircuitPython Libraries

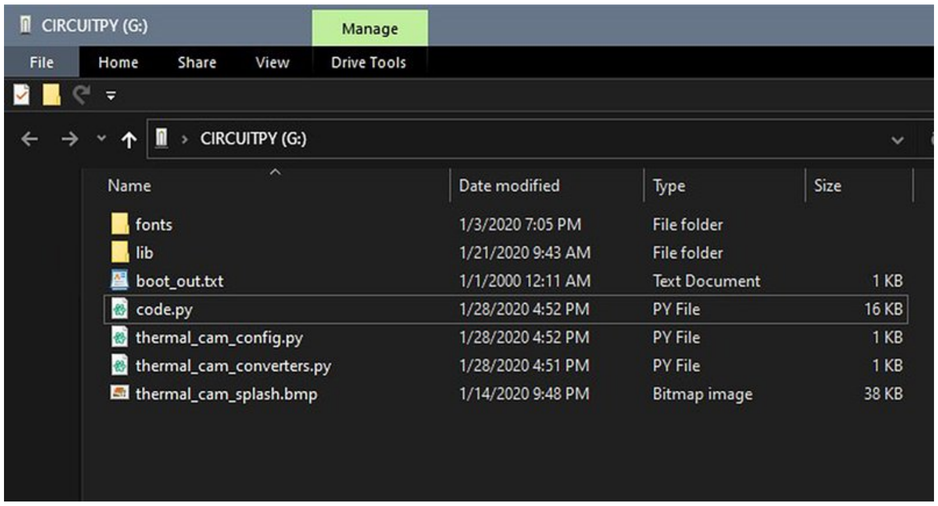
Your PyGamer should now have the latest version of CircuitPython and the collection of libraries needed for this project. Confirm that the **lib** folder on the PyGamer's **CIRCUITPY** flash drive contains these required libraries:

- **adafruit_simpleio**
- **adafruit_display_text**
- **adafruit_bitmap_font**
- **adafruit_display_shapes**
- **adafruit_amg88xx**
- **adafruit_pybadger**

PyGamer Thermal Camera Source Code

Download the project's source files and copy them to the PyGamer's **CIRCUITPY** root directory, with the exception of the **OpenSans-9.bdf** font file which should be placed into a **fonts** folder.

In the code window below, click the link Download: Project Zip. This will download a zip file containing the code (4 .py files) and the font file used for the display.



Name	Date modified	Type	Size
fonts	1/3/2020 7:05 PM	File folder	
lib	1/21/2020 9:43 AM	File folder	
boot_out.txt	1/1/2000 12:11 AM	Text Document	1 KB
code.py	1/28/2020 4:52 PM	PY File	16 KB
thermal_cam_config.py	1/28/2020 4:52 PM	PY File	1 KB
thermal_cam_converters.py	1/28/2020 4:51 PM	PY File	1 KB
thermal_cam_splash.bmp	1/14/2020 9:48 PM	Bitmap image	38 KB

The zip folder contains the following folders and files:

- **fonts** folder
  - *OpenSans-9.bdf* font file
- **code.py** main thermal camera code
- **thermal_cam_config.py** power-up default settings
- **thermal_cam_converters.py** temperature unit converter helpers
- **thermal_cam_splash.bmp** startup screen graphic

