



Star Trek Data Dispenser

Created by Ruiz Brothers



<https://learn.adafruit.com/star-trek-data-dispenser>

Last updated on 2026-04-08 03:15:41 AM UTC

Table of Contents

Overview	3
• Parts	
Circuit Diagram	8
Install CircuitPython	8
• CircuitPython Quickstart	
Code	11
• Code the Data Dispenser	
3D Printing	49
• Grid Setting for Sensor	
• Design Source Files	
Assembly	50
• Solder Motor	
• STEMMA QT Chain of Devices	
• STEMMA QT Sandwich	
• Mount Motor	
• Mount Boards	
• Connect Lipo Battery	
• Weathering Paint	
• Snap-fit Case Top	
• Assemble Magnetic Backing	
Usage	55

Overview



The TNG Data Dispenser is a wearable environmental monitor built into a 3D printed replica of the Drug Dispenser prop from "Encounter at Farpoint," the pilot episode of Star Trek: The Next Generation.



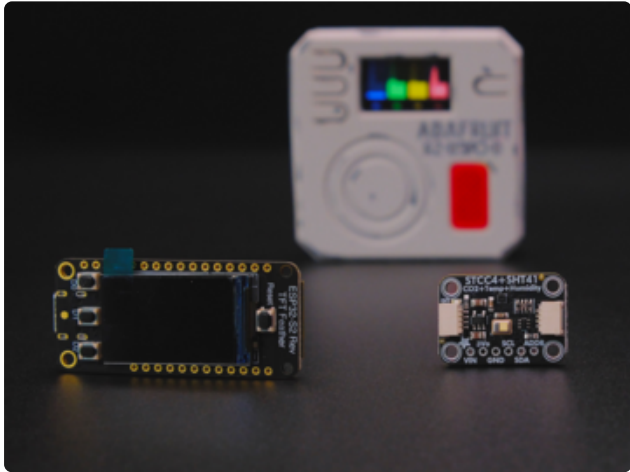
In the episode, the entity Q puts humanity on trial as a "dangerously savage child race." A soldier from Earth's Post-Atomic Horror era wears the device as an aerosol inhaler. "Rapid progress to where humans learned to control their military with drugs."

We thought - what if instead of dispensing drugs, it dispensed data? Real environmental data you can actually use.



This unit displays live CO2, humidity, temperature, and battery level as shaped colored bars that match the on-screen prop graphics. When CO2 levels get too high, a haptic motor buzzes to alert you.

Six display modes cycle through LCARS-style graphs, diagnostics, and a customizable name badge with a scannable QR code. It runs on a LiPo battery with automatic sleep and a brightness toggle for indoor use.



The Adafruit ESP32-S3 Reverse TFT Feather makes it easy to build display-first wearable projects. It has a built-in 240 x 135 color TFT and a STEMMA QT connector for plug-and-play I2C sensors. It features the ESP32-S3 with 4 megabytes of flash, 2 megabytes of PSRAM, WiFi, Bluetooth and an onboard MAX17048 battery fuel gauge that gives an accurate charge percentage and time remaining.

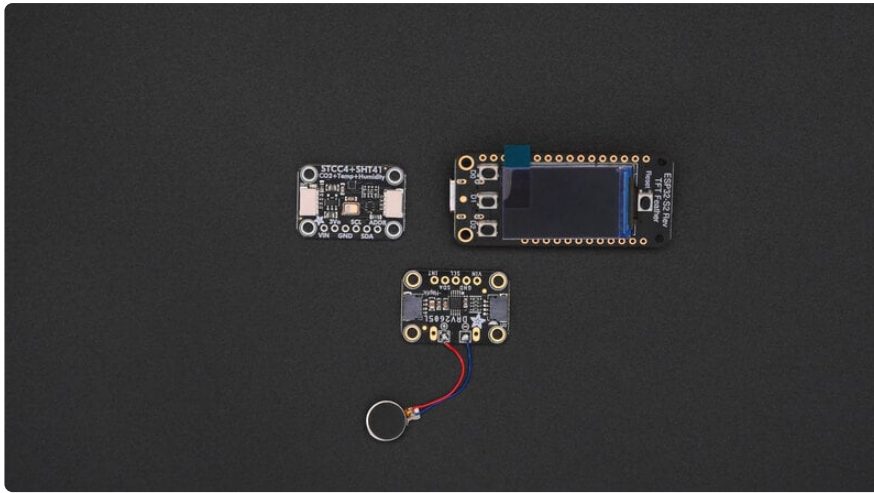
The STCC4 is a tiny CO2 sensor that can also read humidity, temperature and fits in just about any enclosure.



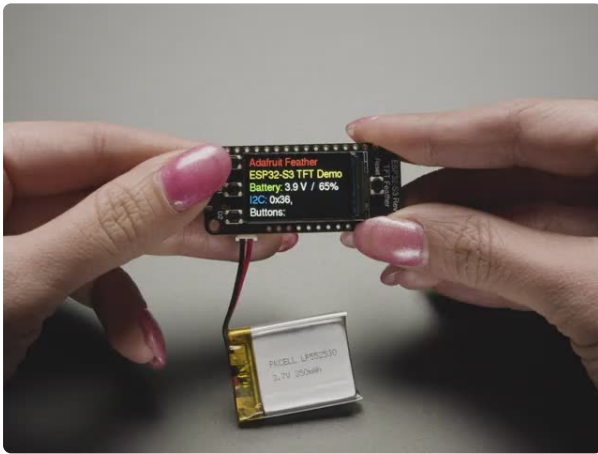
We think this is a great project for cosplay, conventions and maker events where you want a prop that actually does something.

The LCARS graphs make it feel like a real piece of Starfleet equipment. The name badge screen shows your name, department, registry number, and a QR code that links to your project page.



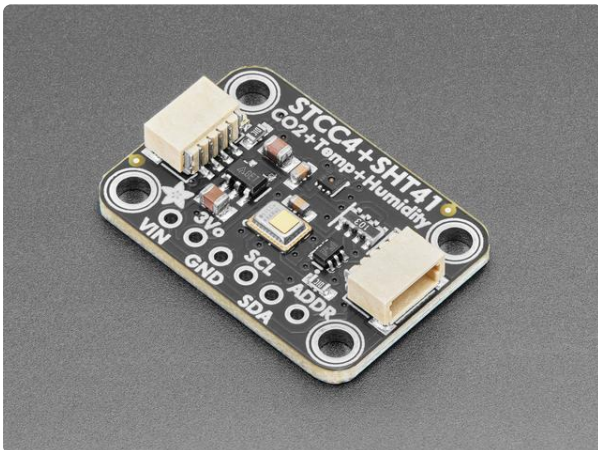


Parts



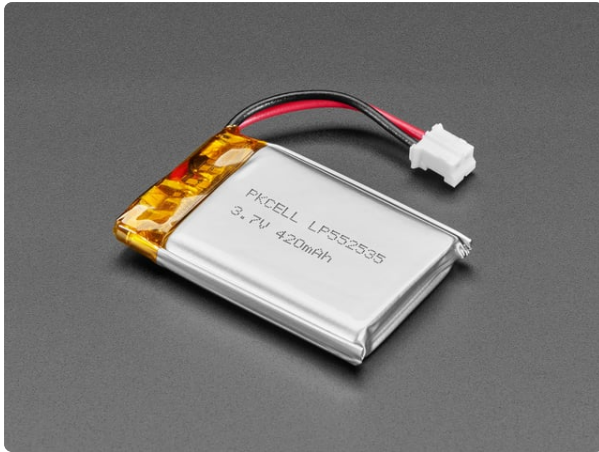
[Adafruit ESP32-S3 Reverse TFT Feather](https://www.adafruit.com/product/5691)
Like Missy Elliot, we like to "put our [Feather] down, flip it and reverse it" and that's exactly what...

<https://www.adafruit.com/product/5691>



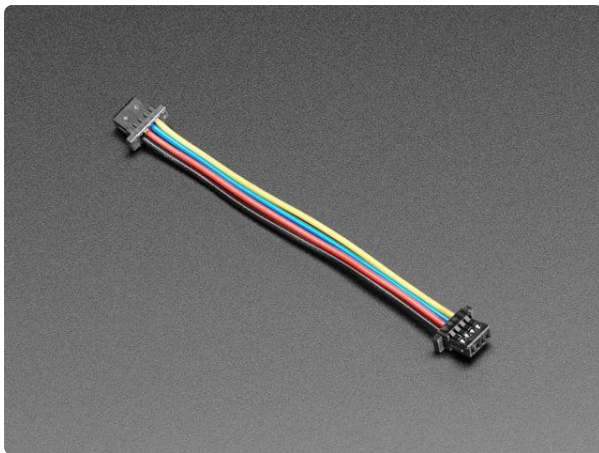
[Adafruit STCC4 and SHT41 - CO2, Temperature & Humidity Sensor](https://www.adafruit.com/product/6478)
CO₂ sensors are essential for determining if a room is too 'stuffy' - high CO₂ makes humans grumpy and tired. That's why it's always nice to take a deep breath...

<https://www.adafruit.com/product/6478>



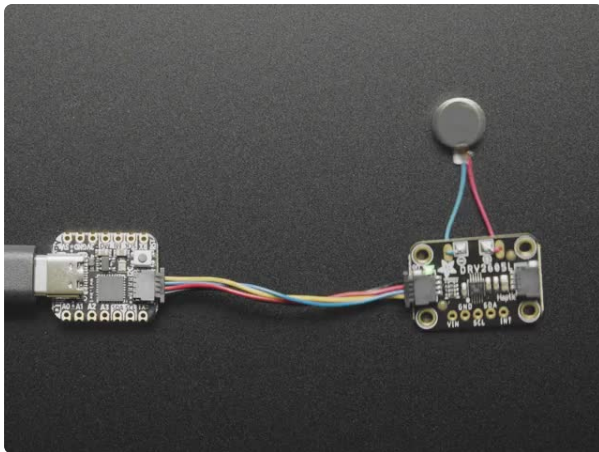
Lithium Ion Polymer Battery with Short Cable - 3.7V 420mAh

Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...
<https://www.adafruit.com/product/4236>



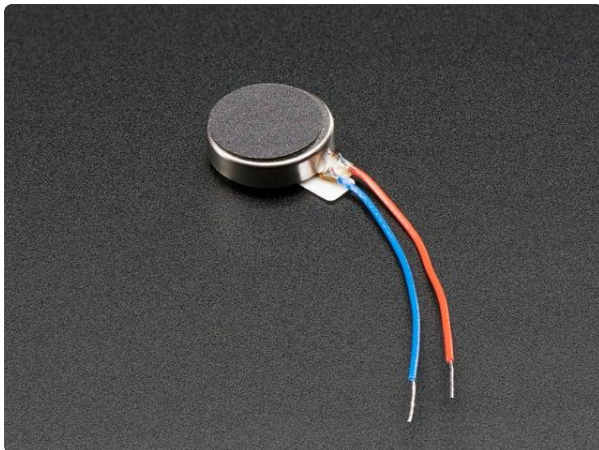
STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long

This 4-wire cable is 50mm / 1.9" long and fitted with JST SH female 4-pin connectors on both ends. Compared with the chunkier JST PH these are 1mm pitch instead of 2mm, but...
<https://www.adafruit.com/product/4399>



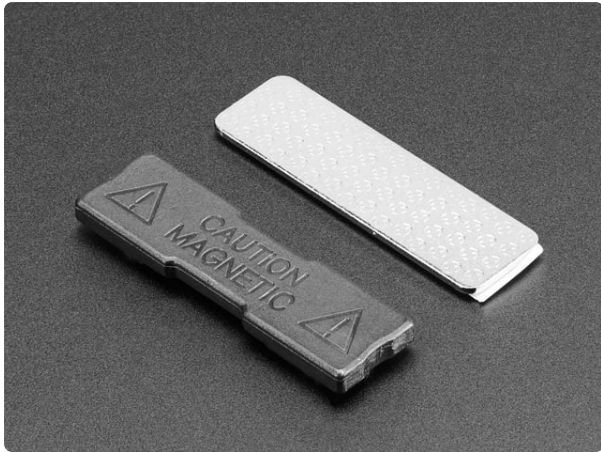
Adafruit DRV2605L Haptic Motor Controller - STEMMA QT / Qwiic

The DRV2605 from TI is a fancy little motor driver. Rather than controlling a stepper motor or DC motor, its designed specifically for controlling haptic motors
-...
<https://www.adafruit.com/product/2305>



Vibrating Mini Motor Disc

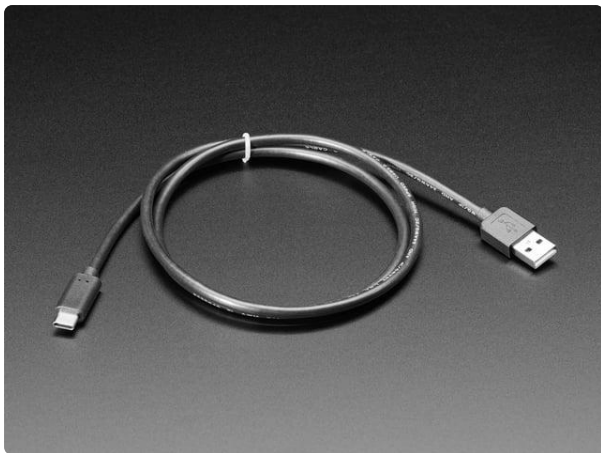
BZZZZZZZZZZ Feel that? That's your little buzzing motor, and for any haptic feedback project you'll want to pick up a few of them. These vibe motors are tiny discs,...
<https://www.adafruit.com/product/1201>



Magnetic Pin Back

These magnetic pin backs have two pieces: a metal bar with adhesive strip and a plastic piece with two strong rare-earth magnets. Affix the metal bar to your FLORA projects-- it's...

<https://www.adafruit.com/product/1170>



USB Type A to Type C Cable - approx 1 meter / 3 ft long

As technology changes and adapts, so does Adafruit. This USB Type A to Type C cable will help you with the transition to USB C, even if you're still...

<https://www.adafruit.com/product/4474>

4 x M2.5X6mm Screws

2 for the Feather 2 for the Magnet holder

2 x M2X6mm Screws

Feather mounts near the Reset button

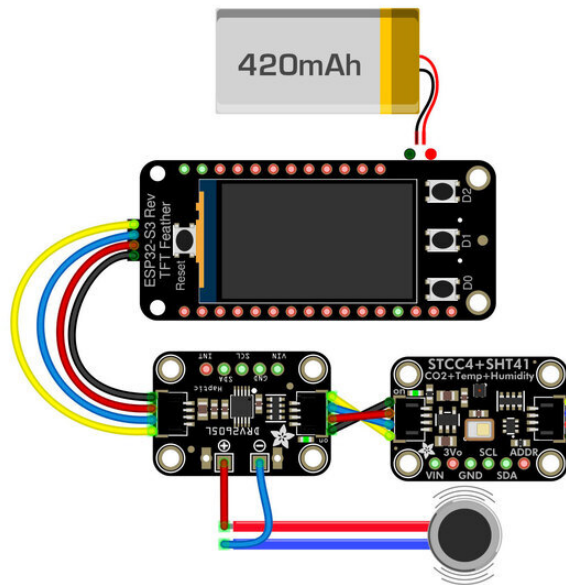
4 x M2X10mm Screws

STEMMA holder



Circuit Diagram

This provides a visual reference for wiring of the components. This diagram was created using [Fritzing software \(https://adafru.it/oEP\)](https://adafru.it/oEP).