Here's the main CircuitPython code for the Thermal Camera. It's also contained in the project zip folder as ***code.py*** .

```
# Thermal_Cam_v32.py
# 2020-01-29 v3.2
# (c) 2020 Jan Goolsbey for Adafruit Industries

import time
import board
import displayio
from simpleio import map_range
from adafruit_display_text.label import Label
from adafruit_bitmap_font import bitmap_font
from adafruit_display_shapes.rect import Rect
import adafruit_amg88xx
from adafruit_pybadger import PyBadger
from thermal_cam_converters import celsius_to_fahrenheit, fahrenheit_to_celsius
# Load default alarm and min/max range values list from config file
from thermal_cam_config import ALARM_F, MIN_RANGE_F, MAX_RANGE_F

# Establish panel instance and check for joystick
panel = PyBadger(pixels_brightness=0.1)  # Set NeoPixel brightness
panel.pixels.fill(0)                     # Clear all NeoPixels
if hasattr(board, "JOYSTICK_X"):
    panel.has_joystick = True     # PyGamer
else: panel.has_joystick = False  # Must be PyBadge

# Establish I2C interface for the AMG8833 Thermal Camera
i2c = board.I2C()
amg8833 = adafruit_amg88xx.AMG88XX(i2c)

# Load the text font from the fonts folder
font = bitmap_font.load_font("/fonts/OpenSans-9.bdf")

# Display spash graphics and play startup tones
with open("/thermal_cam_splash.bmp", "rb") as bitmap_file:
    bitmap = displayio.OnDiskBitmap(bitmap_file)
    splash = displayio.Group()
    splash.append(displayio.TileGrid(bitmap,
                                     pixel_shader=displayio.ColorConverter()))
    board.DISPLAY.show(splash)
    time.sleep(0.1)  # Allow the splash to display
panel.play_tone(440, 0.1)  # A4
panel.play_tone(880, 0.1)  # A5

# The image sensor's design-limited temperature range
MIN_SENSOR_C = 0
MAX_SENSOR_C = 80
MIN_SENSOR_F = celsius_to_fahrenheit(MIN_SENSOR_C)
MAX_SENSOR_F = celsius_to_fahrenheit(MAX_SENSOR_C)

# Convert default alarm and min/max range values from config file
ALARM_C     = fahrenheit_to_celsius(ALARM_F)
MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)

# The board's integral display size
WIDTH  = board.DISPLAY.width   # 160 for PyGamer and PyBadge
HEIGHT = board.DISPLAY.height  # 128 for PyGamer and PyBadge

ELEMENT_SIZE = WIDTH // 10  # Size of element_grid blocks in pixels

# Default colors
BLACK   = 0x000000
RED     = 0xFF0000
```

```
ORANGE  = 0xFF8811
YELLOW  = 0xFFFF00
GREEN   = 0x00FF00
CYAN    = 0x00FFFF
BLUE    = 0x0000FF
VIOLET  = 0x9900FF
WHITE   = 0xFFFFFF
GRAY    = 0x444455

# Block colors for the thermal image grid
element_color = [GRAY, BLUE, GREEN, YELLOW, ORANGE, RED, VIOLET, WHITE]
# Text colors for on-screen parameters
param_list = [("ALARM", WHITE), ("RANGE", RED), ("RANGE", CYAN)]

### Helpers ###
def element_grid(col0, row0):  # Determine display coordinates for column, row
    x = int(ELEMENT_SIZE * col0 + 30)  # x coord + margin
    y = int(ELEMENT_SIZE * row0 + 1)   # y coord + margin
    return x, y  # Return display coordinates

def flash_status(text="", duration=0.05):  # Flash status message once
    status_label.color = WHITE
    status_label.text  = text
    time.sleep(duration)
    status_label.color = BLACK
    time.sleep(duration)
    return

def update_image_frame():  # Get camera data and display
    minimum = MAX_SENSOR_C  # Set minimum to sensor's maximum C value
    maximum = MIN_SENSOR_C  # Set maximum to sensor's minimum C value

    min_histo.text   = ""  # Clear histogram legend
    max_histo.text   = ""
    range_histo.text = ""

    sum_bucket = 0  # Clear bucket for building average value

    for row1 in range(0, 8):  # Parse camera data list and update display
        for col1 in range(0, 8):
            value = map_range(image[7 - row1][7 - col1],
                              MIN_SENSOR_C, MAX_SENSOR_C,
                              MIN_SENSOR_C, MAX_SENSOR_C)
            color_index = int(map_range(value, MIN_RANGE_C, MAX_RANGE_C, 0, 7))
            image_group[((row1 * 8) + col1) + 1].fill = element_color[color_index]
            sum_bucket = sum_bucket + value  # Calculate sum for average
            minimum = min(value, minimum)
            maximum = max(value, maximum)
    return minimum, maximum, sum_bucket

def update_histo_frame():
    minimum = MAX_SENSOR_C  # Set minimum to sensor's maximum C value
    maximum = MIN_SENSOR_C  # Set maximum to sensor's minimum C value

    min_histo.text   = str(MIN_RANGE_F)  # Display histogram legend
    max_histo.text   = str(MAX_RANGE_F)
    range_histo.text = "-RANGE-"

    sum_bucket = 0  # Clear bucket for building average value
    histo_bucket = [0, 0, 0, 0, 0, 0, 0, 0]  # Clear histogram bucket
    for row2 in range(7, -1, -1):  # Collect camera data and calculate spectrum
        for col2 in range(0, 8):
            value = map_range(image[col2][row2],
                              MIN_SENSOR_C, MAX_SENSOR_C,
                              MIN_SENSOR_C, MAX_SENSOR_C)
            histo_index = int(map_range(value, MIN_RANGE_C, MAX_RANGE_C, 0, 7))
            histo_bucket[histo_index] = histo_bucket[histo_index] + 1
            sum_bucket = sum_bucket + value  # Calculate sum for average
            minimum = min(value, minimum)
            maximum = max(value, maximum)

    for col2 in range(0, 8):  # Display histogram
        for row2 in range(0, 8):
            if histo_bucket[col2] / 8 > 7 - row2:
                image_group[((row2 * 8) + col2) + 1].fill = element_color[col2]
            else:
                image_group[((row2 * 8) + col2) + 1].fill = BLACK
    return minimum, maximum, sum_bucket

#pylint: disable=too-many-branches,too-many-statements
def setup_mode():  # Set alarm threshold and minimum/maximum range values
    status_label.color = WHITE
    status_label.text  = "-SET-"

    ave_label.color = BLACK  # Turn off average label and value display
    ave_value.color = BLACK

    max_value.text = str(MAX_RANGE_F)  # Display maximum range value
    min_value.text = str(MIN_RANGE_F)  # Display minimum range value

    time.sleep(0.8)  # Show SET status text before setting parameters
    status_label.text  = ""  # Clear status text

    param_index = 0  # Reset index of parameter to set

    # Select parameter to set
    while not panel.button.start:
        while (not panel.button.a) and (not panel.button.start):
            up, down = move_buttons(joystick=panel.has_joystick)
            if up:
                param_index = param_index - 1
            if down:
                param_index = param_index + 1
            param_index = max(0, min(2, param_index))
            status_label.text = param_list[param_index][0]
            image_group[param_index + 66].color = BLACK
            status_label.color = BLACK
            time.sleep(0.2)
            image_group[param_index + 66].color = param_list[param_index][1]
            status_label.color = WHITE
            time.sleep(0.2)

        if panel.button.a:  # Button A pressed
            panel.play_tone(1319, 0.030)  # E6
        while panel.button.a:  # wait for button release
            pass

        # Adjust parameter value
        param_value = int(image_group[param_index + 70].text)
        while (not panel.button.a) and (not panel.button.start):
            up, down = move_buttons(joystick=panel.has_joystick)
            if up:
                param_value = param_value + 1
            if down:
                param_value = param_value - 1
            param_value = max(MIN_SENSOR_F, min(MAX_SENSOR_F, param_value))
            image_group[param_index + 70].text = str(param_value)
            image_group[param_index + 70].color = BLACK
            status_label.color = BLACK
            time.sleep(0.05)
            image_group[param_index + 70].color = param_list[param_index][1]
            status_label.color = WHITE
            time.sleep(0.2)

        if panel.button.a:  # Button A pressed
            panel.play_tone(1319, 0.030)  # E6
        while panel.button.a:  # wait for button release
            pass

    # Exit setup process
    if panel.button.start:  # Start button pressed
        panel.play_tone(784, 0.030)  # G5
    while panel.button.start:  # wait for button release
        pass

    status_label.text = "RESUME"
    time.sleep(0.5)
    status_label.text = ""

    # Display average label and value
    ave_label.color = YELLOW
    ave_value.color = YELLOW
    return int(alarm_value.text), int(max_value.text), int(min_value.text)
#pylint: enable=too-many-branches,too-many-statements

def move_buttons(joystick=False):  # Read position buttons and joystick
    move_u = move_d = False
    if joystick:  # For PyGamer: interpret joystick as buttons
        if   panel.joystick[1] < 20000:
            move_u = True
        elif panel.joystick[1] > 44000:
            move_d = True
    else:  # For PyBadge read the buttons
        if panel.button.up:
            move_u = True
        if panel.button.down:
            move_d = True
    return move_u, move_d

### Define the display group ###
image_group = displayio.Group(max_size=77)

# Create a background color fill layer; image_group[0]
color_bitmap = displayio.Bitmap(WIDTH, HEIGHT, 1)
color_palette = displayio.Palette(1)
color_palette[0] = BLACK
background = displayio.TileGrid(color_bitmap, pixel_shader=color_palette,
                                x=0, y=0)
image_group.append(background)

# Define the foundational thermal image element layers; image_group[1:64]
#   image_group[#]=(row * 8) + column
for row in range(0, 8):
    for col in range(0, 8):
        pos_x, pos_y = element_grid(col, row)
        element = Rect(x=pos_x, y=pos_y,
                       width=ELEMENT_SIZE, height=ELEMENT_SIZE,
                       fill=None, outline=None, stroke=0)
        image_group.append(element)

# Define labels and values using element grid coordinates
status_label = Label(font, text="", color=BLACK, max_glyphs=6)
pos_x, pos_y = element_grid(2.5, 4)
status_label.x = pos_x
status_label.y = pos_y
image_group.append(status_label)  # image_group[65]

alarm_label = Label(font, text="alm", color=WHITE, max_glyphs=3)
pos_x, pos_y = element_grid(-1.8, 1.5)
alarm_label.x = pos_x
alarm_label.y = pos_y
image_group.append(alarm_label)  # image_group[66]

max_label = Label(font, text="max", color=RED, max_glyphs=3)
pos_x, pos_y = element_grid(-1.8, 3.5)
max_label.x = pos_x
max_label.y = pos_y
image_group.append(max_label)  # image_group[67]

min_label = Label(font, text="min", color=CYAN, max_glyphs=3)
pos_x, pos_y = element_grid(-1.8, 7.5)
min_label.x = pos_x
min_label.y = pos_y
image_group.append(min_label)  # image_group[68]

ave_label = Label(font, text="ave", color=YELLOW, max_glyphs=3)
pos_x, pos_y = element_grid(-1.8, 5.5)
ave_label.x = pos_x
ave_label.y = pos_y
image_group.append(ave_label)  # image_group[69]

alarm_value = Label(font, text=str(ALARM_F), color=WHITE, max_glyphs=5)
pos_x, pos_y = element_grid(-1.8, 0.5)
alarm_value.x = pos_x
alarm_value.y = pos_y
image_group.append(alarm_value)  # image_group[70]

max_value = Label(font, text=str(MAX_RANGE_F), color=RED, max_glyphs=5)
pos_x, pos_y = element_grid(-1.8, 2.5)
max_value.x = pos_x
max_value.y = pos_y
image_group.append(max_value)  # image_group[71]

min_value = Label(font, text=str(MIN_RANGE_F), color=CYAN, max_glyphs=5)
pos_x, pos_y = element_grid(-1.8, 6.5)
min_value.x = pos_x
min_value.y = pos_y
```

```
image_group.append(min_value)  # image_group[72]

ave_value = Label(font, text="---", color=YELLOW, max_glyphs=5)
pos_x, pos_y = element_grid(-1.8, 4.5)
ave_value.x = pos_x
ave_value.y = pos_y
image_group.append(ave_value)  # image_group[73]

min_histo = Label(font, text="", color=CYAN, max_glyphs=3)
pos_x, pos_y = element_grid(0.5, 7.5)
min_histo.x = pos_x
min_histo.y = pos_y
image_group.append(min_histo)  # image_group[74]

max_histo = Label(font, text="", color=RED, max_glyphs=3)
pos_x, pos_y = element_grid(6.5, 7.5)
max_histo.x = pos_x
max_histo.y = pos_y
image_group.append(max_histo)  # image_group[75]

range_histo = Label(font, text="", color=BLUE, max_glyphs=7)
pos_x, pos_y = element_grid(2.5, 7.5)
range_histo.x = pos_x
range_histo.y = pos_y
image_group.append(range_histo)  # image_group[76]

###--- PRIMARY PROCESS SETUP ---###
display_image = True   # Image display mode; False for histogram
display_hold  = False  # Active display mode; True to hold display
display_focus = False  # Standard display range; True to focus display range
orig_max_range_f = 0   # There are no initial range values
orig_min_range_f = 0

# Activate display and play welcome tone
board.DISPLAY.show(image_group)
panel.play_tone(880, 0.1)  # A5; ready to start looking

###--- PRIMARY PROCESS LOOP ---###
while True:
    if display_hold:  # Flash hold status text label
        flash_status("-HOLD-")
    else:
        image = amg8833.pixels  # Get camera data list if not in hold mode
        status_label.text = ""  # Clear hold mode status text label

    if display_image:  # Image display mode and gather min, max, and sum stats
        v_min, v_max, v_sum = update_image_frame()
    else:  # Histogram display mode and gather min, max, and sum stats
        v_min, v_max, v_sum = update_histo_frame()

    # Display alarm setting and maxumum, minimum, and average stats
    alarm_value.text = str(ALARM_F)
    max_value.text   = str(celsius_to_fahrenheit(v_max))
    min_value.text   = str(celsius_to_fahrenheit(v_min))
    ave_value.text   = str(celsius_to_fahrenheit(v_sum // 64))

    # Flash first NeoPixel and play alarm notes if alarm threshold is exceeded
    # Second alarm note frequency is proportional to value above threshold
    if v_max >= ALARM_C:
        panel.pixels.fill(RED)
        panel.pixels.fill(RED)
        panel.play_tone(880, 0.015)  # A5
        panel.play_tone(880 + (10 * (v_max - ALARM_C)), 0.015)  # A5
        panel.pixels.fill(BLACK)

    # See if a panel button is pressed
    if panel.button.a:  # Toggle display hold (shutter = button A)
        panel.play_tone(1319, 0.030)  # E6
        while panel.button.a:
            pass   # wait for button release
        if not display_hold:
            display_hold = True
        else:
            display_hold = False

    if panel.button.b:  # Toggle image/histogram mode (display mode = button B)
        panel.play_tone(659, 0.030)  # E5
        while panel.button.b:
            pass  # wait for button release
        if display_image:
            display_image = False
        else:
            display_image = True

    if panel.button.select:  # toggle focus mode (focus mode = select button)
        panel.play_tone(698, 0.030)  # F5
        if display_focus:
            display_focus = False  # restore previous (original) range values
            MIN_RANGE_F = orig_min_range_f
            MAX_RANGE_F = orig_max_range_f
            # update range min and max values in Celsius
            MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
            MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)
            flash_status("ORIG", 0.2)
        else:
            display_focus = True  # set range values to image min/max
            orig_min_range_f = MIN_RANGE_F
            orig_max_range_f = MAX_RANGE_F
            MIN_RANGE_F = celsius_to_fahrenheit(v_min)
            MAX_RANGE_F = celsius_to_fahrenheit(v_max)
            MIN_RANGE_C = v_min  # update range temp in Celsius
            MAX_RANGE_C = v_max  # update range temp in Celsius
            flash_status("FOCUS", 0.2)
        while panel.button.select:
            pass  # wait for button release

    if panel.button.start:  # activate setup mode (setup mode = start button)
        panel.play_tone(784, 0.030)  # G5
        while panel.button.start:
            pass  # wait for button release

        # Update alarm and range values
        ALARM_F, MAX_RANGE_F, MIN_RANGE_F = setup_mode()
        ALARM_C = fahrenheit_to_celsius(ALARM_F)
        MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
        MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)

    # bottom of primary loop
```

The Thermal Camera needs some helpers to convert back and forth between Celsius and Fahrenheit units. This file is contained in the project zip folder as **thermal_cam_converters.py**.

```
# thermal_cam_converters.py

def celsius_to_fahrenheit(deg_c=None):  # convert C to F; round to 1 degree C
    return round(((9 / 5) * deg_c) + 32)

def fahrenheit_to_celsius(deg_f=None):  # convert F to C; round to 1 degree F
    return round((deg_f - 32) * (5 / 9))
```