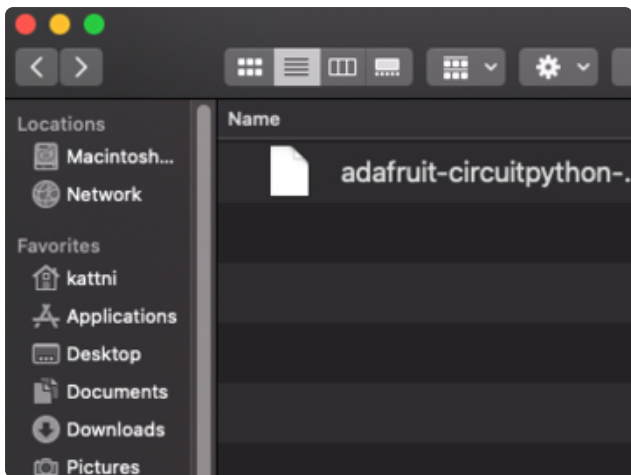
Install CircuitPython

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

CircuitPython Quickstart

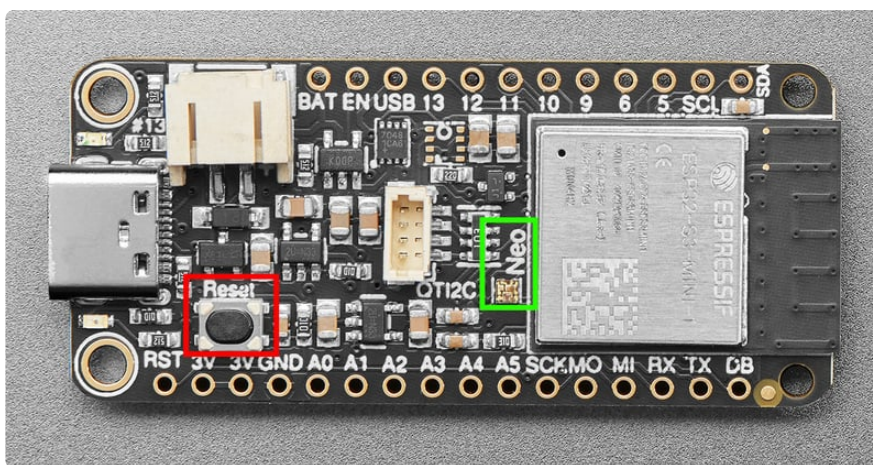
Follow this step-by-step to quickly get CircuitPython running on your board.

<https://adafru.it/18A3>



Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.



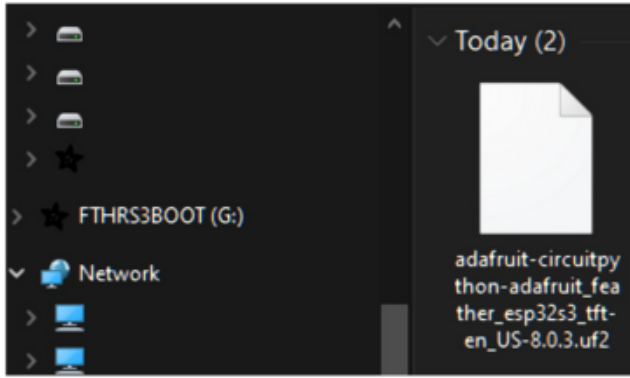
Plug your board into your computer, using a known-good data-sync cable, directly, or via an adapter if needed.

Double-click the **reset** button (highlighted in red above), and you will see the **RGB status LED(s)** turn green (highlighted in green above). If you see red, try another port, or if you're using an adapter or hub, try without the hub, or different adapter or hub.

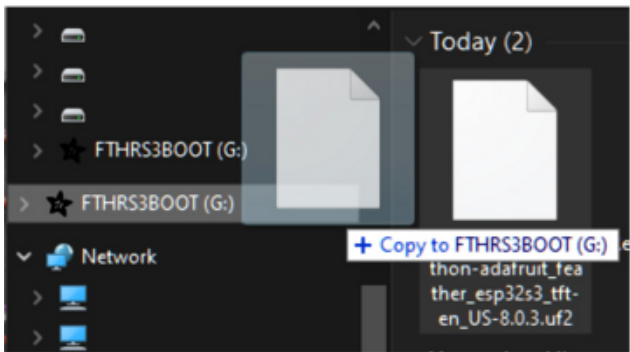
For this board, tap reset and wait for the LED to turn purple, and as soon as it turns purple, tap reset again. The second tap needs to happen while the LED is still purple.

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!

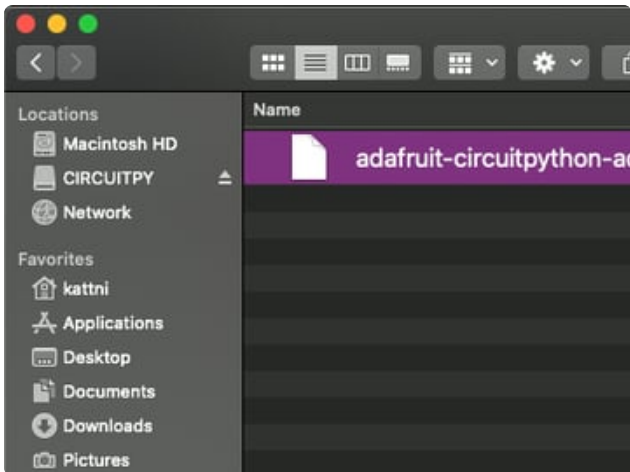
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



You will see a new disk drive appear called **FTHRS3BOOT**.



Drag the `adafruit_circuitpython_etc.uf2` file to **FTHRS3BOOT**.



The **BOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it!

Code

Code the Data Dispenser

The `code.py` file runs the entire Data Dispenser - display graphics, sensor reads, button interactions, LCARS screens and sleep management. This page walks through each major section so you can customize the project or learn from the techniques used.

Click on the Download Project Bundle to get all the files for this project. Save the zip file where you can find it. Plug your Feather via USB into your computer via a known good USB data+power. A folder called **CIRCUITPY** should show up in Finder or File Explorer (depending on your operating system). Open the zip file and copy the files to the **CIRCUITPY** drive.



```
# SPDX-FileCopyrightText: 2026 Pedro Ruiz for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# pylint: disable=global-statement,invalid-name,redefined-outer-name,too-many-lines

"""Star Trek TNG Data Dispenser Prop - Environmental Monitor.

Displays CO2, humidity, temperature and battery level as
shaped colored bars on the ESP32-S3 Reverse TFT Feather.

Button map:
D0 short: rescan animation (bars) / return to bars
D0 double: demo mode (auto-cycle all screens)
D0 long: CO2 offset calibration
D1 short: toggle Fahrenheit / Celsius
D1 long: toggle LCARS name badge
D2 short: cycle views (bars -> stats -> CO2 graph)
D2 long: toggle display brightness
D0+D2 long: sleep mode (any button wakes)
5min idle: auto sleep"""

import time
import struct
import alarm
import board
import digitalio
import displayio
import vectorio
```

```

import terminalio
from adafruit_display_text import label
import adafruit_stcc4
import adafruit_max1704x
import adafruit_driv2605

# =====
# TUNABLES
# =====
CO2_MIN = 400
CO2_MAX = 2000
CO2_WARN = 700
CO2_FLOOR_PCT = 8
TEMP_MIN = 15.0
TEMP_MAX = 40.0
UPDATE_INTERVAL = 2
HAPTIC_COOLDOWN = 30
LONG_PRESS_MS = 2000 # hold time for long press
IDLE_SLEEP_S = 300 # 5 min idle -> deep sleep
DIM_BRIGHTNESS = 0.15 # dimmed display brightness
FULL_BRIGHTNESS = 1.0

# =====
# NAME BADGE - edit these to personalise your prop
# =====
BADGE_NAME = "P. RUIZ" # your name (2x scale)
BADGE_TITLE = "ENGINEERING" # department / role
BADGE_ID = "NCC-1701-D" # registry / ID number
BADGE_PROJECT = "TNG DATA DISPENSER" # project name
BADGE_QR_URL = "https://learn.adafruit.com/u/pixil3d"

# =====
# COLORS
# =====
COLOR_CO2 = 0x4466FF
COLOR_HUMIDITY = 0x44FF66
COLOR_TEMP = 0xFFCC00
COLOR_BATTERY = 0xFF4444
BG_COLOR = 0x000000
BAR_COLORS = (
    COLOR_CO2, COLOR_HUMIDITY, COLOR_TEMP, COLOR_BATTERY
)

DIM_FACTOR = 5
GRAD_DIM_FACTOR = 20
SOFT_DIM_FACTOR = 4
MID_DIM_FACTOR = 10

def dim_color(color, factor=DIM_FACTOR):
    """Return a dimmed version of a 24-bit color."""
    r = ((color >> 16) & 0xFF) // factor
    g = ((color >> 8) & 0xFF) // factor
    b = (color & 0xFF) // factor
    return (r << 16) | (g << 8) | b

TRACK_COLORS = tuple(
    dim_color(c, DIM_FACTOR) for c in BAR_COLORS
)
GRAD_COLORS = tuple(
    dim_color(c, GRAD_DIM_FACTOR) for c in BAR_COLORS
)
SOFT_COLORS = tuple(
    dim_color(c, SOFT_DIM_FACTOR) for c in BAR_COLORS
)
OUTLINE_COLORS = tuple(
    dim_color(c, MID_DIM_FACTOR) for c in BAR_COLORS
)

```

```

# =====
# LAYOUT - 240 x 135 display
# =====
DISPLAY_W = 240
DISPLAY_H = 135
NUM_BARS = 4
BAR_W = 45
BAR_GAP = 10
BAR_H = 90
BAR_Y = 8
LABEL_Y = BAR_Y + BAR_H + 16
TOTAL_BAR_W = NUM_BARS * BAR_W + (NUM_BARS - 1) * BAR_GAP
X_OFFSET = (DISPLAY_W - TOTAL_BAR_W) // 2

# =====
# BAR SHAPE
# =====
PEAK_L = 11
PEAK_R = 13
TAPER_L = 2
TAPER_R = 19
SHELF_L_Y = 57
SHELF_R_Y = 56
NUM_PTS = 8
OUTLINE_PX = 1

def bar_shape():
    """Full bar outline as 8 points."""
    return [
        (PEAK_L, 0), (PEAK_R, 0),
        (TAPER_R, SHELF_R_Y), (BAR_W, SHELF_R_Y),
        (BAR_W, BAR_H), (0, BAR_H),
        (0, SHELF_L_Y), (TAPER_L, SHELF_L_Y),
    ]

def outline_shape():
    """Bar outline expanded by OUTLINE_PX."""
    o = OUTLINE_PX
    return [
        (PEAK_L - o, 0), (PEAK_R + o, 0),
        (TAPER_R + o, SHELF_R_Y),
        (BAR_W + o, SHELF_R_Y - o),
        (BAR_W + o, BAR_H + o),
        (0 - o, BAR_H + o),
        (0 - o, SHELF_L_Y - o),
        (TAPER_L - o, SHELF_L_Y),
    ]

def mask_shape(pct):
    """Mask polygon covering unfilled portion."""
    fill_y = max(int(BAR_H * (100 - pct) / 100), 1)
    if fill_y <= SHELF_R_Y:
        frac = fill_y / SHELF_R_Y
        ml = int(PEAK_L + (TAPER_L - PEAK_L) * frac)
        mr = int(PEAK_R + (TAPER_R - PEAK_R) * frac)
        return [
            (PEAK_L, 0), (PEAK_R, 0),
            (mr, fill_y), (mr, fill_y),
            (mr, fill_y), (ml, fill_y),
            (ml, fill_y), (ml, fill_y),
        ]
    return [
        (PEAK_L, 0), (PEAK_R, 0),
        (TAPER_R, SHELF_R_Y), (BAR_W, SHELF_R_Y),
        (BAR_W, fill_y), (0, fill_y),
    ]

```

```

    (0, SHELF_L_Y), (TAPER_L, SHELF_L_Y),
]

def soft_mask_shape(pct):
    """Soft anti-alias mask expanded lpx into fill."""
    o = 1
    fill_y = max(int(BAR_H * (100 - pct) / 100), 1)
    if fill_y <= SHELF_R_Y:
        frac = fill_y / SHELF_R_Y
        ml = int(PEAK_L + (TAPER_L - PEAK_L) * frac)
        mr = int(PEAK_R + (TAPER_R - PEAK_R) * frac)
        fy = min(fill_y + o, BAR_H)
        return [
            (PEAK_L, 0), (PEAK_R, 0),
            (mr + o, fy), (mr + o, fy),
            (mr + o, fy), (ml - o, fy),
            (ml - o, fy), (ml - o, fy),
        ]
    fy = min(fill_y + o, BAR_H)
    return [
        (PEAK_L, 0), (PEAK_R, 0),
        (TAPER_R + o, SHELF_R_Y),
        (BAR_W + o, SHELF_R_Y),
        (BAR_W + o, fy), (0, fy),
        (0, SHELF_L_Y), (TAPER_L - o, SHELF_L_Y),
    ]

# =====
# GRADIENT BACKGROUND
# =====
GRAD_STEPS = 64

def lerp_color(c1, c2, t):
    """Linearly interpolate two 24-bit colours."""
    r1 = (c1 >> 16) & 0xFF
    g1 = (c1 >> 8) & 0xFF
    b1 = c1 & 0xFF
    r2 = (c2 >> 16) & 0xFF
    g2 = (c2 >> 8) & 0xFF
    b2 = c2 & 0xFF
    r = int(r1 + (r2 - r1) * t)
    g = int(g1 + (g2 - g1) * t)
    b = int(b1 + (b2 - b1) * t)
    return (r << 16) | (g << 8) | b

def build_gradient():
    """Build a 1-row gradient bitmap."""
    bmp = displayio.Bitmap(
        TOTAL_BAR_W, 1, GRAD_STEPS
    )
    pal = displayio.Palette(GRAD_STEPS)
    centres = []
    for i in range(NUM_BARS):
        centres.append(
            i * (BAR_W + BAR_GAP) + BAR_W // 2
        )
    for i in range(GRAD_STEPS):
        t = i / (GRAD_STEPS - 1)
        pal[i] = _grad_at(t, centres)
    for x_pos in range(TOTAL_BAR_W):
        t = x_pos / (TOTAL_BAR_W - 1)
        idx = int(t * (GRAD_STEPS - 1))
        bmp[x_pos, 0] = idx
    return bmp, pal

```

```

def _grad_at(t, centres):
    """Gradient colour at position t."""
    x_pos = t * (TOTAL_BAR_W - 1)
    for i in range(NUM_BARS - 1):
        if x_pos <= centres[i + 1]:
            span = centres[i + 1] - centres[i]
            local_t = (x_pos - centres[i]) / span
            local_t = max(0.0, min(1.0, local_t))
            return lerp_color(
                GRAD_COLORS[i], GRAD_COLORS[i + 1],
                local_t
            )
    return GRAD_COLORS[-1]

# =====
# SLEEP MEMORY - persist CO2 offset across deep sleep
# =====
# Bytes 0-3: CO2 offset as signed int (big-endian)
co2_offset = 0
try:
    raw = bytes(alarm.sleep_memory[0:4])
    co2_offset = struct.unpack('>i', raw)[0]
    if abs(co2_offset) > 1000:
        co2_offset = 0
    if co2_offset != 0:
        print(f"Restored CO2 offset: {co2_offset}")
except (IndexError, ValueError, TypeError):
    co2_offset = 0

def save_co2_offset():
    """Write co2_offset to alarm.sleep_memory."""
    try:
        packed = struct.pack('>i', co2_offset)
        for i in range(4):
            alarm.sleep_memory[i] = packed[i]
    except (IndexError, TypeError):
        pass

# =====
# DISPLAY INIT
# =====
display = board.DISPLAY
display.brightness = FULL_BRIGHTNESS
is_dimmed = False
root = displayio.Group()

bg_pal = displayio.Palette(1)
bg_pal[0] = BG_COLOR
root.append(
    vectorio.Rectangle(
        pixel_shader=bg_pal,
        width=DISPLAY_W,
        height=DISPLAY_H,
        x=0, y=0,
    )
)

status_lbl = label.Label(
    terminalio.FONT,
    text="SCANNING I2C...",
    color=0xFFFFFF,
    anchor_point=(0.5, 0.5),
    anchored_position=(
        DISPLAY_W // 2, DISPLAY_H // 2
    ),
)
)

```

```

root.append(status_lbl)
display.root_group = root

# =====
#  BUTTON SETUP
# =====
btn_d0 = digitalio.DigitalInOut(board.D0)
btn_d0.direction = digitalio.Direction.INPUT
btn_d0.pull = digitalio.Pull.UP

btn_d1 = digitalio.DigitalInOut(board.D1)
btn_d1.direction = digitalio.Direction.INPUT
btn_d1.pull = digitalio.Pull.DOWN

btn_d2 = digitalio.DigitalInOut(board.D2)
btn_d2.direction = digitalio.Direction.INPUT
btn_d2.pull = digitalio.Pull.DOWN

# =====
#  I2C BUS + SENSORS
# =====
i2c = board.STEMMA_I2C()

sensor = adafruit_stcc4.STCC4(i2c)
sensor.continuous_measurement = True

batt = adafruit_max1704x.MAX17048(i2c)

haptic = adafruit_drv2605.DRV2605(i2c)
drv_effect = adafruit_drv2605.Effect

# =====
#  BUILD BAR GRAPH UI
# =====
root.pop() # remove status label

# Gradient background
grad_bmp, grad_pal = build_gradient()
grad_grid = displayio.TileGrid(
    grad_bmp,
    pixel_shader=grad_pal,
    width=1,
    height=BAR_H,
    tile_width=TOTAL_BAR_W,
    tile_height=1,
    x=X_OFFSET,
    y=BAR_Y,
)
root.append(grad_grid)

bar_masks = []
soft_masks = []
pct_labels = []

for bar_idx in range(NUM_BARS):
    bar_x = X_OFFSET + bar_idx * (BAR_W + BAR_GAP)

    out_pal = displayio.Palette(1)
    out_pal[0] = OUTLINE_COLORS[bar_idx]
    root.append(
        vectorio.Polygon(
            pixel_shader=out_pal,
            points=outline_shape(),
            x=bar_x, y=BAR_Y,
        )
    )
)

```

```

trk_pal = displayio.Palette(1)
trk_pal[0] = TRACK_COLORS[bar_idx]
root.append(
    vectorio.Polygon(
        pixel_shader=trk_pal,
        points=bar_shape(),
        x=bar_x, y=BAR_Y,
    )
)

fill_pal = displayio.Palette(1)
fill_pal[0] = BAR_COLORS[bar_idx]
root.append(
    vectorio.Polygon(
        pixel_shader=fill_pal,
        points=bar_shape(),
        x=bar_x, y=BAR_Y,
    )
)

soft_pal = displayio.Palette(1)
soft_pal[0] = SOFT_COLORS[bar_idx]
soft = vectorio.Polygon(
    pixel_shader=soft_pal,
    points=bar_shape(),
    x=bar_x, y=BAR_Y,
)
root.append(soft)
soft_masks.append(soft)

mask = vectorio.Polygon(
    pixel_shader=trk_pal,
    points=bar_shape(),
    x=bar_x, y=BAR_Y,
)
root.append(mask)
bar_masks.append(mask)

lbl = label.Label(
    terminalio.FONT,
    text="---",
    color=BAR_COLORS[bar_idx],
    anchor_point=(0.5, 0.0),
    anchored_position=(
        bar_x + BAR_W // 2, LABEL_Y
    ),
)
root.append(lbl)
pct_labels.append(lbl)

# =====
# CHARGING INDICATOR
# =====
BOLT_POINTS = [
    (4, 0), (7, 0), (3, 5),
    (6, 5), (1, 12), (4, 6), (1, 6),
]
BATT_BAR_X = X_OFFSET + 3 * (BAR_W + BAR_GAP)
BATT_BASE_MID = BAR_Y + SHELF_R_Y + (
    BAR_H - SHELF_R_Y
) // 2
BOLT_X = BATT_BAR_X + 3
BOLT_Y = BATT_BASE_MID - 6 # center 12px bolt

bolt_pal = displayio.Palette(1)
bolt_pal[0] = 0xFFFFFF
bolt_shape_obj = vectorio.Polygon(
    pixel_shader=bolt_pal,
    points=BOLT_POINTS,

```

```

    x=BOLT_X, y=BOLT_Y,
)
bolt_visible = False
is_charging = False
charge_blink = False

# In-bar voltage label (right of bolt)
volt_lbl = label.Label(
    terminalio.FONT,
    text="",
    color=0xFFFFF,
    anchor_point=(0.5, 0.5),
    anchored_position=(
        BATT_BAR_X + 28, BATT_BASE_MID
    ),
)
volt_visible = False

# =====
# STATE
# =====
last_haptic = 0.0
use_fahrenheit = True
co2_min = 9999
co2_max = 0
last_activity = time.monotonic()

SCAN_STEPS = 20
SCAN_DELAY = 0.03
SCAN_STAGGER = 3

# CO2 history for LCARS graph
HISTORY_SIZE = 120 # 2 hours of data
SAMPLE_INTERVAL = 60 # seconds between samples
co2_history = []
hum_history = []
temp_history = []
last_sample = 0.0

# View modes
VIEW_BARS = 0
VIEW_STATS = 1
VIEW_CO2 = 2
VIEW_HUM = 3
VIEW_TEMP = 4
VIEW_ABOUT = 5
NUM_CYCLE_VIEWS = 5 # bars->stats->co2->hum->temp
current_view = VIEW_BARS

# Graph layout constants
GRAPH_X = 56 # left edge of graph area
GRAPH_Y = 10
GRAPH_W = 180 # pixels wide
GRAPH_H = 100 # pixels tall
GRAPH_MIN_BAR = 1 # minimum bar width
GRAPH_MAX_BAR = 12 # maximum bar width (few samples)
GRAPH_GAP_PX = 1 # gap between bars
GRAPH_CO2_LO = 300 # CO2 graph floor (below 400)
GRAPH_CO2_HI = 1200 # CO2 graph ceiling
GRAPH_HUM_LO = 0 # humidity floor
GRAPH_HUM_HI = 100 # humidity ceiling
GRAPH_TEMP_LO = 50 # temp floor (F)
GRAPH_TEMP_HI = 100 # temp ceiling (F)
GRAPH_TEMP_LO_C = 10 # temp floor (C)
GRAPH_TEMP_HI_C = 38 # temp ceiling (C)

def graph_layout(n_samples, zoomed=False):
    """Compute bar width, slot, offset for n samples.

```

```

Normal: starts at half width, grows to full.
Zoomed: always fills full width with last
ZOOM_SAMPLES readings.
Returns (bar_px, slot, x_start, count)."""
if zoomed:
    count = min(n_samples, ZOOM_SAMPLES)
else:
    count = n_samples

if count <= 0:
    return GRAPH_MAX_BAR, GRAPH_MAX_BAR + 1, 0, 0

if zoomed:
    avail_w = GRAPH_W
else:
    fill_frac = min(
        0.5 + 0.5 * count / HISTORY_SIZE, 1.0
    )
    avail_w = int(GRAPH_W * fill_frac)

bar_px = max(
    (avail_w - (count - 1) * GRAPH_GAP_PX)
    // count,
    GRAPH_MIN_BAR,
)
bar_px = min(bar_px, GRAPH_MAX_BAR)
slot = bar_px + GRAPH_GAP_PX
total_w = count * slot - GRAPH_GAP_PX

x_start = (GRAPH_W - total_w) // 2
x_start = max(x_start, 0)

return bar_px, slot, x_start, count

# LCARS colors
LCARS_GOLD = 0xFFCC66
LCARS_BLUE = 0x6688CC
LCARS_ORANGE = 0xFF9944
LCARS_GREEN = 0x44CC66
LCARS_YELLOW = 0xCCCC44
LCARS_RED = 0xCC4444

# Min/max for all sensors
hum_min = 999
hum_max = 0
temp_min_val = 999.0
temp_max_val = -999.0

# Consolidated graph storage {view_mode: (group, bmp, val_lbl, mm_lbl)}
sensor_graphs = {}

# Stats display group (built on first show)
stats_group = None
stats_uptime_lbl = None
stats_samples_lbl = None
stats_co2_lbl = None
stats_hum_lbl = None
stats_temp_lbl = None
stats_batt_lbl = None

# About display group (built on first show)
about_group = None
boot_time = time.monotonic()

# Graph zoom
ZOOM_SAMPLES = 15      # 15 minutes of data when zoomed
graph_zoomed = False

```

```

# =====
# HELPERS
# =====
def clamp(val, lo, hi):
    """Clamp a value between lo and hi."""
    return max(lo, min(hi, val))

def to_pct(val, lo, hi):
    """Map val from [lo..hi] to 0-100."""
    return int(
        clamp((val - lo) / (hi - lo) * 100, 0, 100)
    )

def update_bar(idx, pct, txt):
    """Set bar fill height and label text."""
    soft_masks[idx].points = soft_mask_shape(pct)
    bar_masks[idx].points = mask_shape(pct)
    pct_labels[idx].text = txt

def scan_buzz():
    """Three quick haptic pulses for scan events."""
    for pulse in range(3): # pylint: disable=unused-variable
        haptic.sequence[0] = drv_effect(47)
        haptic.play()
        time.sleep(0.15)
        haptic.stop()
        if pulse < 2:
            time.sleep(0.1)

def buzz_alert(force=False):
    """Double-pulse haptic warning."""
    global last_haptic
    now = time.monotonic()
    if not force:
        if now - last_haptic < HAPTIC_COOLDOWN:
            return
    last_haptic = now
    for pulse in range(2): # pylint: disable=unused-variable
        haptic.sequence[0] = drv_effect(47)
        haptic.play()
        time.sleep(0.3)
        haptic.stop()
        if pulse == 0:
            time.sleep(0.15)

def scan_animation(targets):
    """Tricorder-style sweep with staggered text."""
    scan_buzz()
    for i in range(NUM_BARS):
        pct_labels[i].text = ""
        time.sleep(0.08)
    total = SCAN_STEPS + SCAN_STAGGER * 3
    for frame in range(total):
        for i in range(NUM_BARS):
            local = frame - i * SCAN_STAGGER
            if local < 0:
                pct = 0
            elif local >= SCAN_STEPS:
                pct = 100
            else:
                pct = int(local * 100 / SCAN_STEPS)
            soft_masks[i].points = soft_mask_shape(
                pct
            )

```

```

    )
    bar_masks[i].points = mask_shape(pct)
    time.sleep(SCAN_DELAY)
time.sleep(0.3)
for frame in range(total):
    for i in range(NUM_BARS):
        local = frame - i * SCAN_STAGGER
        t_pct, t_txt = targets[i]
        if local < 0:
            pct = 100
        elif local >= SCAN_STEPS:
            pct = t_pct
        else:
            frac = local / SCAN_STEPS
            pct = int(
                100 + (t_pct - 100) * frac
            )
        soft_masks[i].points = soft_mask_shape(
            pct
        )
        bar_masks[i].points = mask_shape(pct)
        if local >= SCAN_STEPS:
            pct_labels[i].text = t_txt
    time.sleep(SCAN_DELAY)

def rescan_animation(current, targets):
    """Rescan: current -> 100% -> 0% -> targets.

    Labels stay on screen throughout. current is a list
    of current pct values, targets is (pct, txt) pairs.
    """
    scan_buzz()

    # Capture starting percentages
    starts = list(current)

    # Phase 1: fill from current to 100% (no stagger)
    for frame in range(SCAN_STEPS):
        frac = frame / SCAN_STEPS
        for i in range(NUM_BARS):
            pct = int(starts[i] + (
                100 - starts[i]) * frac)
            soft_masks[i].points = soft_mask_shape(
                pct
            )
            bar_masks[i].points = mask_shape(pct)
        time.sleep(SCAN_DELAY)

    # Brief hold at peak
    for i in range(NUM_BARS):
        soft_masks[i].points = soft_mask_shape(100)
        bar_masks[i].points = mask_shape(100)
    time.sleep(0.15)

    # Phase 2: drain 100% -> 0% (no stagger)
    for frame in range(SCAN_STEPS):
        frac = frame / SCAN_STEPS
        pct = int(100 - 100 * frac)
        for i in range(NUM_BARS):
            soft_masks[i].points = soft_mask_shape(
                pct
            )
            bar_masks[i].points = mask_shape(pct)
        time.sleep(SCAN_DELAY)

    time.sleep(0.15)

    # Phase 3: fill 0% -> targets with stagger

```

```

total = SCAN_STEPS + SCAN_STAGGER * 3
for frame in range(total):
    for i in range(NUM_BARS):
        local = frame - i * SCAN_STAGGER
        t_pct, t_txt = targets[i]
        if local < 0:
            pct = 0
        elif local >= SCAN_STEPS:
            pct = t_pct
        else:
            frac = local / SCAN_STEPS
            pct = int(t_pct * frac)
        soft_masks[i].points = soft_mask_shape(
            pct
        )
        bar_masks[i].points = mask_shape(pct)
        if local >= SCAN_STEPS:
            pct_labels[i].text = t_txt
    time.sleep(SCAN_DELAY)

def solo_scan(idx, current_pct, target_pct, target_txt):
    """Rescan a single bar: current -> 100% -> 0% -> target.

    Label stays visible throughout."""
    scan_buzz()
    start = current_pct

    # Phase 1: fill from current to 100%
    for frame in range(SCAN_STEPS):
        frac = frame / SCAN_STEPS
        pct = int(start + (100 - start) * frac)
        soft_masks[idx].points = soft_mask_shape(pct)
        bar_masks[idx].points = mask_shape(pct)
        time.sleep(SCAN_DELAY)

    soft_masks[idx].points = soft_mask_shape(100)
    bar_masks[idx].points = mask_shape(100)
    time.sleep(0.15)

    # Phase 2: drain 100% to 0%
    for frame in range(SCAN_STEPS):
        frac = frame / SCAN_STEPS
        pct = int(100 - 100 * frac)
        soft_masks[idx].points = soft_mask_shape(pct)
        bar_masks[idx].points = mask_shape(pct)
        time.sleep(SCAN_DELAY)

    time.sleep(0.15)

    # Phase 3: fill 0% to target
    for frame in range(SCAN_STEPS):
        frac = frame / SCAN_STEPS
        pct = int(target_pct * frac)
        soft_masks[idx].points = soft_mask_shape(pct)
        bar_masks[idx].points = mask_shape(pct)
        time.sleep(SCAN_DELAY)

    update_bar(idx, target_pct, target_txt)

def show_message(txt, duration=2.0):
    """Overlay a message with dark background bar."""
    msg_bg_pal = displayio.Palette(1)
    msg_bg_pal[0] = 0x000000
    bg_bar = vectorio.Rectangle(
        pixel_shader=msg_bg_pal,
        width=DISPLAY_W,
        height=24,

```

```

        x=0,
        y=DISPLAY_H // 2 - 12,
    )
    msg = label.Label(
        terminalio.FONT,
        text=txt,
        color=0xFFFFFFFF,
        anchor_point=(0.5, 0.5),
        anchored_position=(
            DISPLAY_W // 2, DISPLAY_H // 2
        ),
    )
    root.append(bg_bar)
    root.append(msg)
    time.sleep(duration)
    root.remove(msg)
    root.remove(bg_bar)

def do_co2_calibrate():
    """Software CO2 offset calibration."""
    global co2_offset
    if sensor is None:
        show_message("NO SENSOR")
        return
    show_message("CALIBRATING...")
    try:
        raw = sensor.CO2
    except (RuntimeError, OSError):
        show_message("READ ERROR")
        return
    co2_offset = 400 - int(raw)
    save_co2_offset()
    show_message(
        f"OFFSET: {co2_offset}", 2.0
    )
    print(f"CO2 offset set to {co2_offset}")

# =====
# GRAPH ZONE DEFINITIONS
# =====
CO2_ZONES = [
    (600, 1), # green below 600
    (800, 2), # yellow 600-800
    (800, 3), # red above 800
]
HUM_ZONES = [
    (30, 1), # green below 30
    (60, 2), # yellow 30-60
    (80, 3), # red above 80
]
TEMP_ZONES_F = [
    (65, 1), # green below 65F
    (80, 2), # yellow 65-80F
    (90, 3), # red above 90F
]
TEMP_ZONES_C = [
    (18, 1), # green below 18C
    (27, 2), # yellow 18-27C
    (32, 3), # red above 32C
]

def _build_lcars_graph(title, unit, lo, hi, # pylint: disable=too-many-locals
    frame_color, zones):
    """Generic LCARS graph builder.

    zones is a list of (threshold, palette_index)