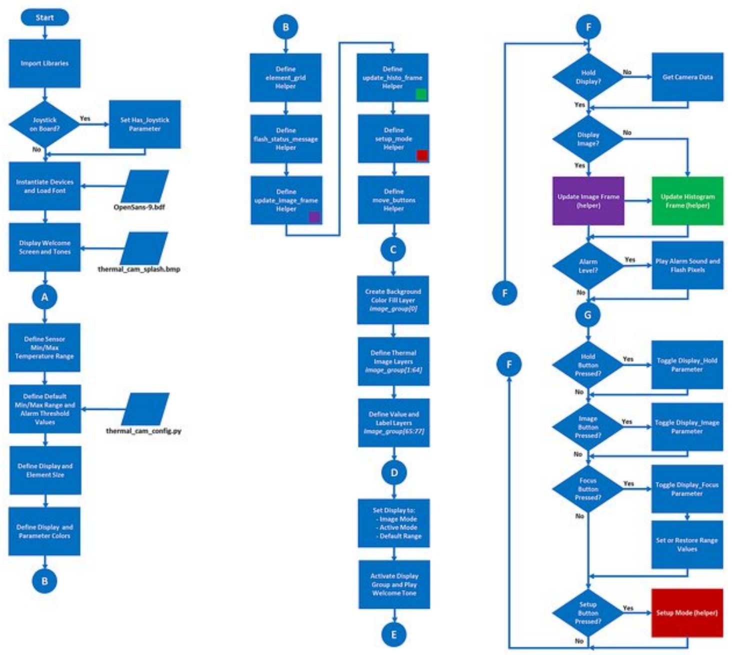
Finally, the power-up alarm and temperature display range settings are contained in the **thermal_cam_config.py** file. All values are in degrees Fahrenheit.

```
# thermal_cam_config.py
### Alarm and range default values in Farenheit ###
ALARM_F     = 120
MIN_RANGE_F =  60
MAX_RANGE_F = 120
```

After copying all the project files to the PyGamer, you'll see the camera's splash graphics, hear a couple of beeps, and the thermal image will appear.

The next section shows the features of the camera and how it operates.

CircuitPython Code Details


Start
Import Libraries
Joystick on Board?
Yes
Set Has_Joystick Parameter
No
Instantiate Devices and Load Font
OpenSans-9.bdf
Display Welcome Screen and Tones
thermal_cam_splash.bmp
A
Define Sensor Min/Max Temperature Range
Define Default Min/Max Range and Alarm Threshold Values
thermal_cam_config.py
Define Display and Element Size
Define Display and Parameter Colors
B
B
Define element_grid Helper
Define flash_status_message Helper
Define update_image_frame Helper
Define update_histo_frame Helper
Define setup_mode Helper
Define move_buttons Helper
C
Create Background Color Fill Layer image_group[0]
Define Thermal Image Layers image_group[1:64]
Define Value and Label Layers image_group[65:77]
D
Set Display to: - Image Mode - Active Mode - Default Range
Activate Display Group and Play Welcome Tone
E
F
Hold Display?
No
Get Camera Data
Yes
Display Image?
No
Yes
Update Image Frame (helper)
Update Histogram Frame (helper)
Alarm Level?
Yes
Play Alarm Sound and Flash Pixels
No
F
G
Hold Button Pressed?
Yes
Toggle Display_Hold Parameter
No
F
Image Button Pressed?
Yes
Toggle Display_Image Parameter
No
Focus Button Pressed?
Yes
Toggle Display_Focus Parameter
Set or Restore Range Values
No
Setup Button Pressed?
Yes
Setup Mode (helper)
No

The CircuitPython code for the Thermal Camera project is contained in three files, the main **code.py** file, the temperature unit conversion helper **thermal_cam_converters.py** file, and the start-up default parameter **thermal_cam_config.py** file.

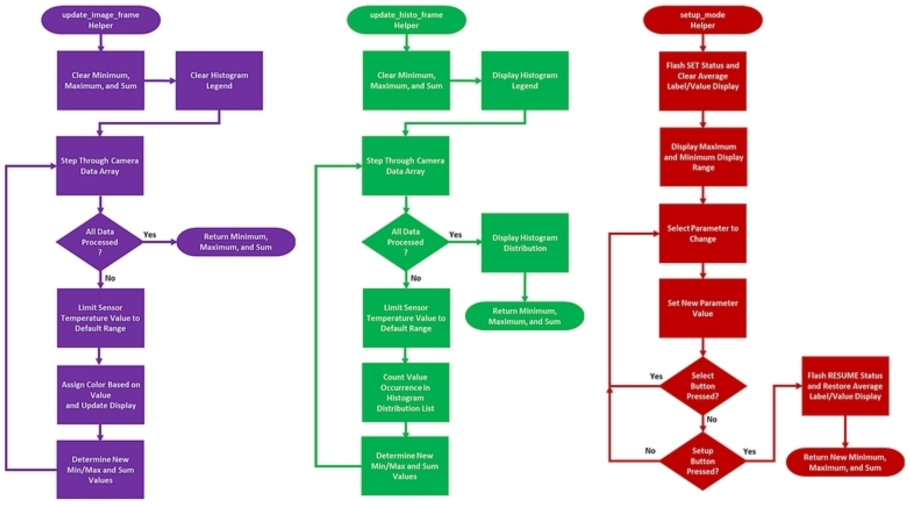
It looks daunting, but each section will be explained in detail in later pages in this guide.

Code Details

Let's take a walk through the code and look in more detail how each section works starting with the main module, **code.py**, on the next page.

Details: Main Module

code.py


update_image_frame Helper
Clear Minimum, Maximum, and Sum
Clear Histogram Legend
Step Through Camera Data Array
All Data Processed ?
Yes
Return Minimum, Maximum, and Sum
No
Limit Sensor Temperature Value to Default Range
Assign Color Based on Value and Update Display
Determine New Min/Max and Sum Values
update_histo_frame Helper
Clear Minimum, Maximum, and Sum
Display Histogram Legend
Step Through Camera Data Array
All Data Processed ?
Yes
Display Histogram Distribution
Return Minimum, Maximum, and Sum
No
Limit Sensor Temperature Value to Default Range
Count Value Occurrence in Histogram Distribution List
Determine New Min/Max and Sum Values
setup_mode Helper
Flash SET Status and Clear Average Label/Value Display
Display Maximum and Minimum Display Range
Select Parameter to Change
Set New Parameter Value
Yes
Select Button Pressed?
No
Flash RESUME Status and Restore Average Label/Value Display
No
Setup Button Pressed?
Yes
Return New Minimum, Maximum, and Sum

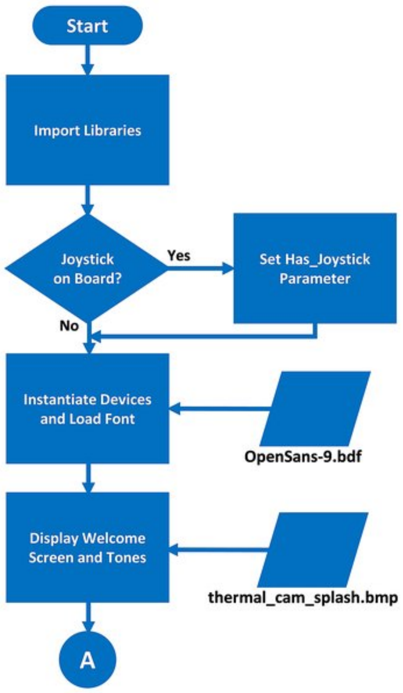
The main module, **code.py**, prepares and operates the Thermal Camera. It consists of the following major sections:

- **Import and Initialize:** Libraries, Devices, and Welcome Screen
- **Constants and Variables:** Display, Min/Max, and Alarm Threshold Values
- **Helpers:** Display, Buttons, and Setup Functions
- **Display:** Define Group Layers
- **Primary Process:** Setup and Loop

Things are started with importing libraries, establishing devices, and saying hello, detailed on the following pages.

Import and Initialize

Libraries, Devices, and Welcome Screen


Start
Import Libraries
Joystick on Board?
Yes
No
Set Has_Joystick Parameter
Instantiate Devices and Load Font
OpenSans-9.bdf
Display Welcome Screen and Tones
thermal_cam_splash.bmp
A

When the PyGamer's power is turned on, the **code.py** module first imports all the required libraries. That includes the **thermal_cam_converters.py** helper file that we'll review later.

```
# Thermal_Cam_v32.py
# 2020-01-29 v3.2
# (c) 2020 Jan Goolsbey for Adafruit Industries

import time
import board
import displayio
from simpleio import map_range
from adafruit_display_text.label import Label
from adafruit_bitmap_font import bitmap_font
from adafruit_display_shapes.rect import Rect
import adafruit_amg88xx
from adafruit_pybadger import PyBadger
from thermal_cam_converters import celsius_to_fahrenheit, fahrenheit_to_celsius
# Load default alarm and min/max range values list from config file
from thermal_cam_config import ALARM_F, MIN_RANGE_F, MAX_RANGE_F
```

Next, the code checks to see if the PyGamer's joystick is present and sets the `panel.has_joystick` flag to `True`. If not, then the host device is probably a PyBadge or EdgeBadge; this code will work for those devices, interpreting the D-Pad buttons like the joystick.

```
# Establish panel instance and check for joystick
panel = PyBadger(pixels_brightness=0.1)  # Set NeoPixel brightness
panel.pixels.fill(0)                     # Clear all NeoPixels
if hasattr(board, "JOYSTICK_X"):
    panel.has_joystick = True     # PyGamer
else: panel.has_joystick = False  # Must be PyBadge
```

The AMG8833 FeatherWing is instantiated on the I2C communications bus, then the ***OpenSans-9.bdf*** font file is loaded from the ***/fonts*** folder.

```
# Establish I2C interface for the AMG8833 Thermal Camera
i2c = board.I2C()
amg8833 = adafruit_amg88xx.AMG88XX(i2c)
```

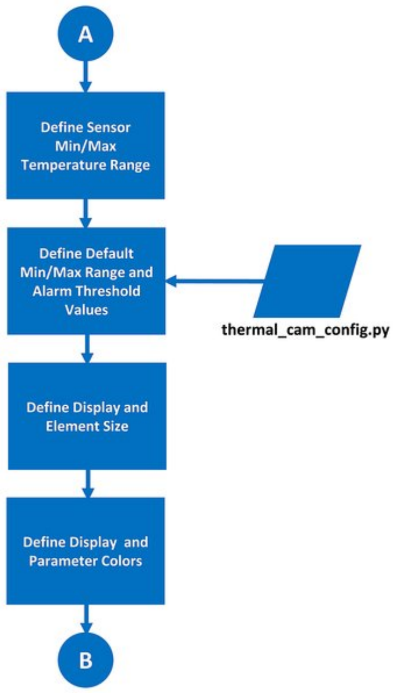
```
# Load the text font from the fonts folder
font = bitmap_font.load_font("/fonts/OpenSans-9.bdf")
```

Finally, the welcome graphics screen, **thermal_cam_splash.bmp** is displayed followed by a two-tone welcoming beep.

```
# Display spash graphics and play startup tones
with open("/thermal_cam_splash.bmp", "rb") as bitmap_file:
    bitmap = displayio.OnDiskBitmap(bitmap_file)
    splash = displayio.Group()
    splash.append(displayio.TileGrid(bitmap,
                  pixel_shader=displayio.ColorConverter()))
    board.DISPLAY.show(splash)
    time.sleep(0.1)  # Allow the splash to display
panel.play_tone(440, 0.1)  # A4
panel.play_tone(880, 0.1)  # A5
```

Constants and Variables

Display, Minimum/Maximum, and Alarm Threshold Values


A
Define Sensor Min/Max Temperature Range
Define Default Min/Max Range and Alarm Threshold Values
thermal_cam_config.py
Define Display and Element Size
Define Display and Parameter Colors
B

After saying hello, a few commonly used constants and variables are defined. These include the sensor's minimum and maximum values (0°C to 80°C for the AMG8833 FeatherWing) as well as the alarm threshold and display min/max settings that were previously loaded from the **thermal_cam_config.py** file. Default values in Celsius are converted to Fahrenheit where needed.

```
# The image sensor's design-limited temperature range
MIN_SENSOR_C = 0
MAX_SENSOR_C = 80
MIN_SENSOR_F = celsius_to_fahrenheit(MIN_SENSOR_C)
MAX_SENSOR_F = celsius_to_fahrenheit(MAX_SENSOR_C)

# Convert default alarm and min/max range values from config file
ALARM_C     = fahrenheit_to_celsius(ALARM_F)
MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)
```

Display width and height are retrieved from the PyGamer's board definitions. The size of individual display blocks is defined as 1/10th of the display's width, corresponding to 10 display columns. In this case, that's a 16 pixel square.

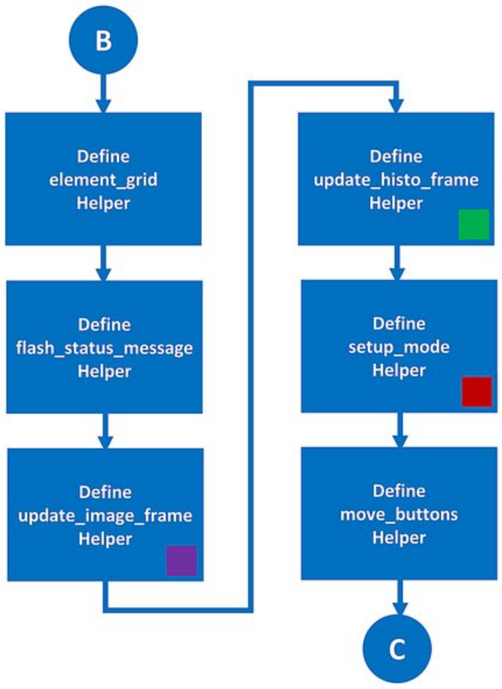
```
# The board's integral display size
WIDTH  = board.DISPLAY.width   # 160 for PyGamer and PyBadge
HEIGHT = board.DISPLAY.height  # 128 for PyGamer and PyBadge

ELEMENT_SIZE = WIDTH // 10  # Size of element_grid blocks in pixels
```

Color values are defined next and are incorporated into two lists, element_color and param_list that are used to color blocks in the image array according to temperature value and for text labels of measured values.

```
# Default colors
BLACK   = 0x000000
RED     = 0xFF0000
ORANGE  = 0xFF8811
YELLOW  = 0xFFFF00
GREEN   = 0x00FF00
CYAN    = 0x00FFFF
BLUE    = 0x0000FF
VIOLET  = 0x9900FF
WHITE   = 0xFFFFFF
GRAY    = 0x444455

# Block colors for the thermal image grid
element_color = [GRAY, BLUE, GREEN, YELLOW, ORANGE, RED, VIOLET, WHITE]
# Text colors for on-screen parameters
param_list = [("ALARM", WHITE), ("RANGE", RED), ("RANGE", CYAN)]
```

Helpers

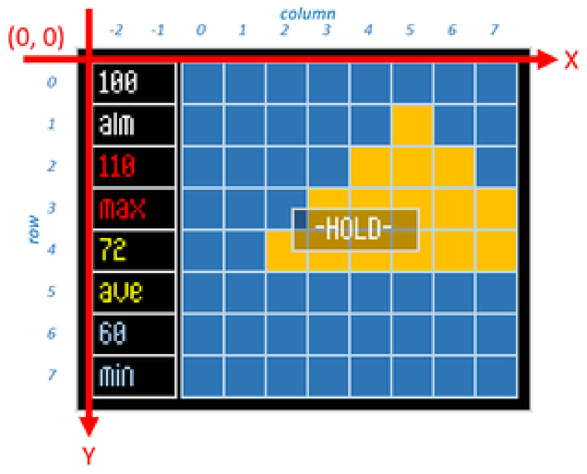
Helpers for Display, Buttons, and Setup Functions


B
Define element_grid Helper
Define flash_status_message Helper
Define update_image_frame Helper
Define update_histo_frame Helper
Define setup_mode Helper
Define move_buttons Helper
C

Helpers are used to simplify the primary loop code. The helpers:

- Calculate display coordinates for the row/column grid of blocks used for the image and text labels and values;
- Display a status message in the center of the image area;
- Display and refresh the sensor image;
- Display and refresh the histogram image;
- The parameter setup process.

`element_grid()` Helper



The `element_grid()` helper accepts a column/row tuple and returns a display x/y coordinate address tuple for positioning the upper left corner of each graphic box or text label.

The display is divided into 80 blocks in a matrix of 10 columns and 8 rows. The temperature value sidebar uses all rows of the two leftmost columns while the image area uses the remaining 8 columns.

The Coords assignment is a simple way to create the named tuple of x/y values for use by `element_grid()` or elsewhere.

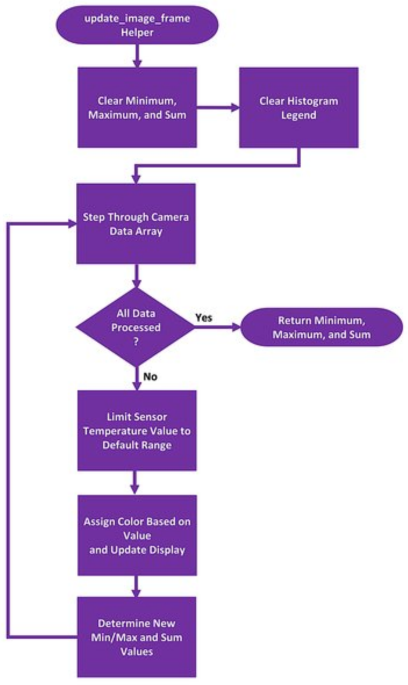
```
### Helpers ###
def element_grid(col0, row0):  # Determine display coordinates for column, row
    x = int(ELEMENT_SIZE * col0 + 30)  # x coord + margin
    y = int(ELEMENT_SIZE * row0 + 1)   # y coord + margin
    return x, y  # Return display coordinates
```

flash_status() Helper

The flash_status() helper accepts a text string and displays it in the status area of the display. The text appears for as white letters for a time specified by duration then as black letters for duration length in seconds. This is very useful for flashing a message that can be seen regardless of the background colors, especially handy while displaying a sensor image.

```
def flash_status(text="", duration=0.05):  # Flash status message once
    status_label.color = WHITE
    status_label.text  = text
    time.sleep(duration)
    status_label.color = BLACK
    time.sleep(duration)
    return
```

update_image_frame() Helper


update_image_frame Helper
Clear Minimum, Maximum, and Sum
Clear Histogram Legend
Step Through Camera Data Array
All Data Processed ?
Yes
Return Minimum, Maximum, and Sum
No
Limit Sensor Temperature Value to Default Range
Assign Color Based on Value and Update Display
Determine New Min/Max and Sum Values

The update_image_frame() helper looks through a list of 64 sensor temperature values stored by row and column in the image list. The helper analyzes all values in degrees Celsius leaving the conversion to Fahrenheit to the primary process loop.

When invoked, the helper clears variables used to find minimum and maximum sensor values as well as a "bucket" (sum_bucket) used to sum all values to be used later to calculate the average temperature. It also makes certain that histogram legend text is not displayed when showing a sensor image.

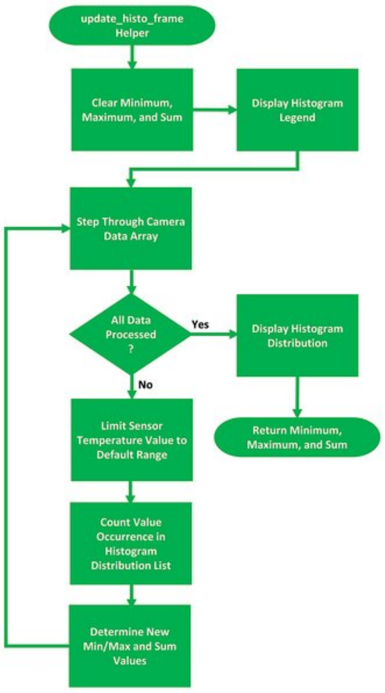
```
def update_image_frame():  # Get camera data and display
    minimum = MAX_SENSOR_C  # Set minimum to sensor's maximum C value
    maximum = MIN_SENSOR_C  # Set maximum to sensor's minimum C value

    min_histo.text   = ""  # Clear histogram legend
    max_histo.text   = ""
    range_histo.text = ""

    sum_bucket = 0  # Clear bucket for building average value
```

Next, the `image` list is examined one element at a time, starting at the upper left of the display's image area array and working down to the lower right. Each sensor element's temperature value is checked against the sensor's minimum and maximum allowed values since the sensor's output value can incorrectly fall outside of that range. Based on the sensor value's relationship to the entire temperature display range, a color is assigned to the value and the corresponding array block is filled with that color and displayed.