```

```

tuples sorted ascending. Returns
(group, bmp, val_lbl, mm_lbl)."""
grp = displayio.Group()

# Black background
p0 = displayio.Palette(1)
p0[0] = BG_COLOR
grp.append(vectorio.Rectangle(
    pixel_shader=p0,
    width=DISPLAY_W, height=DISPLAY_H,
    x=0, y=0,
))

# LCARS left frame
fp = displayio.Palette(1)
fp[0] = frame_color
grp.append(vectorio.Rectangle(
    pixel_shader=fp, width=50, height=8,
    x=0, y=0,
))
grp.append(vectorio.Rectangle(
    pixel_shader=fp, width=8, height=DISPLAY_H,
    x=0, y=0,
))
grp.append(vectorio.Rectangle(
    pixel_shader=fp, width=50, height=8,
    x=0, y=DISPLAY_H - 8,
))

# Accent blocks
ap1 = displayio.Palette(1)
ap1[0] = LCARS_BLUE
grp.append(vectorio.Rectangle(
    pixel_shader=ap1, width=42, height=6,
    x=8, y=10,
))
ap2 = displayio.Palette(1)
ap2[0] = LCARS_ORANGE
grp.append(vectorio.Rectangle(
    pixel_shader=ap2, width=42, height=6,
    x=8, y=DISPLAY_H - 16,
))

# Title + unit
grp.append(label.Label(
    terminalio.FONT, text=title,
    color=frame_color,
    anchor_point=(0.5, 0.0),
    anchored_position=(30, 22),
))

val_lbl = label.Label(
    terminalio.FONT, text="---",
    color=frame_color,
    anchor_point=(0.5, 0.0),
    anchored_position=(30, 34),
)
grp.append(val_lbl)

grp.append(label.Label(
    terminalio.FONT, text=unit,
    color=LCARS_BLUE,
    anchor_point=(0.5, 0.0),
    anchored_position=(30, 46),
))

mm_lbl = label.Label(
    terminalio.FONT, text="",
    color=LCARS_ORANGE,

```

```

        anchor_point=(0.5, 0.0),
        anchored_position=(30, 60),
    )
    grp.append(mm_lbl)

    grp.append(label.Label(
        terminalio.FONT, text="2HR",
        color=frame_color,
        anchor_point=(0.5, 0.0),
        anchored_position=(30, 74),
    ))

    # Bitmap with zone palette
    n_colors = len(zones) + 1
    pal = displayio.Palette(n_colors)
    pal[0] = BG_COLOR
    zone_colors = [LCARS_GREEN, LCARS_YELLOW, LCARS_RED]
    for idx in range(len(zones)):
        pal[idx + 1] = zone_colors[
            min(idx, len(zone_colors) - 1)
        ]

    bmp = displayio.Bitmap(GRAPH_W, GRAPH_H, n_colors)
    grp.append(displayio.TileGrid(
        bmp, pixel_shader=pal,
        x=GRAPH_X, y=GRAPH_Y,
    ))

    # Zone threshold lines
    rng = hi - lo
    for thresh, _ in zones[:-1]:
        yt = GRAPH_Y + GRAPH_H - int(
            (thresh - lo) / rng * GRAPH_H
        )
        zp = displayio.Palette(1)
        zp[0] = 0x444444
        for dx in range(0, GRAPH_W, 6):
            if dx + 2 <= GRAPH_W:
                grp.append(vectorio.Rectangle(
                    pixel_shader=zp,
                    width=2, height=1,
                    x=GRAPH_X + dx, y=yt,
                ))

    # Axis labels
    grp.append(label.Label(
        terminalio.FONT, text=f"{int(lo)}",
        color=0x666666,
        anchor_point=(0.0, 1.0),
        anchored_position=(
            GRAPH_X, GRAPH_Y + GRAPH_H + 10
        ),
    ))
    grp.append(label.Label(
        terminalio.FONT, text=f"{int(hi)}",
        color=0x666666,
        anchor_point=(1.0, 0.0),
        anchored_position=(
            GRAPH_X + GRAPH_W, GRAPH_Y - 2
        ),
    ))

    return grp, bmp, val_lbl, mm_lbl

```

```

def _update_lcars_graph(bmp, history, lo, hi, # pylint: disable=too-many-
    locals, too-many-branches
                        zones, val_lbl, mm_lbl,
                        mn, mx, fmt_fn):

```

```

"""Generic graph bitmap updater with dynamic bars."""
if bmp is None:
    return
for col in range(GRAPH_W):
    for row in range(GRAPH_H):
        bmp[col, row] = 0

num = len(history)
bar_px, slot, x_start, count = graph_layout(
    num, graph_zoomed
)
data = history[-count:] if count > 0 else []
rng = hi - lo
for idx, val in enumerate(data):
    clamped = max(lo, min(hi, val))
    bar_h = max(
        int((clamped - lo) / rng * GRAPH_H), 1
    )
    cidx = 1
    for thresh, zone_idx in zones:
        if clamped > thresh:
            cidx = zone_idx
    bar_x = x_start + idx * slot
    for px in range(bar_px):
        col = bar_x + px
        if col >= GRAPH_W:
            break
        for y_off in range(bar_h):
            row = GRAPH_H - 1 - y_off
            bmp[col, row] = cidx

if val_lbl is not None:
    if history:
        val_lbl.text = fmt_fn(history[-1])
    else:
        val_lbl.text = "---"
if mm_lbl is not None:
    mm_lbl.text = f"{fmt_fn(mn)}-{fmt_fn(mx)}"

def create_sensor_graph(view_mode):
    """Build and store an LCARS graph for a sensor."""
    if view_mode == VIEW_CO2:
        grp, bmp, vlbl, mlbl = _build_lcars_graph(
            "CO2", "PPM",
            GRAPH_CO2_LO, GRAPH_CO2_HI,
            LCARS_GOLD, CO2_ZONES,
        )
    elif view_mode == VIEW_HUM:
        grp, bmp, vlbl, mlbl = _build_lcars_graph(
            "HUM", "%RH",
            GRAPH_HUM_LO, GRAPH_HUM_HI,
            LCARS_GREEN, HUM_ZONES,
        )
    elif view_mode == VIEW_TEMP:
        if use_fahrenheit:
            lo, hi = GRAPH_TEMP_LO, GRAPH_TEMP_HI
            zones = TEMP_ZONES_F
            unit = "DEG.F"
        else:
            lo, hi = GRAPH_TEMP_LO_C, GRAPH_TEMP_HI_C
            zones = TEMP_ZONES_C
            unit = "DEG.C"
        grp, bmp, vlbl, mlbl = _build_lcars_graph(
            "TEMP", unit, lo, hi,
            LCARS_YELLOW, zones,
        )
    else:
        return

```

```

sensor_graphs[view_mode] = (grp, bmp, vlbl, mlbl)

def update_sensor_graph(view_mode):
    """Update the graph bitmap for a sensor."""
    if view_mode not in sensor_graphs:
        return
    _, bmp, vlbl, mlbl = sensor_graphs[view_mode]

    if view_mode == VIEW_CO2:
        mn = co2_min if co2_min < 9999 else 0
        mx = co2_max if co2_max > 0 else 0
        _update_lcars_graph(
            bmp, co2_history,
            GRAPH_CO2_LO, GRAPH_CO2_HI,
            CO2_ZONES, vlbl, mlbl, mn, mx,
            lambda v: f"{int(v)}",
        )
    elif view_mode == VIEW_HUM:
        mn = hum_min if hum_min < 999 else 0
        mx = hum_max if hum_max > 0 else 0
        _update_lcars_graph(
            bmp, hum_history,
            GRAPH_HUM_LO, GRAPH_HUM_HI,
            HUM_ZONES, vlbl, mlbl, mn, mx,
            lambda v: f"{int(v)}",
        )
    elif view_mode == VIEW_TEMP:
        if use_fahrenheit:
            lo, hi = GRAPH_TEMP_LO, GRAPH_TEMP_HI
            zones = TEMP_ZONES_F
            hist = [t * 9.0 / 5.0 + 32
                    for t in temp_history]
            mn = temp_min_val * 9.0 / 5.0 + 32
            mx = temp_max_val * 9.0 / 5.0 + 32
        else:
            lo, hi = GRAPH_TEMP_LO_C, GRAPH_TEMP_HI_C
            zones = TEMP_ZONES_C
            hist = list(temp_history)
            mn = temp_min_val
            mx = temp_max_val
        if temp_min_val > 900:
            mn = 0
        if temp_max_val < -900:
            mx = 0
        _update_lcars_graph(
            bmp, hist, lo, hi,
            zones, vlbl, mlbl, mn, mx,
            lambda v: f"{int(v)}",
        )

def build_stats():
    """Build the LCARS stats display group."""
    global stats_group
    global stats_uptime_lbl, stats_samples_lbl
    global stats_co2_lbl, stats_hum_lbl
    global stats_temp_lbl, stats_batt_lbl

    stats_group = displayio.Group()

    # Black background
    sbg = displayio.Palette(1)
    sbg[0] = BG_COLOR
    stats_group.append(
        vectorio.Rectangle(
            pixel_shader=sbg,
            width=DISPLAY_W, height=DISPLAY_H,
            x=0, y=0,

```

```

    )
)

# LCARS left frame
sg = displayio.Palette(1)
sg[0] = LCARS_BLUE
stats_group.append(
    vectorio.Rectangle(
        pixel_shader=sg,
        width=8, height=DISPLAY_H,
        x=0, y=0,
    )
)
stats_group.append(
    vectorio.Rectangle(
        pixel_shader=sg,
        width=50, height=8,
        x=0, y=0,
    )
)
sb = displayio.Palette(1)
sb[0] = LCARS_ORANGE
stats_group.append(
    vectorio.Rectangle(
        pixel_shader=sb,
        width=50, height=8,
        x=0, y=DISPLAY_H - 8,
    )
)

# Title
stats_group.append(
    label.Label(
        terminalio.FONT,
        text="DIAGNOSTICS",
        color=LCARS_GOLD,
        anchor_point=(0.0, 0.0),
        anchored_position=(14, 14),
    )
)

# Stat labels (left column)
y_start = 32
y_step = 16
stat_names = [
    "UPTIME", "SAMPLES",
    "CO2", "HUM", "TEMP", "BATT"
]
for i, nm in enumerate(stat_names):
    stats_group.append(
        label.Label(
            terminalio.FONT,
            text=nm,
            color=LCARS_BLUE,
            anchor_point=(0.0, 0.0),
            anchored_position=(
                14, y_start + i * y_step
            ),
        )
    )

# Value labels (right column)
stats_uptime_lbl = label.Label(
    terminalio.FONT, text="0:00",
    color=LCARS_GOLD,
    anchor_point=(1.0, 0.0),
    anchored_position=(
        DISPLAY_W - 10, y_start
    ),
),

```

```

)
stats_group.append(stats_uptime_lbl)

stats_samples_lbl = label.Label(
    terminalio.FONT, text="0/120",
    color=LCARS_GOLD,
    anchor_point=(1.0, 0.0),
    anchored_position=(
        DISPLAY_W - 10, y_start + y_step
    ),
)
stats_group.append(stats_samples_lbl)

stats_co2_lbl = label.Label(
    terminalio.FONT, text="---",
    color=LCARS_GREEN,
    anchor_point=(1.0, 0.0),
    anchored_position=(
        DISPLAY_W - 10, y_start + 2 * y_step
    ),
)
stats_group.append(stats_co2_lbl)

stats_hum_lbl = label.Label(
    terminalio.FONT, text="---",
    color=LCARS_GREEN,
    anchor_point=(1.0, 0.0),
    anchored_position=(
        DISPLAY_W - 10, y_start + 3 * y_step
    ),
)
stats_group.append(stats_hum_lbl)

stats_temp_lbl = label.Label(
    terminalio.FONT, text="---",
    color=LCARS_YELLOW,
    anchor_point=(1.0, 0.0),
    anchored_position=(
        DISPLAY_W - 10, y_start + 4 * y_step
    ),
)
stats_group.append(stats_temp_lbl)

stats_batt_lbl = label.Label(
    terminalio.FONT, text="---",
    color=LCARS_RED,
    anchor_point=(1.0, 0.0),
    anchored_position=(
        DISPLAY_W - 10, y_start + 5 * y_step
    ),
)
stats_group.append(stats_batt_lbl)

def update_stats(dis):
    """Update stats screen labels."""
    if stats_uptime_lbl is None:
        return
    elapsed = time.monotonic() - boot_time
    hrs = int(elapsed // 3600)
    mins = int((elapsed % 3600) // 60)
    stats_uptime_lbl.text = f"{hrs}:{mins:02d}"
    stats_samples_lbl.text = (
        f"{len(co2_history)}/{HISTORY_SIZE}"
    )

    co2_r = disp[8]
    co2_lo = int(co2_min) if co2_min < 9999 else "---"
    co2_hi = int(co2_max) if co2_max > 0 else "---"

```

```

stats_co2_lbl.text = (
    f"{int(co2_r)} {co2_lo}-{co2_hi}"
)
hlo = int(hum_min) if hum_min < 999 else "----"
hhi = int(hum_max) if hum_max > 0 else "----"
stats_hum_lbl.text = f"{disp[5]} {hlo}-{hhi}"
if use_fahrenheit:
    tlo = int(
        temp_min_val * 9.0 / 5.0 + 32
    ) if temp_min_val < 900 else "----"
    thi = int(
        temp_max_val * 9.0 / 5.0 + 32
    ) if temp_max_val > -900 else "----"
else:
    tlo = int(
        temp_min_val
    ) if temp_min_val < 900 else "----"
    thi = int(
        temp_max_val
    ) if temp_max_val > -900 else "----"
stats_temp_lbl.text = f"{disp[6]} {tlo}-{thi}"
stats_batt_lbl.text = disp[7]

def build_about(): # pylint: disable=too-many-locals,too-many-statements
    """Build the LCARS name badge display group."""
    global about_group

    about_group = displayio.Group()

    # Black background
    abg = displayio.Palette(1)
    abg[0] = BG_COLOR
    about_group.append(
        vectorio.Rectangle(
            pixel_shader=abg,
            width=DISPLAY_W, height=DISPLAY_H,
            x=0, y=0,
        )
    )

    # LCARS frame - left + top
    ag = displayio.Palette(1)
    ag[0] = LCARS_GOLD
    about_group.append(
        vectorio.Rectangle(
            pixel_shader=ag,
            width=8, height=DISPLAY_H,
            x=0, y=0,
        )
    )
    about_group.append(
        vectorio.Rectangle(
            pixel_shader=ag,
            width=DISPLAY_W, height=8,
            x=0, y=0,
        )
    )
    about_group.append(
        vectorio.Rectangle(
            pixel_shader=ag,
            width=DISPLAY_W, height=8,
            x=0, y=DISPLAY_H - 8,
        )
    )

    # Accent blocks
    abl = displayio.Palette(1)
    abl[0] = LCARS_BLUE

```

```

about_group.append(
    vectorio.Rectangle(
        pixel_shader=ab1,
        width=60, height=8,
        x=10, y=12,
    )
)
ab2 = displayio.Palette(1)
ab2[0] = LCARS_ORANGE
about_group.append(
    vectorio.Rectangle(
        pixel_shader=ab2,
        width=40, height=8,
        x=72, y=12,
    )
)
ab3 = displayio.Palette(1)
ab3[0] = LCARS_RED
about_group.append(
    vectorio.Rectangle(
        pixel_shader=ab3,
        width=30, height=8,
        x=114, y=12,
    )
)

# Name - large
about_group.append(
    label.Label(
        terminalio.FONT,
        text=BADGE_NAME,
        color=LCARS_GOLD,
        scale=2,
        anchor_point=(0.0, 0.0),
        anchored_position=(14, 28),
    )
)

# Title
about_group.append(
    label.Label(
        terminalio.FONT,
        text=BADGE_TITLE,
        color=LCARS_BLUE,
        anchor_point=(0.0, 0.0),
        anchored_position=(14, 52),
    )
)

# ID / registry
about_group.append(
    label.Label(
        terminalio.FONT,
        text=BADGE_ID,
        color=LCARS_ORANGE,
        anchor_point=(0.0, 0.0),
        anchored_position=(14, 68),
    )
)

# Sensor status
stcc_ok = "OK" if sensor else "N/A"
drv_ok = "OK" if haptic else "N/A"
s_status = f"STCC4: {stcc_ok} DRV: {drv_ok}"
about_group.append(
    label.Label(
        terminalio.FONT,
        text=s_status,
        color=0x666666,
    )
)

```

```

        anchor_point=(0.0, 0.0),
        anchored_position=(14, 88),
    )
)

# Project info
about_group.append(
    label.Label(
        terminalio.FONT,
        text=BADGE_PROJECT,
        color=0x666666,
        anchor_point=(0.0, 0.0),
        anchored_position=(14, 104),
    )
)

# QR code on right side
if BADGE_QR_URL:
    try:
import-position
        import adafruit_minqr # pylint: disable=import-outside-toplevel,wrong-
import-position
        qr = adafruit_minqr.QRCode(
            qr_type=3,
            error_correct=adafruit_minqr.L,
        )
        qr.add_data(
            BADGE_QR_URL.encode("utf-8")
        )
        qr.make()
        mtx = qr.matrix
        border = 1
        sz = mtx.width + border * 2
        # Scale to fit ~100px tall
        scale = max(100 // sz, 1)
        qr_bmp = displayio.Bitmap(sz, sz, 2)
        qr_pal = displayio.Palette(2)
        qr_pal[0] = 0xFFFFFF
        qr_pal[1] = 0x000000
        for y in range(mtx.height):
            for x in range(mtx.width):
                if mtx[x, y]:
                    qr_bmp[
                        x + border,
                        y + border
                    ] = 1
                else:
                    qr_bmp[
                        x + border,
                        y + border
                    ] = 0
        qr_grid = displayio.TileGrid(
            qr_bmp,
            pixel_shader=qr_pal,
            x=0, y=0,
        )
        qr_total = sz * scale
        qr_group = displayio.Group(
            scale=scale,
            x=DISPLAY_W - qr_total - 8,
            y=(DISPLAY_H - qr_total) // 2,
        )
        qr_group.append(qr_grid)
        about_group.append(qr_group)
        print(f"QR code generated: {BADGE_QR_URL}")
    except (ImportError, MemoryError) as err:
        print(f"QR skipped: {err}")

def set_view(mode):

```

```

"""Switch to the specified view mode."""
global current_view, graph_zoomed
current_view = mode
# Reset zoom when leaving graph views
if mode not in (VIEW_CO2, VIEW_HUM, VIEW_TEMP):
    graph_zoomed = False
if mode == VIEW_BARS:
    display.root_group = root
elif mode == VIEW_STATS:
    if stats_group is None:
        build_stats()
    display.root_group = stats_group
elif mode in (VIEW_CO2, VIEW_HUM, VIEW_TEMP):
    if mode not in sensor_graphs:
        create_sensor_graph(mode)
    update_sensor_graph(mode)
    display.root_group = sensor_graphs[mode][0]
elif mode == VIEW_ABOUT:
    if about_group is None:
        build_about()
    display.root_group = about_group

def cycle_view():
    """Cycle: bars->stats->co2->hum->temp->bars."""
    nxt = (current_view + 1) % NUM_CYCLE_VIEWS
    set_view(nxt)

def toggle_brightness():
    """Toggle between dim and full brightness."""
    global is_dimmed
    is_dimmed = not is_dimmed
    if is_dimmed:
        display.brightness = DIM_BRIGHTNESS
    else:
        display.brightness = FULL_BRIGHTNESS
    bright_label = "dim" if is_dimmed else "full"
    print(f"Brightness: {bright_label}")

DEMO_HOLD = 3 # seconds per screen in demo mode

def demo_mode(): # pylint: disable=too-many-statements
    """Auto-cycle all screens for video recording."""
    global graph_zoomed
    print("Demo mode started")
    scan_buzz()

    # 1) Bar view with rescan animation
    set_view(VIEW_BARS)
    vals = read_sensors()
    disp = compute_display(*vals)
    co2p = disp[0]
    hump = disp[1]
    tmpp = disp[2]
    batp = disp[3]
    co2t = disp[4]
    humt = disp[5]
    tmpt = disp[6]
    bt = disp[7]
    current = [co2p, hump, tmpp, batp]
    targets = [
        (co2p, co2t), (hump, humt),
        (tmpp, tmpt), (batp, bt),
    ]
    rescan_animation(current, targets)
    update_bar(0, co2p, co2t)

```

```

update_bar(1, hump, humt)
update_bar(2, tmpp, tmpt)
update_bar(3, batp, bt)
time.sleep(DEMO_HOLD)

# 2) Stats screen
set_view(VIEW_STATS)
update_stats(dis)
time.sleep(DEMO_HOLD)

# 3) CO2 graph zoomed
graph_zoomed = True
set_view(VIEW_CO2)
time.sleep(DEMO_HOLD)

# 4) CO2 graph full
graph_zoomed = False
update_sensor_graph(VIEW_CO2)
time.sleep(DEMO_HOLD)

# 5) Humidity graph zoomed
graph_zoomed = True
set_view(VIEW_HUM)
time.sleep(DEMO_HOLD)

# 6) Humidity graph full
graph_zoomed = False
update_sensor_graph(VIEW_HUM)
time.sleep(DEMO_HOLD)

# 7) Temp graph zoomed
graph_zoomed = True
set_view(VIEW_TEMP)
time.sleep(DEMO_HOLD)

# 8) Temp graph full
graph_zoomed = False
update_sensor_graph(VIEW_TEMP)
time.sleep(DEMO_HOLD)

# 9) Badge / about
set_view(VIEW_ABOUT)
time.sleep(DEMO_HOLD)

# 10) Back to bars with scan
graph_zoomed = False
set_view(VIEW_BARS)
scan_animation(targets)
update_bar(0, co2p, co2t)
update_bar(1, hump, humt)
update_bar(2, tmpp, tmpt)
update_bar(3, batp, bt)
print("Demo mode complete")

def enter_sleep():
    """Enter low-power sleep. Any button press wakes."""
    global bolt_visible, volt_visible, last_activity
    print("Entering sleep...")
    show_message("SLEEP", 1.5)

# Remove charging indicators if visible
if bolt_visible:
    root.remove(bolt_shape_obj)
    bolt_visible = False
if volt_visible:
    root.remove(volt_lbl)
    volt_visible = False

```

```

# Save CO2 offset
save_co2_offset()

# Clear bars and labels
for i in range(NUM_BARS):
    update_bar(i, 0, "")

# Wait for ALL buttons to be released
while (not btn_d0.value) or btn_d1.value or btn_d2.value:
    time.sleep(0.05)
time.sleep(0.2)

# Blank display
display.brightness = 0

# Light sleep loop - wake every 2s to check buttons
while True:
    t_alarm = alarm.time.TimeAlarm(
        monotonic_time=time.monotonic() + 2
    )
    alarm.light_sleep_until_alarms(t_alarm)

    # Check if any button is pressed
    d0_wake = not btn_d0.value
    d1_wake = btn_d1.value
    d2_wake = btn_d2.value
    if d0_wake or d1_wake or d2_wake:
        break

# Wake up
display.brightness = (
    DIM_BRIGHTNESS if is_dimmed else
    FULL_BRIGHTNESS
)
last_activity = time.monotonic()

# Wait for button release then boot scan
while (not btn_d0.value) or btn_d1.value or btn_d2.value:
    time.sleep(0.05)
time.sleep(0.2)
print("Awake!")

# =====
# SENSOR READ HELPER
# =====
def read_sensors():
    """Read all sensors and return raw values."""
    global co2_min, co2_max
    global hum_min, hum_max
    global temp_min_val, temp_max_val
    try:
        co2_r = sensor.CO2 if sensor else 0
        co2_r = co2_r + co2_offset
        hum_r = (
            sensor.relative_humidity if sensor else 0
        )
        tmp_r = sensor.temperature if sensor else 0
    except (RuntimeError, OSError) as err:
        print(f"Sensor error: {err}")
        co2_r, hum_r, tmp_r = 0, 0, 0

    try:
        bat_r = batt.cell_percent if batt else 0
        bat_v = batt.cell_voltage if batt else 0
        bat_rate = batt.charge_rate if batt else 0
    except (RuntimeError, OSError) as err:
        print(f"Battery error: {err}")
        bat_r, bat_v, bat_rate = 0, 0, 0

```

```

# Track min/max for all sensors
if sensor is not None:
    if co2_r > 0:
        if co2_r < co2_min:
            co2_min = co2_r
        if co2_r > co2_max:
            co2_max = co2_r
    if hum_r > 0:
        if hum_r < hum_min:
            hum_min = hum_r
        if hum_r > hum_max:
            hum_max = hum_r
    if tmp_r != 0:
        if tmp_r < temp_min_val:
            temp_min_val = tmp_r
        if tmp_r > temp_max_val:
            temp_max_val = tmp_r

    return co2_r, hum_r, tmp_r, bat_r, bat_v, bat_rate

def compute_display(co2_r, hum_r, tmp_r, # pylint: disable=too-many-locals,too-
many-branches
                    bat_r, bat_v, bat_rate):
    """Compute percentages and label text."""
    global is_charging, bolt_visible, volt_visible

    co2p = max(to_pct(co2_r, CO2_MIN, CO2_MAX),
               CO2_FLOOR_PCT)
    hump = int(clamp(hum_r, 0, 100))
    tmpp = to_pct(tmp_r, TEMP_MIN, TEMP_MAX)
    batp = int(clamp(bat_r, 0, 100))

    co2t = f"{int(co2_r)}"
    humt = f"{hump}%"
    if use_fahrenheit:
        tv = tmp_r * 9.0 / 5.0 + 32
        tmpt = f"{int(tv)}F"
    else:
        tmpt = f"{int(tmp_r)}C"

    is_charging = bat_rate > 0.5
    is_discharging = bat_rate < -0.5
    batt_t = f"{batp}%" # always show % below

    # Compute time estimate
    bar_txt = ""
    if is_charging and bat_rate > 0.5:
        hrs = (100.0 - bat_r) / bat_rate
        if hrs < 1.0:
            bar_txt = f"{int(hrs * 60)}m"
        else:
            bar_txt = f"{hrs:.1f}h"
    elif is_discharging and abs(bat_rate) > 0.5:
        hrs = bat_r / abs(bat_rate)
        if hrs < 1.0:
            bar_txt = f"{int(hrs * 60)}m"
        else:
            bar_txt = f"{hrs:.1f}h"

    # Bolt: charging only
    if is_charging:
        if not bolt_visible:
            root.append(bolt_shape_obj)
            bolt_visible = True
    else:
        if bolt_visible:
            root.remove(bolt_shape_obj)

```

```

        bolt_visible = False

# In-bar label: show time for both states
if bar_txt:
    if is_charging:
        volt_lbl.color = 0xFFFFFF
        volt_lbl.anchored_position = (
            BATT_BAR_X + 28, BATT_BASE_MID
        )
    else:
        volt_lbl.color = 0xFFFFFF
        volt_lbl.anchored_position = (
            BATT_BAR_X + BAR_W // 2,
            BATT_BASE_MID
        )
    volt_lbl.text = bar_txt
    if not volt_visible:
        root.append(volt_lbl)
        volt_visible = True
    else:
        if volt_visible:
            root.remove(volt_lbl)
            volt_visible = False

    return (
        co2p, hump, tmp, batp,
        co2t, humt, tmp, batt_t,
        co2_r, tmp_r
    )

# =====
# MAIN LOOP
# =====
time.sleep(1) # sensor stabilization

# Wait for all buttons to be released before boot scan
# (prevents double-scan on wake from deep sleep)
while (not btn_d0.value) or btn_d1.value or btn_d2.value:
    time.sleep(0.05)
time.sleep(0.2)

# Boot scan
vals = read_sensors()
disp = compute_display(*vals)
co2p = disp[0]
hump = disp[1]
tmp = disp[2]
batp = disp[3]
co2t = disp[4]
humt = disp[5]
tmp = disp[6]
batt_t = disp[7]
scan_animation([
    (co2p, co2t), (hump, humt),
    (tmp, tmp), (batp, batt_t),
])

# First sensor sample
co2_history.append(disp[8])
hum_history.append(vals[1])
temp_history.append(vals[2])
last_sample = time.monotonic()

# Button state - read actual values to avoid ghost press
d0_prev = not btn_d0.value # active low, inverted
d1_prev = btn_d1.value
d2_prev = btn_d2.value
d0_down_at = 0

```

```

d1_down_at = 0
d2_down_at = 0
d0_long_fired = False
d1_long_fired = False
d2_long_fired = False
d0_pending = False      # single tap waiting for double
d0_release_at = 0
DOUBLE_TAP_MS = 400     # max gap for double tap
need_scan = False

while True:
    vals = read_sensors()
    disp = compute_display(*vals)
    co2p = disp[0]
    hump = disp[1]
    tmpp = disp[2]
    batp = disp[3]
    co2t = disp[4]
    humt = disp[5]
    tmpt = disp[6]
    batt_t = disp[7]
    co2_raw_adj = disp[8]
    temp_raw = disp[9]

    # --- Sample all sensors to history ---
    now_t = time.monotonic()
    if now_t - last_sample >= SAMPLE_INTERVAL:
        last_sample = now_t
        co2_history.append(co2_raw_adj)
        if len(co2_history) > HISTORY_SIZE:
            co2_history.pop(0)
        hum_history.append(vals[1])
        if len(hum_history) > HISTORY_SIZE:
            hum_history.pop(0)
        temp_history.append(vals[2])
        if len(temp_history) > HISTORY_SIZE:
            temp_history.pop(0)

    # --- Update display based on current view ---
    if current_view in (VIEW_CO2, VIEW_HUM, VIEW_TEMP):
        update_sensor_graph(current_view)
    elif current_view == VIEW_STATS:
        update_stats(disp)
    elif current_view == VIEW_BARS:
        # Wake-from-sleep scan
        if need_scan:
            need_scan = False
            targets = [
                (co2p, co2t), (hump, humt),
                (tmpp, tmpt), (batp, batt_t),
            ]
            scan_animation(targets)

            update_bar(0, co2p, co2t)
            update_bar(1, hump, humt)
            update_bar(2, tmpp, tmpt)
            update_bar(3, batp, batt_t)

        if co2_raw_adj >= CO2_WARN:
            buzz_alert()

    print(
        f"C02: {int(co2_raw_adj)}ppm | Hum: {hump}% | "
        f"Temp: {tmpt} | Batt: {batt_t}"
    )

    # --- Poll buttons at 50ms ---
    poll_end = time.monotonic() + UPDATE_INTERVAL
    blink_time = time.monotonic()