Finally, maximum, minimum, and `sum_bucket` values are returned to the primary process loop to convert to Fahrenheit and display in the sidebar.

```
for row1 in range(0, 8):  # Parse camera data list and update display
        for col1 in range(0, 8):
            value = map_range(image[7 - row1][7 - col1],
                              MIN_SENSOR_C, MAX_SENSOR_C,
                              MIN_SENSOR_C, MAX_SENSOR_C)
            color_index = int(map_range(value, MIN_RANGE_C, MAX_RANGE_C, 0, 7))
            image_group[((row1 * 8) + col1) + 1].fill = element_color[color_index]
            sum_bucket = sum_bucket + value  # Calculate sum for average
            minimum = min(value, minimum)
            maximum = max(value, maximum)
    return minimum, maximum, sum_bucket
```

`update_histo_frame()` Helper


update_histo_frame Helper
Clear Minimum, Maximum, and Sum
Display Histogram Legend
Step Through Camera Data Array
All Data Processed ?
Yes
Display Histogram Distribution
Return Minimum, Maximum, and Sum
No
Limit Sensor Temperature Value to Default Range
Count Value Occurrence in Histogram Distribution List
Determine New Min/Max and Sum Values

The update_histo_frame() helper collects a distribution of 8 temperature sub-ranges within the entire temperature display range (one for each color) and displays a histogram of relative temperature values. The helper scans all 64 sensor temperature values an counts the number of times a value falls within one of 8 sub-ranges. The helper analyzes all values in degrees Celsius leaving the conversion to Fahrenheit to the primary process loop.

When invoked, the helper clears variables used to find minimum and maximum sensor values, the average summing bucket, and the list containing the 8 "buckets" (histo_bucket) representing the distribution of values across the temperature display range. The helper displays the current temperature display range minimum and maximum values in the histogram legend area.

```
def update_histo_frame():
    minimum = MAX_SENSOR_C  # Set minimum to sensor's maximum C value
    maximum = MIN_SENSOR_C  # Set maximum to sensor's minimum C value

    min_histo.text   = str(MIN_RANGE_F)  # Display histogram legend
    max_histo.text   = str(MAX_RANGE_F)
    range_histo.text = "-RANGE-"

    sum_bucket = 0  # Clear bucket for building average value
```

Next, the 64 sensor values are examined one element at a time. The sensor element value is checked to make certain that it only falls within the allowable sensor range.

Each value is evaluated against the current display range to create the histogram distribution. An index is created (histo_index) that is used to increment the corresponding element of histo_bucket .

Minimum and maximum values are captured along with the summed values in sum_bucket as each element is evaluated.

The histo_bucket list now contains the distribution of 8 temperature sub-ranges found within the temperature display range. These values will be used to create the histogram image on the display.

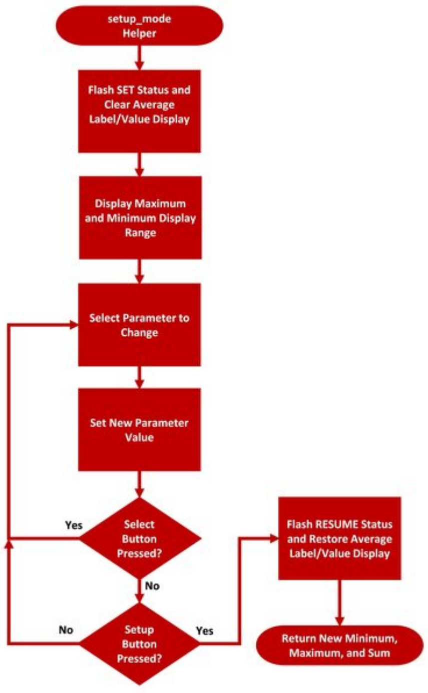
```
histo_bucket = [0, 0, 0, 0, 0, 0, 0, 0]  # Clear histogram bucket
    for row2 in range(7, -1, -1):  # Collect camera data and calculate spectrum
        for col2 in range(0, 8):
            value = map_range(image[col2][row2],
                              MIN_SENSOR_C, MAX_SENSOR_C,
                              MIN_SENSOR_C, MAX_SENSOR_C)
            histo_index = int(map_range(value, MIN_RANGE_C, MAX_RANGE_C, 0, 7))
            histo_bucket[histo_index] = histo_bucket[histo_index] + 1
            sum_bucket = sum_bucket + value  # Calculate sum for average
            minimum = min(value, minimum)
            maximum = max(value, maximum)
```

The second part of the helper updates the image area to show the histogram, starting at the upper left of the display's image array area and working down to the lower right. Each box in the array is filled with a color that corresponds to the relative temperature, proportional to the count in that sub-range. The remainder of boxes in the histogram display area are colored black.

Finally, minimum, maximum, and sum_bucket values are returned to the primary process loop to convert to Fahrenheit and display in the sidebar.

```
for col2 in range(0, 8):  # Display histogram
        for row2 in range(0, 8):
            if histo_bucket[col2] / 8 > 7 - row2:
                image_group[((row2 * 8) + col2) + 1].fill = element_color[col2]
            else:
                image_group[((row2 * 8) + col2) + 1].fill = BLACK
    return minimum, maximum, sum_bucket
```

setup_mode() Helper


setup_mode Helper
Flash SET Status and Clear Average Label/Value Display
Display Maximum and Minimum Display Range
Select Parameter to Change
Set New Parameter Value
Select Button Pressed?
Yes
No
Setup Button Pressed?
No
Yes
Flash RESUME Status and Restore Average Label/Value Display
Return New Minimum, Maximum, and Sum

The `setup_mode()` helper pauses normal operation and collects user input to set alarm threshold and display range min/max values. During the **Setup** mode, the display's average value and label are blanked since it's not possible to set the computed average value.

The joystick or PyBadge D-Pad is used to select the parameter to change and to increase or decrease the parameter value. The **HOLD** button acts as the parameter select button. Pressing the **SET** button at any time during the **Setup** mode will exit back to the primary process loop.

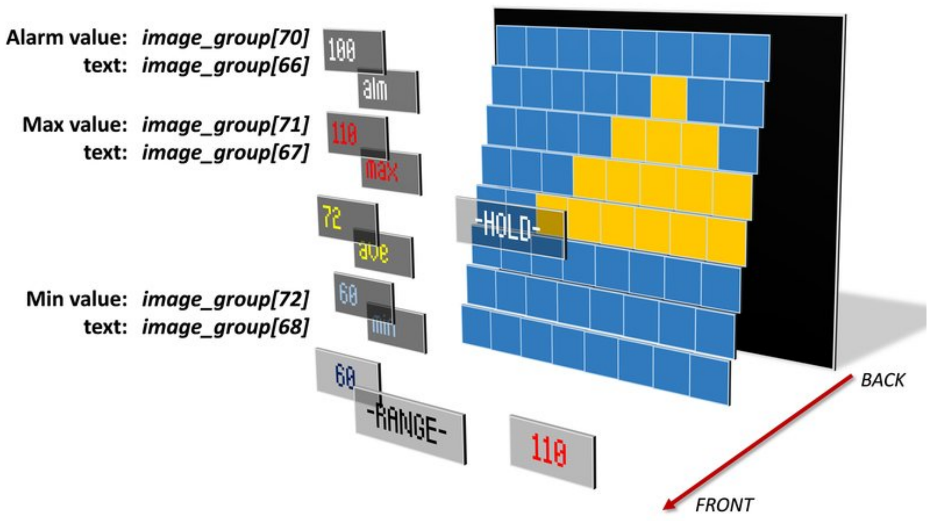
The first task is to temporarily display a status message that indicates the camera is in the **Setup** mode. The display's average value and label are blanked and the measured maximum and minimum values are replaced with the current maximum and minimum display range values (`MAX_RANGE_F` and `MIN_RANGE_F`).

After waiting a bit for the status message to be read and prior to watching for button and joystick changes, the index pointer (`param_index`) is reset to point to the alarm threshold parameter.

```
def setup_mode():  # Set alarm threshold and minimum/maximum range values
    status_label.color = WHITE
    status_label.text  = "-SET-"

    ave_label.color = BLACK  # Turn off average label and value display
    ave_value.color = BLACK

    max_value.text = str(MAX_RANGE_F)  # Display maximum range value
    min_value.text = str(MIN_RANGE_F)  # Display minimum range value

    time.sleep(0.8)  # Show SET status text before setting parameters
    status_label.text  = ""  # Clear status text

    param_index = 0  # Reset index of parameter to set
```

The following is the meat of the setup process. First, the process waits until the **SET** button has been released before moving on to choosing which parameter to set.

As long as the **HOLD** (select) or the **SET** (setup mode exit) buttons have not been pressed, the code loops. During the loop, the joystick is watched using the `move_buttons()` helper. If the joystick is moved down, the parameter index is incremented, pointing to the next parameter. If moved up, the index will point to the previous parameter. The parameter label text flashes black and white, indicating which parameter is ready to be changed.



In the `image_group` list (that is defined later, just before the primary process loop) the three parameter text labels for alarm, maximum, and minimum are sequentially positioned in the list:

- Alarm text label --> `image_group[66]`
- Maximum text label --> `image_group[67]`
- Minimum text label --> `image_group[68]`

Using an indexed position in `image_group` for the parameters makes it simpler to sequentially step from one parameter to the next.

```
# Select parameter to set
    while not panel.button.start:
        while (not panel.button.a) and (not panel.button.start):
            up, down = move_buttons(joystick=panel.has_joystick)
            if up:
                param_index = param_index - 1
            if down:
                param_index = param_index + 1
            param_index = max(0, min(2, param_index))
            status_label.text = param_list[param_index][0]
            image_group[param_index + 66].color = BLACK
            status_label.color = BLACK
            time.sleep(0.2)
            image_group[param_index + 66].color = param_list[param_index][1]
            status_label.color = WHITE
            time.sleep(0.2)
```

After the **HOLD** button is pressed and released, the selected parameter, represented by the value of `param_index`, can be changed.