```

```

while time.monotonic() < poll_end:
    now = time.monotonic()

    # Bolt blink
    if is_charging and bolt_visible:
        if now - blink_time >= 0.5:
            charge_blink = not charge_blink
            if charge_blink:
                bolt_pal[0] = 0xFFFFFF
            else:
                bolt_pal[0] = TRACK_COLORS[3]
            blink_time = now

    # Idle sleep check
    if now - last_activity >= IDLE_SLEEP_S:
        enter_sleep()
        need_scan = True
        break # restart main loop with scan

    # Read buttons
    d0_now = not btn_d0.value # active low
    d1_now = btn_d1.value
    d2_now = btn_d2.value

    # --- D0 pending single-tap timeout ---
    if d0_pending:
        elapsed_ms = (now - d0_release_at) * 1000
        if elapsed_ms >= DOUBLE_TAP_MS:
            d0_pending = False
            # Fire delayed single-tap action
            if current_view == VIEW_BARS:
                current = [
                    co2p, hump, tmp, batp,
                ]
                targets = [
                    (co2p, co2t),
                    (hump, humt),
                    (tmp, tmpt),
                    (batp, batt_t),
                ]
                rescan_animation(
                    current, targets
                )
            else:
                set_view(VIEW_BARS)
                last_activity = now

    # --- D0 press tracking ---
    if d0_now and not d0_prev:
        # Check for doubletap
        if d0_pending:
            d0_pending = False
            d0_long_fired = True
            demo_mode()
            last_activity = now
        else:
            d0_down_at = now
            d0_long_fired = False
            last_activity = now
    if d0_now and not d0_long_fired:
        held = (now - d0_down_at) * 1000
        if held >= LONG_PRESS_MS:
            # Check combo D0+D2
            if d2_now:
                d0_long_fired = True
                d2_long_fired = True
                enter_sleep()
                need_scan = True
                break # restart main loop

```

```

        d0_long_fired = True
        do_co2_calibrate()
        last_activity = now
    if not d0_now and d0_prev:
        if not d0_long_fired:
            # Queue as pending, wait for
            # possible double tap
            d0_pending = True
            d0_release_at = now
    d0_prev = d0_now

# --- D1 press tracking ---
if d1_now and not d1_prev:
    d1_down_at = now
    d1_long_fired = False
    last_activity = now
if d1_now and not d1_long_fired:
    held = (now - d1_down_at) * 1000
    if held >= LONG_PRESS_MS:
        d1_long_fired = True
        if current_view == VIEW_ABOUT:
            set_view(VIEW_BARS)
        else:
            set_view(VIEW_ABOUT)
        last_activity = now
if not d1_now and d1_prev:
    if not d1_long_fired:
        if current_view in (
            VIEW_CO2, VIEW_HUM, VIEW_TEMP
        ):
            # Zoom toggle on graph views
            graph_zoomed = not graph_zoomed
            zoom_label = (
                "15min" if graph_zoomed
                else "full"
            )
            print(f"Zoom: {zoom_label}")
        else:
            # F/C toggle on other views
            use_fahrenheit = not use_fahrenheit
            if use_fahrenheit:
                tv = (
                    temp_raw * 9.0 / 5.0 + 32
                )
                tmpt = f"{int(tv)}F"
            else:
                tmpt = f"{int(temp_raw)}C"
            if current_view == VIEW_BARS:
                solo_scan(
                    2, tmpp, tmpp, tmpt
                )
        last_activity = now
d1_prev = d1_now

# --- D2 press tracking ---
if d2_now and not d2_prev:
    d2_down_at = now
    d2_long_fired = False
    last_activity = now
if d2_now and not d2_long_fired:
    held = (now - d2_down_at) * 1000
    if held >= LONG_PRESS_MS:
        d2_long_fired = True
        toggle_brightness()
        last_activity = now
if not d2_now and d2_prev:
    if not d2_long_fired:
        # Short press - cycle views
        cycle_view()

```

```
        last_activity = now
    d2_prev = d2_now

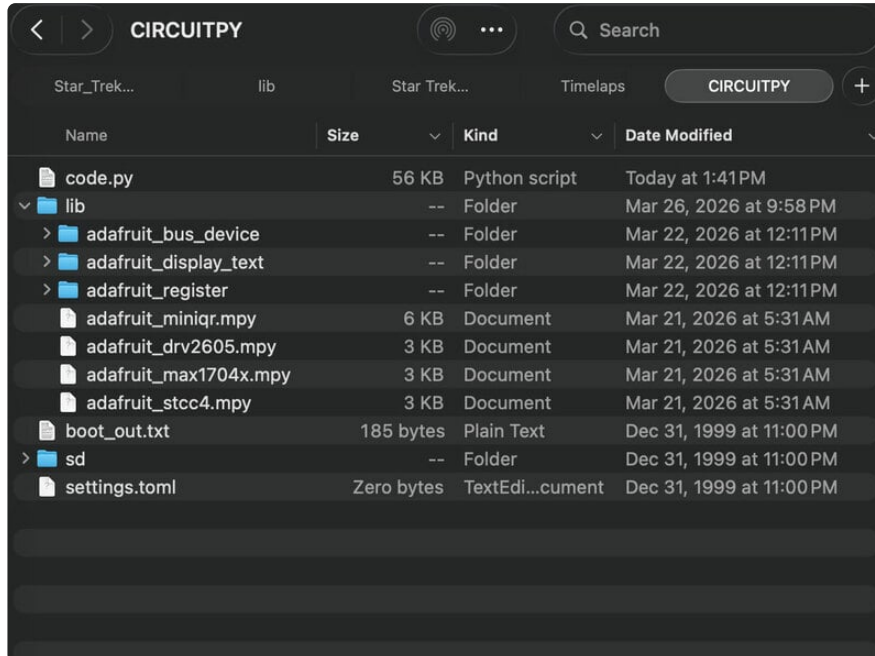
    time.sleep(0.05)
```

Required Libraries

Copy the **lib** folder from the zip file to your **CIRCUITPY** drive.

- /adafruit_display_text
- adafruit_stcc4.mpy
- adafruit_max1704x.mpy
- /adafruit_bus_device
- /adafruit_register
- adafruit_drv2605.mpy
- adafruit_miniq.r.mpy

When all the files are copied to **CIRCUITPY**, it should look like the file listing below. If you're missing any files, look in the zip file and copy them to the appropriate place.



The first block of constants at the top of **code.py** controls the sensor ranges, timing, and display behavior. These are the values you are most likely to want to edit.

```
C02_MIN = 400
C02_MAX = 2000
```

```
CO2_WARN = 700
CO2_FLOOR_PCT = 8
TEMP_MIN = 15.0
TEMP_MAX = 40.0
UPDATE_INTERVAL = 2
HAPTIC_COOLDOWN = 30
LONG_PRESS_MS = 2000
IDLE_SLEEP_S = 300
DIM_BRIGHTNESS = 0.15
FULL_BRIGHTNESS = 1.0
```

- **CO2_WARN** — the CO2 level in ppm that triggers the haptic alert. Office air is typically 400 to 600 ppm.
- **CO2_FLOOR_PCT** — minimum visible bar height as a percentage so the CO2 bar is never completely empty.
- **TEMP_MIN** and **TEMP_MAX** — the temperature range in Celsius that maps to 0 to 100 percent on the bar.
- **UPDATE_INTERVAL** — seconds between sensor reads and display updates.
- **HAPTIC_COOLDOWN** — minimum seconds between haptic alerts to avoid constant buzzing.
- **LONG_PRESS_MS** — milliseconds a button must be held to trigger a long press action.
- **IDLE_SLEEP_S** — seconds of no button activity before the device automatically sleeps.
- **DIM_BRIGHTNESS** — display brightness when dimmed via D2 long press. Range is 0.0 to 1.0.

Name Badge

The badge text is editable right below the tunables. These strings appear on the LCARS name badge screen and the QR code links to the URL you provide.

```
BADGE_NAME = "P. RUIZ"
BADGE_TITLE = "ENGINEERING"
BADGE_ID = "NCC-1701-D"
BADGE_PROJECT = "TNG DATA DISPENSER"
BADGE_QR_URL = "https://learn.adafruit.com/u/pixil3d"
```

- **BADGE_NAME** — your name, displayed at 2x scale on the badge screen.
- **BADGE_TITLE** — department or role, shown in LCARS blue.
- **BADGE_ID** — registry number or ID, shown in LCARS orange.
- **BADGE_PROJECT** — project title shown at the bottom of the badge.

- `BADGE_QR_URL` — the URL encoded into the QR code. Use a short URL for a smaller, more scannable code. Set to an empty string to hide the QR code entirely.

Colors and Anti-Aliasing

Each bar has a unique color — blue for CO2, green for humidity, yellow for temperature, red for battery. The code generates several dimmed variants of each color for the anti-aliasing layers.

```
COLOR_CO2 = 0x4466FF
COLOR_HUMIDITY = 0x44FF66
COLOR_TEMP = 0xFFCC00
COLOR_BATTERY = 0xFF4444
```

The `dim_color()` function divides each RGB channel by a factor to produce darker versions. These are used for the five-layer polygon stack that makes each bar look smooth on the small 240 by 135 display. From bottom to top the layers are: outline glow at 10 percent brightness, colored track at 20 percent, full color fill, soft transition mask at 25 percent and dark mask for the unfilled portion.

Bar Shape Geometry

The bar shape is an 8-point polygon that matches the tapered design from the original prop — narrow peak at the top, an asymmetric taper, and a wide base at the bottom. These coordinates were traced from an SVG of the prop's screen graphic.

```
PEAK_L = 11
PEAK_R = 13
TAPER_L = 2
TAPER_R = 19
SHELF_L_Y = 57
SHELF_R_Y = 56
```

The `bar_shape()` function returns the outline, `mask_shape(pct)` computes the unfilled region for a given percentage, and `soft_mask_shape(pct)` adds a 1-pixel transition zone between the fill and the mask. This creates a subtle anti-alias effect at the fill line that softens the hard edge.

Sensor Setup

All libraries are imported at the top of the file and sensors are initialized directly. The STEMMA QT I2C bus is created with a single call to `board.STEMMA_I2C()`. Each sensor is initialized without try/except wrapping — if a sensor is not found on the bus, the library raises an error that tells you to check your wiring.

```
i2c = board.STEMMA_I2C()

sensor = adafruit_stcc4.STCC4(i2c)
sensor.continuous_measurement = True

batt = adafruit_max1704x.MAX17048(i2c)

haptic = adafruit_drv2605.DRV2605(i2c)
drv_effect = adafruit_drv2605.Effect
```

The STCC4 provides CO2, temperature and humidity. The MAX17048 provides battery percentage, voltage and charge rate. The DRV2605 drives the haptic motor for CO2 warning alerts and scan animation feedback.

Charging Indicator and Battery Time

The MAX17048 fuel gauge provides `charge_rate` in percent per hour — positive when charging, negative when discharging. The code uses this to estimate time remaining.

```
if is_charging:
    hrs = (100.0 - bat_r) / bat_rate
elif is_discharging:
    hrs = bat_r / abs(bat_rate)
```

When charging, a white lightning bolt and time-to-full appear inside the wide base of the battery bar. When discharging, the time remaining appears centered in the base without the bolt. The percentage label always stays in its normal position below the bar. Time shows as hours with one decimal, or switches to minutes when under one hour.

Scan Animations

There are three animation styles. The boot scan sweeps all four bars from zero to 100 percent with a staggered cascade, then settles each bar to its sensor reading. The rescan animation starts from the current fill level, surges to 100 percent, drains to zero and then fills to the new reading. The solo scan does the same sequence on a single bar for the Fahrenheit to Celsius toggle.

```
def rescan_animation(current, targets):
    # Phase 1: fill from current to 100%
    # Phase 2: drain 100% to 0%
    # Phase 3: fill 0% to targets with stagger
```

All animations use `SCAN_STEPS` of 20 frames at `SCAN_DELAY` of 30 milliseconds each, with a `SCAN_STAGGER` of 3 frames between bars. A three-pulse haptic buzz plays at the start of every scan.

LCARS Graph System

Three LCARS-style bar histogram screens show two hours of CO2, humidity and temperature history. All three use the same pair of functions: `create_sensor_graph(view_mode)` builds the display group and `update_sensor_graph(view_mode)` redraws the bitmap. Each graph is stored in the `sensor_graphs` dictionary and built lazily on first access.

```
def create_sensor_graph(view_mode):
    if view_mode == VIEW_CO2:
        grp, bmp, vlbl, mlbl = _build_lcars_graph(
            "CO2", "PPM",
            GRAPH_CO2_LO, GRAPH_CO2_HI,
            LCARS_GOLD, CO2_ZONES,
            )
        ...
    sensor_graphs[view_mode] = (grp, bmp, vlbl, mlbl)
```

The `graph_layout()` function dynamically sizes the bars based on how much data exists. With only a few samples the bars are wide and chunky, filling half the graph width. As data accumulates over two hours the bars slim down and the graph expands to fill the full width.

Each bar is colored by the reading's value — green for safe levels, yellow for elevated and red for high. A 1-pixel gap separates each bar. Each sensor type has its own zone thresholds defined as a list of tuples.

- `GRAPH_CO2_LO` and `GRAPH_CO2_HI` — the floor and ceiling for the CO2 graph. The floor is set to 300 so that normal 400 ppm readings have visible bar height.
- `HISTORY_SIZE` — number of samples stored. At 120 samples with a 60-second interval this gives 2 hours of data.
- `SAMPLE_INTERVAL` — seconds between history samples. All three sensors are sampled together.
- `ZOOM_SAMPLES` — number of samples shown when zoomed in. Set to 15 for a 15-minute window.

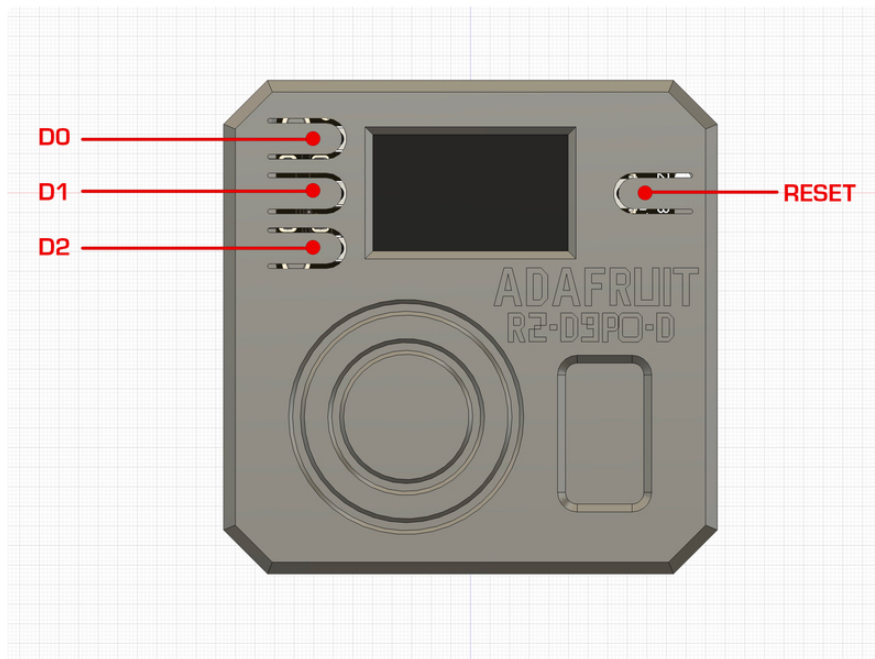
LCARS Stats Screen

The diagnostics screen shows session data in a two-column layout with LCARS blue labels on the left and gold values on the right. It displays uptime in hours and minutes, samples collected out of 120, and current reading with min/max range for CO2,

humidity and temperature. The battery percentage is shown at the bottom. Temperature min/max respects the current Fahrenheit or Celsius setting.

LCARS Name Badge

The badge uses a gold LCARS frame with blue, orange, and red accent blocks across the top. Your name renders at 2x scale for visibility. Below that are the title, ID and project name. The QR code is generated at runtime using the `adafruit_miniqr` library and positioned on the right side of the screen. It uses low error correction and type-3 encoding to keep the module count small enough to scan from the 135-pixel display.



View Cycling

Six views are available. D2 short press cycles through the first five in order: bars, stats, CO2 graph, humidity graph, temperature graph. The about/badge screen is accessed separately via D1 long press. D0 short press always returns to the bar view from any other screen.

```
VIEW_BARS = 0
VIEW_STATS = 1
VIEW_CO2 = 2
VIEW_HUM = 3
VIEW_TEMP = 4
VIEW_ABOUT = 5
```

Each LCARS screen is built lazily the first time it is accessed using the `sensor_graphs` dictionary. The graph zoom state resets when cycling away from graph views.

Button System

The button polling runs inside the sensor update interval at 50-millisecond resolution. Each button tracks its press-down timestamp for long press detection and a flag to prevent the short press from firing after a long press.

D0 also supports double-tap detection. On release, the short press action is delayed by `DOUBLE_TAP_MS` of 400 milliseconds. If D0 is pressed again within that window, it triggers demo mode instead. The D0 plus D2 combo hold is checked inside the D0 long press handler — if D2 is also held when D0 reaches the long press threshold, it triggers sleep.

```
Button map:
D0 short:  rescan / return to bars
D0 double: demo mode
D0 long:   CO2 calibration
D1 short:  F/C toggle (bars) / zoom (graphs)
D1 long:   toggle name badge
D2 short:  cycle views
D2 long:   brightness toggle
D0+D2:    sleep
```

Sleep Mode

The project uses light sleep instead of deep sleep because the ESP32-S3 Reverse TFT Feather's D0/BOOT0 pin has external circuitry that causes false wake triggers with PinAlarm. The light sleep approach blanks the display and enters a loop that sleeps for 2 seconds at a time using TimeAlarm, then polls the buttons.

```
while True:
    t_alarm = alarm.time.TimeAlarm(
        monotonic_time=time.monotonic() + 2
    )
    alarm.light_sleep_until_alarms(t_alarm)
    if not btn_d0.value or btn_d1.value or btn_d2.value:
        break
```

Before sleeping, the code waits for all buttons to be released to prevent immediate wake. The CO2 calibration offset is saved to `alarm.sleep_memory` so it persists. On wake, the display brightness is restored and a boot scan animation plays. Sleep triggers automatically after 5 minutes of no button activity or manually with a D0 plus D2 long press.

CO2 Calibration

Long pressing D0 runs a software CO2 offset calibration. Take the device outside to fresh air and hold D0 for 2 seconds. The code reads the current CO2 value and computes the offset needed to map it to 400 ppm, the standard outdoor baseline.

```
co2_offset = 400 - int(raw)
save_co2_offset()
```

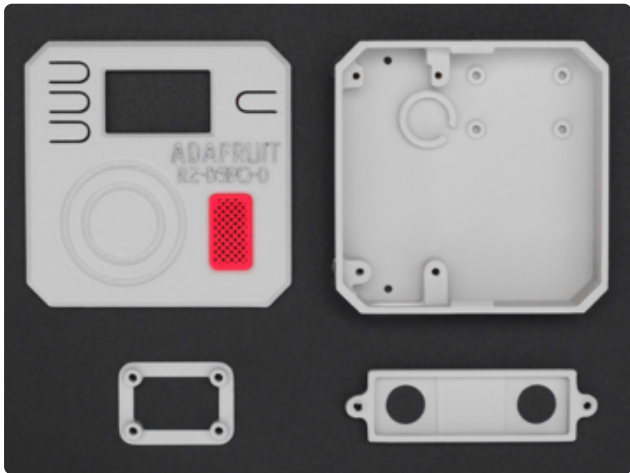
The offset is stored as a 4-byte signed integer in `alarm.sleep_memory` bytes 0 through 3, written byte-by-byte since the ESP32-S3 port does not support `struct.pack_into` directly on the sleep memory object. The offset applies to all readings going forward and survives sleep/wake cycles. To clear it, calibrate again in known-good air.

Demo Mode

Double-tapping D0 activates demo mode for hands-free video recording. It cycles through every screen automatically with a 3-second hold on each.

1. Bar view with rescan animation
 2. LCARS diagnostics
 3. CO2 graph zoomed in (15 minutes)
 4. CO2 graph full view
 5. Humidity graph zoomed in
 6. Humidity graph full view
 7. Temperature graph zoomed in
 8. Temperature graph full view
 9. LCARS name badge with QR code
 10. Return to bars with scan animation
- `DEMO_HOLD` — seconds to hold on each screen. Default is 3.

3D Printing



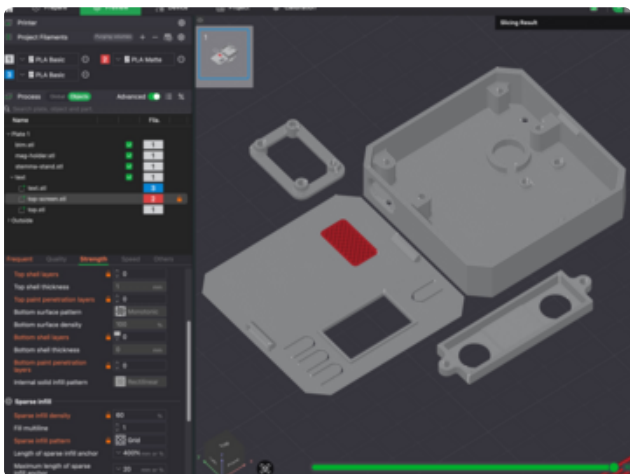
3MF files for 3D printing are oriented and ready to print on FDM machines using PLA filament. Original design source files may be downloaded using the links below.

<https://adafru.it/1aCi>

<https://adafru.it/1aCj>



dropdown on the Fusion 360 site allows you to pick your preferred 3D file format like STEP, STL, etc.



Slice with settings for PLA material

The parts were sliced using BambuStudio using the slice settings below.

PLA filament 220c extruder

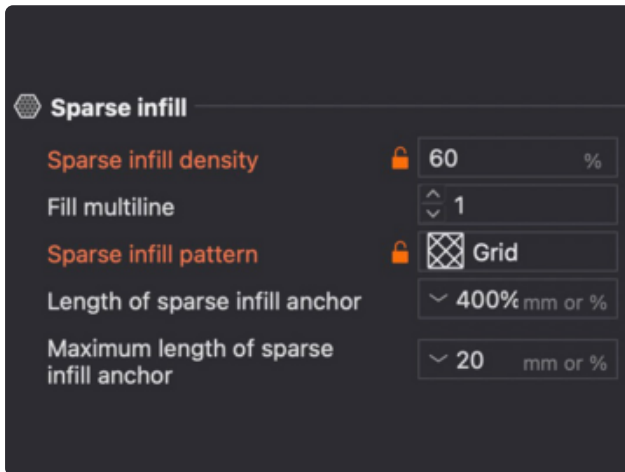
0.2 layer height

10% gyroid infill

200mm/s print speed

Tree Supports

60 C heated bed



Grid Setting for Sensor

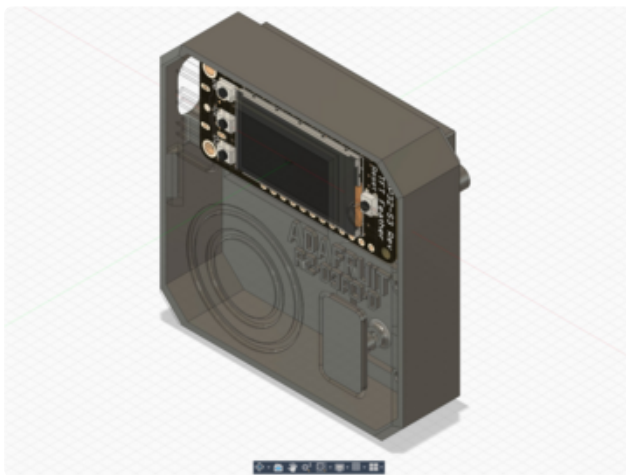
The sensor portion of the case uses separate settings to generate a grid infill with no tops or bottoms.

Infill density: 60%

Sparse infill pattern: Grid

Top + Bottom shell: 0

Top + Bottom penetration layers: 0

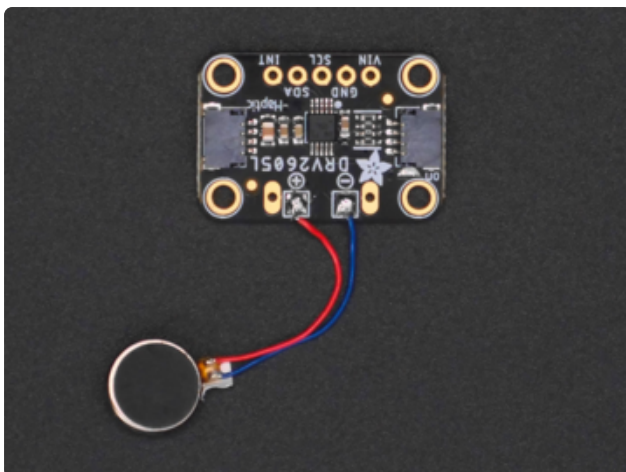


Design Source Files

The project assembly was designed in Fusion 360. Once opened in Fusion 360, It can be exported in different formats like STEP, STL and more.

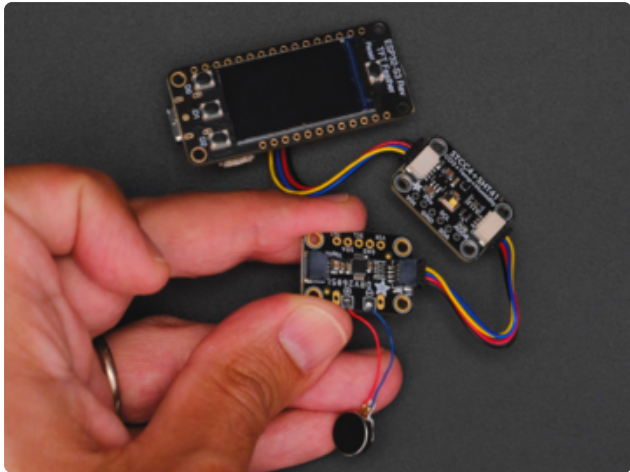
Electronic components like Adafruit's boards, displays, connectors and more can be downloaded from the [Adafruit CAD parts GitHub Repo](https://adafru.it/RvF) (<https://adafru.it/RvF>)

Assembly



Solder Motor

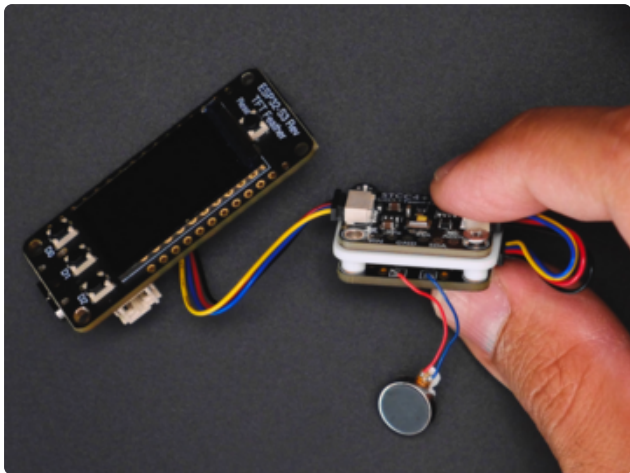
Solder the **Red** wire to the + pad and the **Blue** to the - pad on the DRV board.



STEMMA QT Chain of Devices

Connect a 50mm long STEMMA QT cable to the port on the Feather.

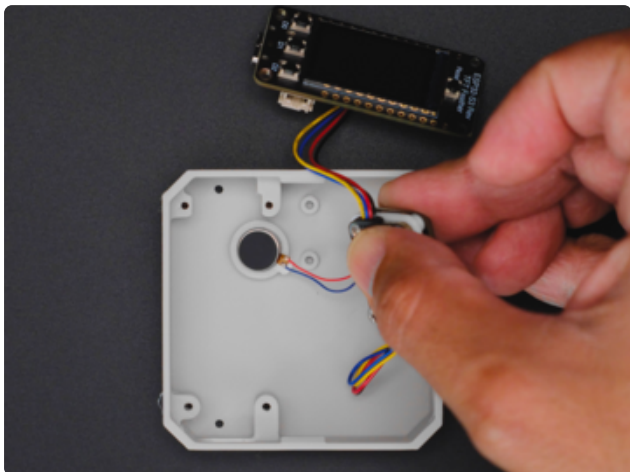
The STCC4 / SHT41 - CO2 board connects next in the STEMMA QT chain, followed by the DRV board at the end.



STEMMA QT Sandwich

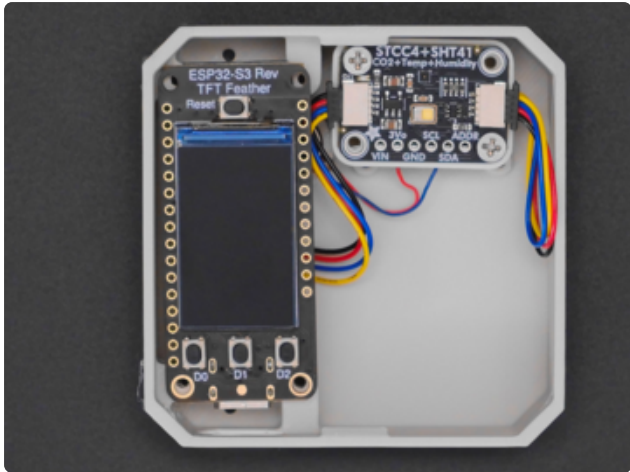
Place the board standoff between the DVR board below and the STCC4 board on top. Use the extruded standoff portion to clear the STEMMA QT ports on the DVR board.

Use M2.5x10mm screws to securely mount the STEMMA QT sandwich together.



Mount Motor

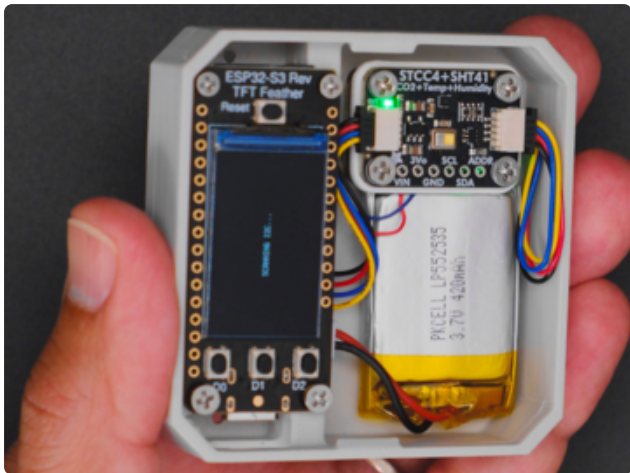
Align the wires on the vibration motor to the cutaway on the circular wall in the case.



Mount Boards

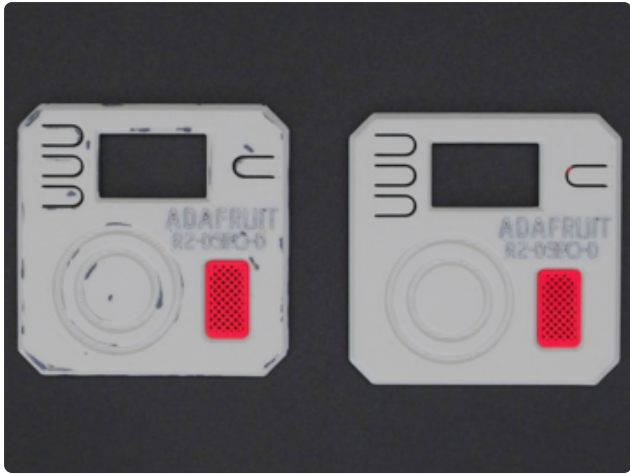
Secure the Feather to the stand-offs on the case with two M2.5x6mm screws on the side closest to the three buttons. Use two M2x6mm screws on the remaining screw mounts closest to the Reset button.