The selected parameter value is incrementally changed by the joystick's up and down movements as provided to this helper from the `move_buttons()` helper. The new value is checked against and limited to the sensor's factory min/max limits (MIN_SENSOR_F , MAX_SENSOR_F).

In the image_group list the three parameter value labels for alarm, maximum, and minimum are sequentially positioned in the list:

- Alarm value label --> image_group[70]
- Maximum value label --> image_group[71]
- Minimum value label --> image_group[72]

The value label for the selected parameter is changed and displayed.

Meanwhile, a flashing status message indicates which type of parameter is being changed, either the alarm or one of the range values.

When the desired value is reached and the **HOLD** (select) button is pressed, the Setup process continues back to the parameter select mode. If **SET** is pressed instead, the **Setup** process prepares to exit back to the primary process loop.

```
if panel.button.a:  # Button A pressed
            panel.play_tone(1319, 0.030)  # E6
        while panel.button.a:  # wait for button release
            pass

        # Adjust parameter value
        param_value = int(image_group[param_index + 70].text)
        while (not panel.button.a) and (not panel.button.start):
            up, down = move_buttons(joystick=panel.has_joystick)
            if up:
                param_value = param_value + 1
            if down:
                param_value = param_value - 1
            param_value = max(MIN_SENSOR_F, min(MAX_SENSOR_F, param_value))
            image_group[param_index + 70].text = str(param_value)
            image_group[param_index + 70].color = BLACK
            status_label.color = BLACK
            time.sleep(0.05)
            image_group[param_index + 70].color = param_list[param_index][1]
            status_label.color = WHITE
            time.sleep(0.2)

        if panel.button.a:  # Button A pressed
            panel.play_tone(1319, 0.030)  # E6
        while panel.button.a:  # wait for button release
            pass

    # Exit setup process
    if panel.button.start:  # Start button pressed
        panel.play_tone(784, 0.030)  # G5
    while panel.button.start:  # wait for button release
        pass
```

Before exiting, a resumption status message is displayed and the display of the average label and value are restored.

Finally, the text strings that may have changed during the **Setup** process are converted to integer numeric values and returned to the primary process loop.

```
status_label.text = "RESUME"
    time.sleep(0.5)
    status_label.text = ""

    # Display average label and value
    ave_label.color = YELLOW
    ave_value.color = YELLOW
    return int(alarm_value.text), int(max_value.text), int(min_value.text)
```

move_buttons() Helper

The move_buttons() helper first resets the variables that indicate joystick movement or D-Pad button presses. If the joystick argument is True, the joystick movements beyond set thresholds will be registered like button depressions. For example, a value for panel.joystick[1] of less than 20000 means that the joystick was moved upwards; greater than 44000 indicates downward movement.

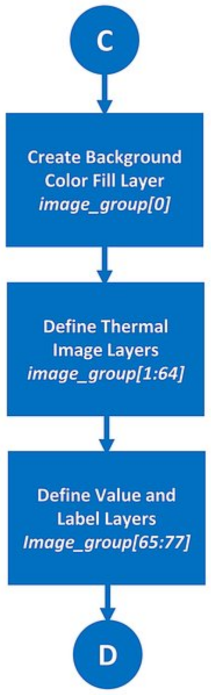
If the joystick argument is False, instead of watching the joystick, the D-Pad buttons are checked to see if any are depressed.

Finally, the movement indicating values are returned to the calling module.

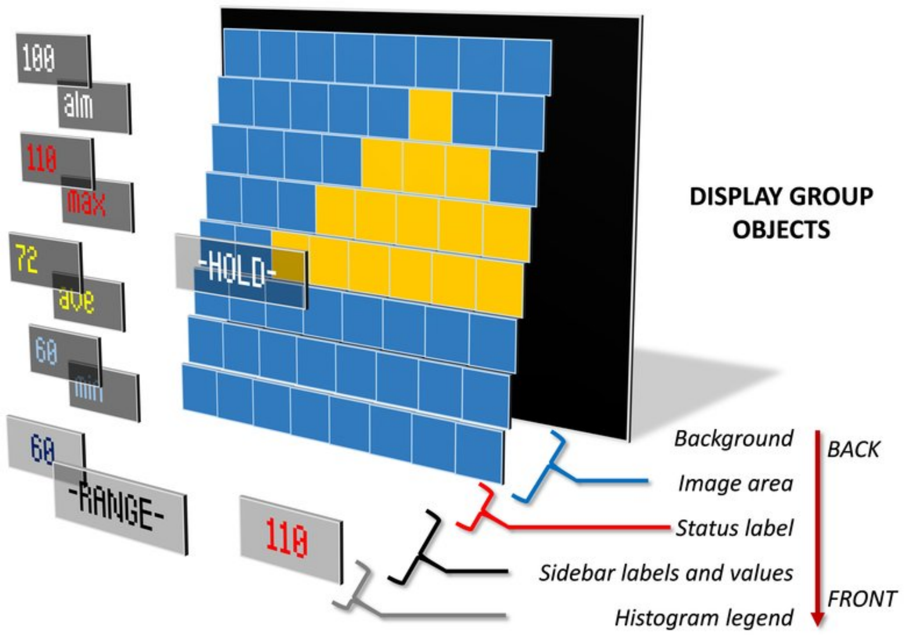
```
def move_buttons(joystick=False):  # Read position buttons and joystick
    move_u = move_d = False
    if joystick:  # For PyGamer: interpret joystick as buttons
        if   panel.joystick[1] < 20000:
            move_u = True
        elif panel.joystick[1] > 44000:
            move_d = True
    else:  # For PyBadge read the buttons
        if panel.button.up:
            move_u = True
        if panel.button.down:
            move_d = True
    return move_u, move_d
```

Display

Define Display Group Layers


C
Create Background Color Fill Layer
image_group[0]
Define Thermal Image Layers
image_group[1:64]
Define Value and Label Layers
Image_group[65:77]
D

Within CircuitPython's displayio library, a display group is a list of label or graphic attributes that are defined for each object of the display. This section of the Thermal Camera's primary process module defines the image_group display group that the camera will use to show measured values, the sensor image or histogram, status message, and the histogram legend.

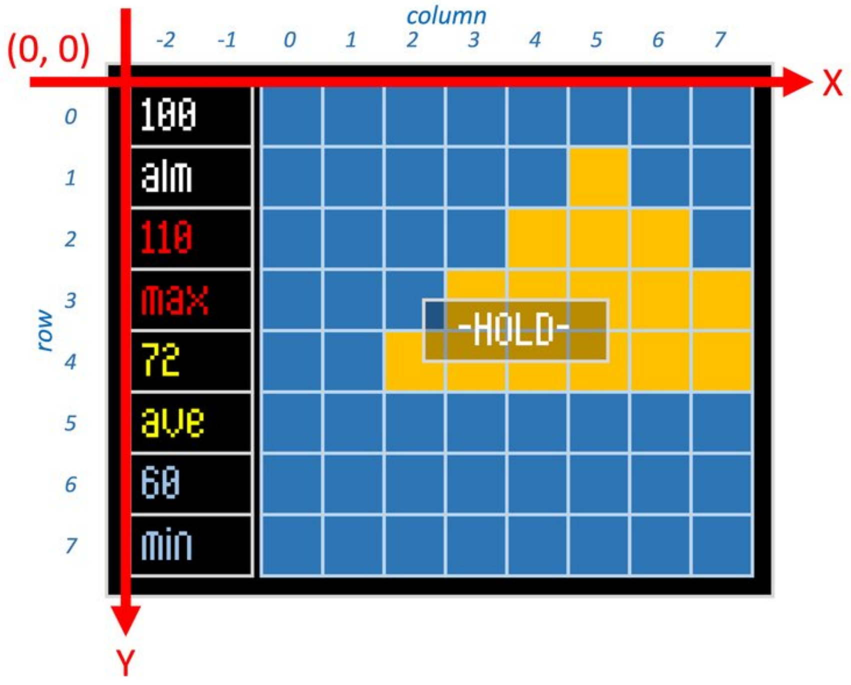


The camera's display group, `image_group`, consists of 77 layered objects that start with a black screen-sized background in the back-most position. The next 64 objects of the display make up the colored blocks used for the image area.

The status message label comes next, followed by the values and labels in the display sidebar area. Finally, objects that make up the histogram legend top off the stack of display objects in `image_group`.

The objects and their attributes are appended to the `image_group` list one-at-a-time when first defined. When appended to `image_group`, the attributes of each object are defined. For example, the alarm label **alm** is defined as a **Label** object with attributes that include the label's font, text contents, text color, and maximum character count (glyphs):

```
alarm_label = Label(font, text="alm", color=WHITE, max_glyphs=3)
```



The x/y position on the PyGamer's display screen is an attribute of the label object. Keeping with the grid block positioning scheme, the `element_grid()` helper is used to convert the block column/row position to the x/y coordinates of the board's display screen. In this case, `element_grid()` returns an x/y tuple for the intersection of column -1.8 and row 1.5:

```
pos = element_grid(-1.8, 1.5 )
```

The alarm label's display screen x/y coordinate attributes are set using the x/y tuple, `pos`:

```
alarm_label.x = pos.x
alarm_label.y = pos.y
```

After defining the display object's attributes, it is appended to the `image_group` display group:

```
image_group.append(alarm_label)
```

This process is repeated, starting from the back of the display and progressing towards the front, as each new object is appended to the display group.

Let's look at how all of the display objects are defined for the Thermal Camera.

Define Image Group and Create the Background Layer

The image_group list needs to be defined before we can append any display group objects. The max_size argument is set to the number of objects that will be used. In this case, we'll need 77 objects (image_group[0:76]).

Now that the display group is ready, we'll define the background bitmap object and its attributes. The bitmap's size is the same as the display's width and height; color is black. The bitmap will be placed at x/y coordinate 0,0 (the upper left corner of the display). After it's defined, the background object is appended as the first object in the image_group list.

```
### Define the display group ###
image_group = displayio.Group(max_size=77)

# Create a background color fill layer; image_group[0]
color_bitmap = displayio.Bitmap(WIDTH, HEIGHT, 1)
color_palette = displayio.Palette(1)
color_palette[0] = BLACK
background = displayio.TileGrid(color_bitmap, pixel_shader=color_palette,
                                x=0, y=0)
image_group.append(background)
```

Define the Thermal Image Display Group Layers

Next, the 64 squares used to represent sensor array temperatures will be defined and appended. Two for loops are used to step through each column and row of squares. Each square is defined as a rectangle with width and height equal to ELEMENT_SIZE . No color attribute is defined for the square, making it transparent -- for now.

```
# Define the foundational thermal image element layers; image_group[1:64]
#   image_group[#]=(row * 8) + column
for row in range(0, 8):
    for col in range(0, 8):
        pos_x, pos_y = element_grid(col, row)
        element = Rect(x=pos_x, y=pos_y,
                       width=ELEMENT_SIZE, height=ELEMENT_SIZE,
                       fill=None, outline=None, stroke=0)
        image_group.append(element)
```

Define the Text Label Display Group Layers

Finally, the remaining text objects that display legends and values are defined and appended to the image_group display group.

For each object, the label name is defined along with the font, text contents, font color, and the maximum number of characters to display. Next, the object's position attribute is defined using the element_grid() helper to calculate the hardware-specific x/y display coordinates. After the attributes are defined, each object is appended to image_group .

```
# Define labels and values using element grid coordinates
status_label = Label(font, text="", color=BLACK, max_glyphs=6)
status_label = Label(font, text=" ", color=BLACK, max_glyphs=6)
pos_x, pos_y = element_grid(2.5, 4)
status_label.x = pos_x
status_label.y = pos_y
image_group.append(status_label)  # image_group[65]

alarm_label = Label(font, text="alm", color=WHITE, max_glyphs=3)
pos_x, pos_y = element_grid(-1.8, 1.5)
alarm_label.x = pos_x
alarm_label.y = pos_y
image_group.append(alarm_label)  # image_group[66]

max_label = Label(font, text="max", color=RED, max_glyphs=3)
pos_x, pos_y = element_grid(-1.8, 3.5)
max_label.x = pos_x
max_label.y = pos_y
image_group.append(max_label)  # image_group[67]

min_label = Label(font, text="min", color=CYAN, max_glyphs=3)
pos_x, pos_y = element_grid(-1.8, 7.5)
min_label.x = pos_x
min_label.y = pos_y
image_group.append(min_label)  # image_group[68]

ave_label = Label(font, text="ave", color=YELLOW, max_glyphs=3)
pos_x, pos_y = element_grid(-1.8, 5.5)
ave_label.x = pos_x
ave_label.y = pos_y
image_group.append(ave_label)  # image_group[69]

alarm_value = Label(font, text=str(ALARM_F), color=WHITE, max_glyphs=5)
pos_x, pos_y = element_grid(-1.8, 0.5)
alarm_value.x = pos_x
alarm_value.y = pos_y
image_group.append(alarm_value)  # image_group[70]

max_value = Label(font, text=str(MAX_RANGE_F), color=RED, max_glyphs=5)
pos_x, pos_y = element_grid(-1.8, 2.5)
max_value.x = pos_x
max_value.y = pos_y
image_group.append(max_value)  # image_group[71]

min_value = Label(font, text=str(MIN_RANGE_F), color=CYAN, max_glyphs=5)
pos_x, pos_y = element_grid(-1.8, 6.5)
min_value.x = pos_x
min_value.y = pos_y
image_group.append(min_value)  # image_group[72]

ave_value = Label(font, text="---", color=YELLOW, max_glyphs=5)
pos_x, pos_y = element_grid(-1.8, 4.5)
ave_value.x = pos_x
ave_value.y = pos_y
image_group.append(ave_value)  # image_group[73]

min_histo = Label(font, text="", color=CYAN, max_glyphs=3)
pos_x, pos_y = element_grid(0.5, 7.5)
min_histo.x = pos_x
min_histo.y = pos_y
image_group.append(min_histo)  # image_group[74]
```

```
max_histo = Label(font, text="", color=RED, max_glyphs=3)
pos_x, pos_y = element_grid(6.5, 7.5)
max_histo.x = pos_x
max_histo.y = pos_y
image_group.append(max_histo)  # image_group[75]

range_histo = Label(font, text="", color=BLUE, max_glyphs=7)
pos_x, pos_y = element_grid(2.5, 7.5)
range_histo.x = pos_x
range_histo.y = pos_y
image_group.append(range_histo)  # image_group[76]
```

Fun Facts about Display Group Objects

Objects and their attributes in the display group can be accessed in two ways. The most commonly-used method is to assign a name attribute to the object. For example, the text of the status message label can be set to display the text *WELCOME* in this manner:

`status_label.text = "WELCOME"`

Objects in `image_group` can also be accessed by their indexed position in the display group. An index of 0 is the back-most object in the display group; the highest index value is front-most. The status message text can also be changed using the index:

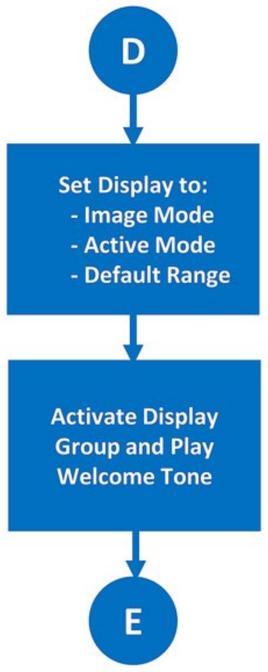
`image_group[65].text = "WELCOME"`

The Thermal Camera uses both techniques. Named display objects are used whenever possible to clearly identify which object is being changed. For efficiency, however, the index position method is used when stepping through a sequence of `image_group` objects, as when displaying the 64 colored blocks for the sensor image. The index position method is also used by the `setup_mode()` helper when moving on-screen to select the alarm, maximum, or minimum parameter.

Primary Process

Primary Process Setup and Loop

Setup



We're finally in the main portion of the code that controls the Thermal Camera's primary process. We still need to define a couple of things to get ready to use the camera.

The first portion of the primary process loop sets the default display modes. The process sets the camera to display active images using the display range that was provided by the **thermal_cam_config.py** file.

After the default mode flags are set, the `image_group` display group is activated and a "ready" tone is sounded.

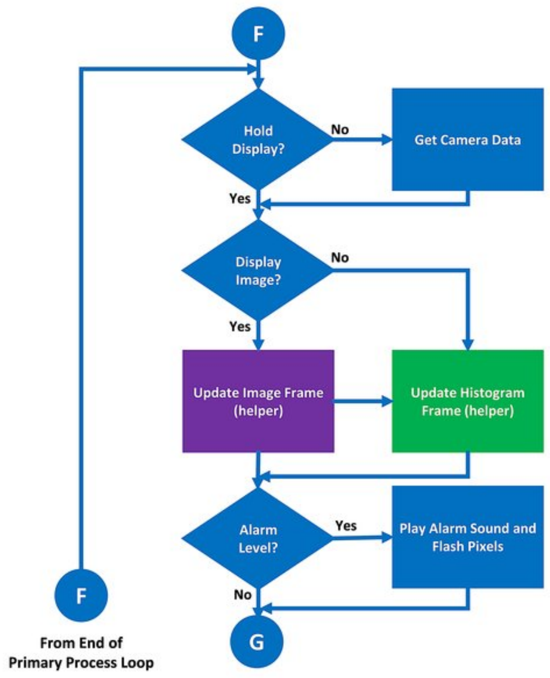
The primary is now ready to start looping.

```
###--- PRIMARY PROCESS SETUP ---###
display_image = True   # Image display mode; False for histogram
display_hold  = False  # Active display mode; True to hold display
display_focus = False  # Standard display range; True to focus display range
orig_max_range_f = 0   # There are no initial range values
orig_min_range_f = 0

# Activate display and play welcome tone
board.DISPLAY.show(image_group)
panel.play_tone(880, 0.1)  # A5; ready to start looking
```

Primary Process Loop, Part I

Because of its complexity, the primary process loop is divided into two sections to make it easier to understand. The first section fetches the image sensor's data, analyzes and displays the sensor data as an image or histogram, and checks to see if any of the sensor elements have exceeded the alarm threshold.

Retrieve Sensor Data, Display Image or Histogram, Check Alarm Threshold


F
Hold Display?
No
Get Camera Data
Yes
Display Image?
No
Yes
Update Image Frame (helper)
Update Histogram Frame (helper)
Alarm Level?
Yes
Play Alarm Sound and Flash Pixels
No
F
From End of Primary Process Loop
G

The image sensor's 64 data elements are moved into the `image` list as long as the `display_hold` flag is false; otherwise a "-HOLD-" status message is displayed. Also, the status message area is cleared whenever gathering image sensor data.

```
###--- PRIMARY PROCESS LOOP ---###
while True:
    if display_hold:  # Flash hold status text label
        flash_status("-HOLD-")
    else:
        image = amg8833.pixels  # Get camera data list if not in hold mode
        status_label.text = ""  # Clear hold mode status text label
```

This section checks the `display_image` flag to see whether to display a sensor image or histogram. If `display_image` is `True`, the `update_image_frame()` helper is used to display the data contained in the `image` list as an thermal image. When `False`, `update_histo_frame()` displays the data as a histogram distribution.

```
if display_image:  # Image display mode and gather min, max, and sum stats
        v_min, v_max, v_sum = update_image_frame()
    else:  # Histogram display mode and gather min, max, and sum stats
        v_min, v_max, v_sum = update_histo_frame()
```

After the display is updated with an image or histogram, the current alarm threshold value, `ALARM_F`, is converted to a string and displayed.

The minimum and maximum Celsius values returned by the `update_image_frame()` and `update_histo_frame()` helpers are converted to a string representation of the equivalent Fahrenheit temperature values and displayed.

The returned "sum bucket" value (`v_sum`) is divided by the number of sensor elements to produce an average temperature value. The average temperature value is converted to a string representation of the equivalent Fahrenheit value and displayed.

```
# Display alarm setting and maxumum, minimum, and average stats
    alarm_value.text = str(ALARM_F)
    max_value.text   = str(celsius_to_fahrenheit(v_max))
    min_value.text   = str(celsius_to_fahrenheit(v_min))
    ave_value.text   = str(celsius_to_fahrenheit(v_sum // 64))
```

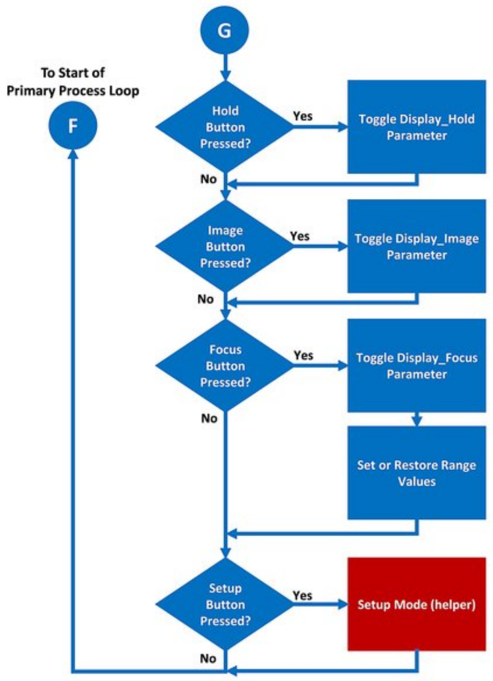
The next step in the primary process loop checks the returned maximum value against the current alarm threshold (`ALARM_C`). If the threshold is met or exceeded, the NeoPixels flash red while two tones are played through the speaker. The second tone's frequency is proportional to the maximum temperature relative value above the alarm threshold.

```
# Flash first NeoPixel and play alarm notes if alarm threshold is exceeded
    # Second alarm note frequency is proportional to value above threshold
    if v_max >= ALARM_C:
        panel.pixels.fill(RED)
        panel.play_tone(880, 0.015)  # A5
        panel.play_tone(880 + (10 * (v_max - ALARM_C)), 0.015)  # A5
        panel.pixels.fill(BLACK)
```

Primary Process Loop, Part II

The second portion of the primary process loop checks to see if any buttons have been pressed and sets the appropriate flags to modify display functions. This section also watches the **SET** button to activate the `setup_mode()` helper to permit changing camera parameters.

Watch the Buttons and Change Parameters


G
To Start of Primary Process Loop
F
Hold Button Pressed?
Yes
Toggle Display_Hold Parameter
No
Image Button Pressed?
Yes
Toggle Display_Image Parameter
No
Focus Button Pressed?
Yes
Toggle Display_Focus Parameter
No
Set or Restore Range Values
Setup Button Pressed?
Yes
Setup Mode (helper)
No

This section of the primary loop code looks for any of the buttons to be pressed. If any of the **HOLD**, **IMG/HST**, or **FOCUS** buttons are pressed, then the corresponding display mode flag is toggled. After a flag is toggled, the code waits until the button is released then plays a short confirmation tone before continuing in the primary loop.

```
# See if a panel button is pressed
    if panel.button.a:  # Toggle display hold (shutter = button A)
        panel.play_tone(1319, 0.030)  # E6
        while panel.button.a:
            pass   # wait for button release
        if not display_hold:
            display_hold = True
        else:
            display_hold = False

    if panel.button.b:  # Toggle image/histogram mode (display mode = button B)
        panel.play_tone(659, 0.030)  # E5
        while panel.button.b:
            pass  # wait for button release
        if display_image:
            display_image = False
        else:
            display_image = True
```

When the **FOCUS** button is first pressed, not only is the `display_focus` flag set, but the currently measured minimum and maximum values are used as the display range. When in **FOCUS** mode, the new display range values will provide more detail based on the range of temperatures measured in the image. Pressing the **FOCUS** button the second time will restore the original default or set display range values, restoring the original image resolution.

```
if panel.button.select:  # toggle focus mode (focus mode = select button)
        panel.play_tone(698, 0.030)  # F5
        if display_focus:
            display_focus = False  # restore previous (original) range values
            MIN_RANGE_F = orig_min_range_f
            MAX_RANGE_F = orig_max_range_f
            # update range min and max values in Celsius
            MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
            MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)
            flash_status("ORIG", 0.2)
        else:
            display_focus = True  # set range values to image min/max
            orig_min_range_f = MIN_RANGE_F
            orig_max_range_f = MAX_RANGE_F
            MIN_RANGE_F = celsius_to_fahrenheit(v_min)
            MAX_RANGE_F = celsius_to_fahrenheit(v_max)
            MIN_RANGE_C = v_min  # update range temp in Celsius
            MAX_RANGE_C = v_max  # update range temp in Celsius
            flash_status("FOCUS", 0.2)
        while panel.button.select:
            pass  # wait for button release
```

When the **SET** button is pressed, the `setup_mode()` helper is executed. When setup has completed, this section of code goes back to the start of the primary loop to start the display and button monitoring process again.

```
if panel.button.start:  # activate setup mode (setup mode = start button)
        panel.play_tone(784, 0.030)  # G5
        while panel.button.start:
            pass  # wait for button release

        # Update alarm and range values
        ALARM_F, MAX_RANGE_F, MIN_RANGE_F = setup_mode()
        ALARM_C = fahrenheit_to_celsius(ALARM_F)
        MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
        MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)

    # bottom of primary loop
```

Details: Converter Helpers

thermal_cam_converters.py

The **thermal_cam_converters.py** module consists of two temperature converters, one for Celsius to Fahrenheit and the other for Fahrenheit to Celsius. The value to be converted is passed as an argument to the appropriate helper. Because the Thermal Camera's sensor has limited accuracy, a rounded integer value is returned.

```
# thermal_cam_converters.py

def celsius_to_fahrenheit(deg_c=None):  # convert C to F; round to 1 degree C
    return round(((9 / 5) * deg_c) + 32)

def fahrenheit_to_celsius(deg_f=None):  # convert F to C; round to 1 degree F
    return round((deg_f - 32) * (5 / 9))
```

Details: Configuration Module

thermal_cam_config.py

When imported, the **thermal_cam_config.py** file provides the Thermal Camera's initial power-up alarm threshold as well as minimum and maximum display range values. The power-up configuration parameters can be changed by editing the file with your favorite text editor.

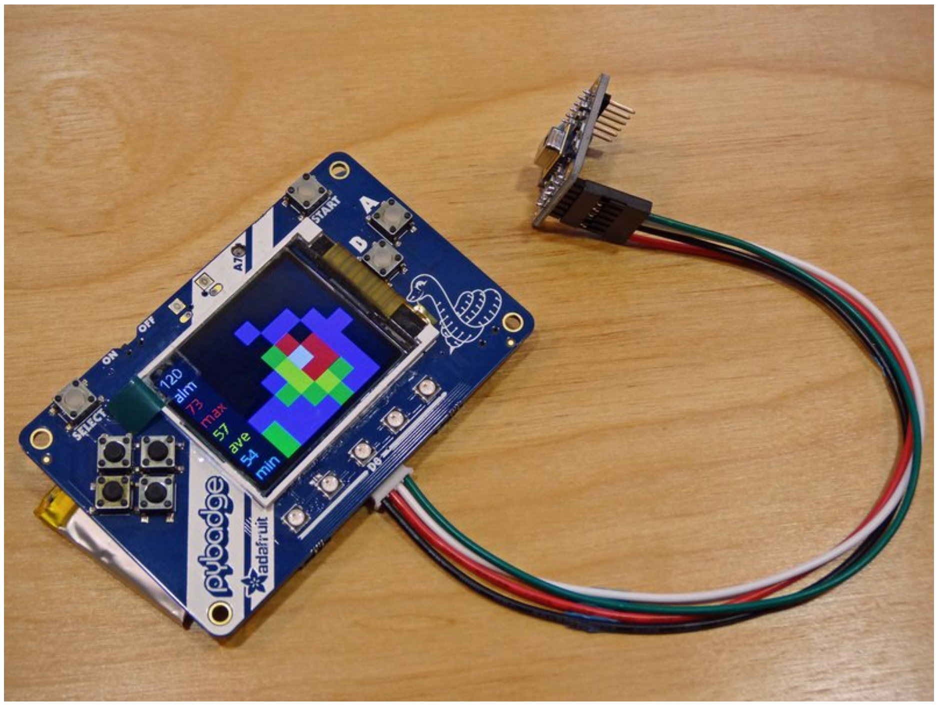
Values are in degrees Fahrenheit.

```
# thermal_cam_config.py
### Alarm and range default values in Farenheit ###
ALARM_F     = 120
MIN_RANGE_F =  60
MAX_RANGE_F = 120
```

Hacking the PyGamer Thermal Camera

This implementation of the Thermal Camera was focused on a limited set of requirements, one of which was to confirm that CircuitPython was fast enough to read, scale, and display an array of temperature values at a useful frame rate. Not only was CircuitPython's performance acceptable, it made it possible to add a few features to make the camera more robust.

To become even more useful, this implementation could easily be modified or expanded with some additional work. What would you do?



Here are some ideas to get you thinking about the camera's potential upgrades:

- The code is already compatible with the PyBadge. What other devices could easily run the code? Hallowing M4? Internet of things (IoT) use via the PyPortal?
- For a non-portable version, replace the AMG8833 FeatherWing with the breakout board version connected via Stemma so that the sensor could be positioned independently from the display board.
- Provide a setup selection for switching between Celsius and Fahrenheit displayed values.
- Add a graph image that displays min/max/ave/alarm over time.
- Can the display be interpolated or the sensor upgraded to improve detail? Would CircuitPython be agile enough to maintain a reasonable frame rate?
- Store snapshots to the SD card as comma-delimited or JSON files.
- Image transfer to a phone via Bluetooth LE.
- Jazz up the button labels.
- 3D printed enclosure designed for mounting permanently.

Below is a PyGamer case cover ("skin") designed for this project.

THERMAL CAMERA
THERMAL
CAMERA
HOLD
IMAGE
FOCUS
SET