Position the STEMMA QT sandwich to the stand-offs inside the case.



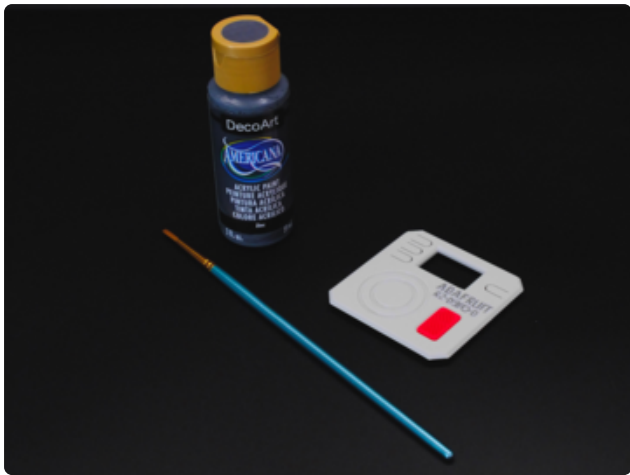
Connect Lipo Battery

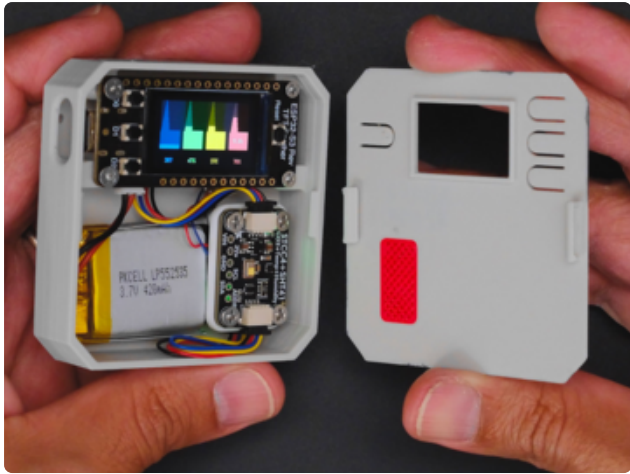
Align the battery cable to the JST port on the Feather.



Weathering Paint

Use dark grey acrylic paint to add weathering details to the front and side of the case.

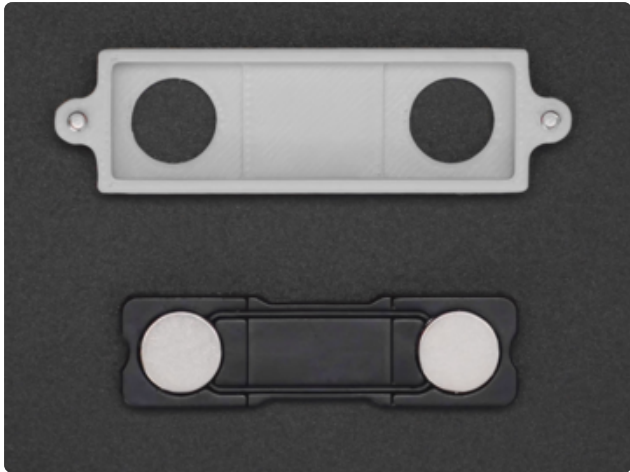




Snap-fit Case Top

Align the snap fit nubs to connect the top of the case to the bottom of the case.





Assemble Magnetic Backing

The magnetic pin backing attaches to the case with a printed holder.



Align the magnets through the cutouts on the holder.

Use two M2.5x6mm screws to attach the holder to the back of the case.



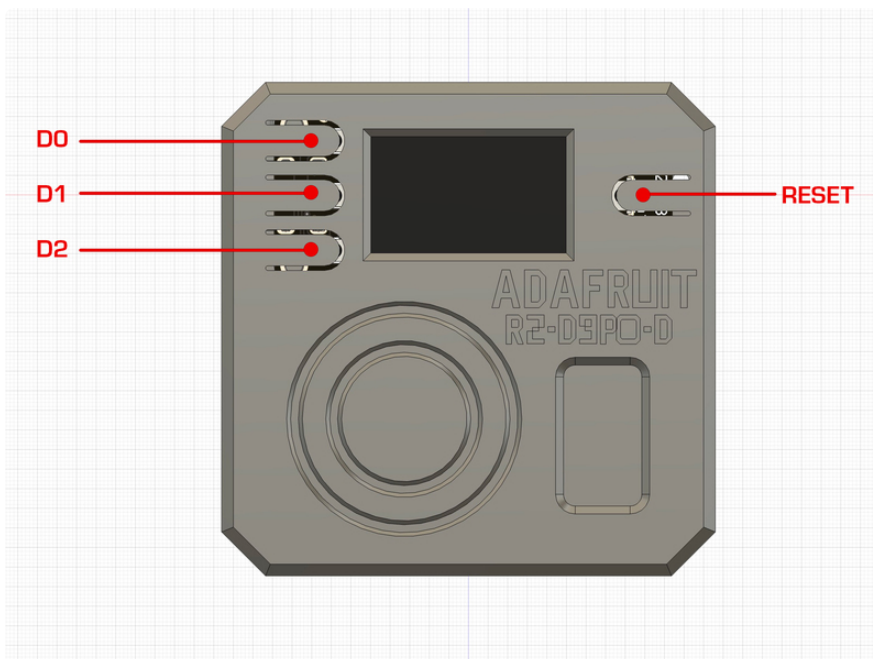
Usage

The Data Dispenser has three buttons along the edge of the Feather board, labeled D0, D1, and D2 from top to bottom. Each button has a short press, long press and in some cases a double-tap or combo action. This page covers every feature and how to access it.



Quick Reference

- D0 short Rescan animation / return to bars
- D0 double Demo mode (auto-cycle all screens)
- D0 long CO2 offset calibration
- D1 short F/C toggle (bars) / zoom (graphs)
- D1 long Toggle LCARS name badge
- D2 short Cycle views
- D2 long Toggle brightness (full / dim)
- D0+D2 long Sleep
- 5 min idle Auto sleep
- Any button Wake from sleep



Boot Up

When you power on the device or wake it from sleep, the display runs a scan animation. All four bars sweep up from zero to full, then settle to the current sensor readings. A three-pulse haptic buzz confirms the scan. The STCC4 sensor needs a couple of seconds to stabilize after power-on, so the first CO2 reading may be slightly off — it settles within a few update cycles.

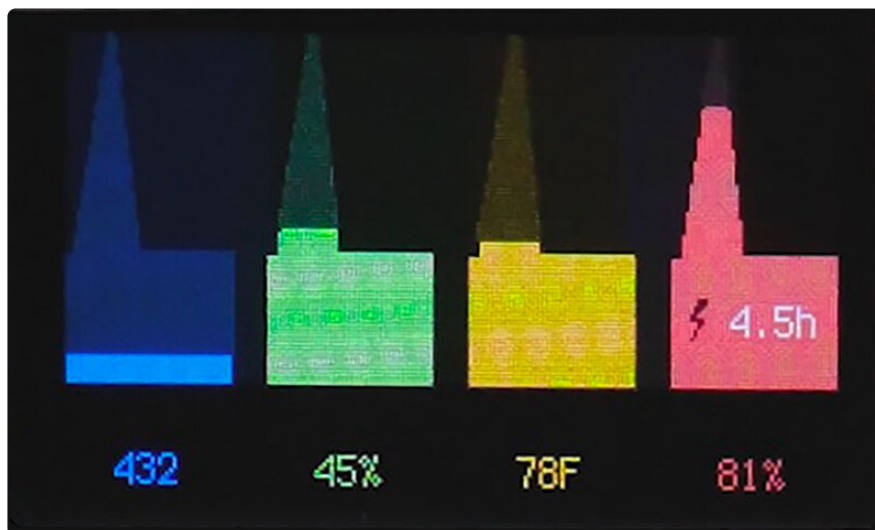
Bar Graph View

This is the default home screen. Four shaped colored bars show live sensor data, each updating every two seconds. From left to right the bars are:

- Blue for CO2 in parts per million
- Green for humidity as a percentage
- Yellow for temperature in Fahrenheit or Celsius
- Red for battery level as a percentage.

The bar shape matches the original prop's on-screen graphic — a narrow peak at the top that tapers into a wide base. Each bar has a subtle anti-aliased edge and a colored gradient background behind it.

The label below each bar shows the current value. CO2 displays as a raw ppm number, humidity and battery show a percentage, and temperature shows the value either in Fahrenheit (F) or Celsius (C).



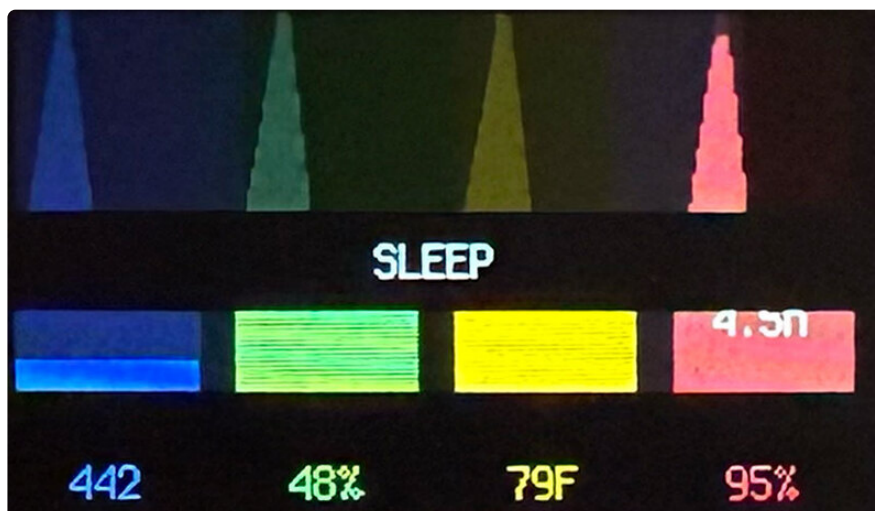
Battery Indicator

When the device is charging via USB, a blinking white lightning bolt appears inside the base of the battery bar along with the estimated time to full charge. The percentage label stays in its normal position below the bar.

When running on battery, the estimated time remaining appears inside the bar base in white text. The estimate becomes more accurate after a few minutes as the MAX17048 fuel gauge learns the current draw. Under one hour it switches from hours to minutes.

CO2 Warning Alert

When the CO2 reading exceeds the warning threshold of 700 ppm, the haptic motor fires a double-pulse vibration alert. The alert has a 30-second cooldown to avoid constant buzzing in a stuffy room. You can adjust the threshold by changing `CO2_WARN` in the tunables section of `code.py`.

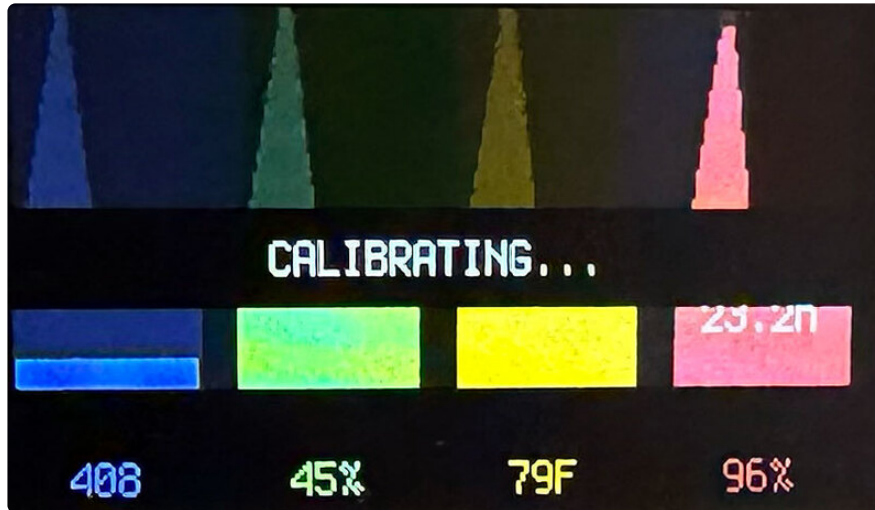


Sleep Mode

D0 plus D2 long press: Hold both D0 and D2 for two seconds to put the device to sleep immediately. The display blanks and the device enters a low-power light sleep loop.

Auto sleep: If no button is pressed for five minutes, the device sleeps automatically. You can change this timeout by editing `IDLE_SLEEP_S` in the tunables.

Wake: Press any button to wake the device. The display restores to the previous brightness setting and a boot scan animation plays. The CO2 calibration offset is preserved across sleep.

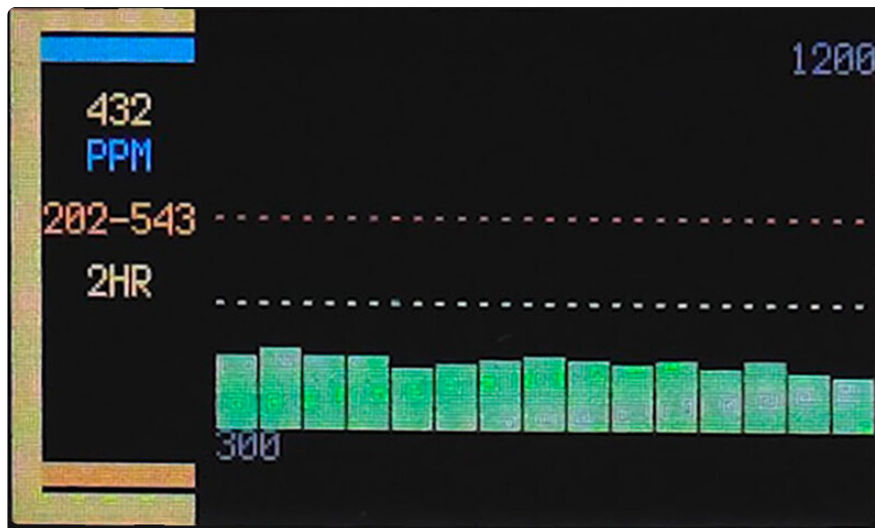


Button D0 — Scan and Calibrate

Short press: On the bar view, D0 triggers a rescan animation. The bars surge from their current level to full, drain to empty, then refill to the latest sensor values with a staggered cascade. On any other screen, D0 returns you to the bar view.

Double tap: Quickly press D0 twice to enter demo mode. The device automatically cycles through every screen with a three-second hold on each — bar view with animation, diagnostics, all three graphs in both zoomed and full views, the name badge, then back to bars. This is hands-free and great for video recording or showing off the prop.

Long press (2 seconds): Runs CO2 offset calibration. Take the device outside to fresh air first. Hold D0 for two seconds and the screen shows CALIBRATING followed by the computed offset value. The offset adjusts all future CO2 readings so that outdoor air reads as 400 ppm. The offset is saved to sleep memory and survives sleep/wake cycles. To reset the calibration, repeat the process in known-good outdoor air.



Button D1 — Toggle and Zoom

Short press on bar view: Toggles the temperature display between Fahrenheit and Celsius. The temperature bar plays a solo rescan animation — surging to full, draining, then refilling with the new unit label.

Short press on graph views: Toggles zoom. When zoomed in, the graph shows the last 15 minutes of data with wide chunky bars that fill the full graph width. Press again to zoom back out to the full history view. The zoom state resets when you leave the graph views.

Long press (2 seconds): Toggles the LCARS name badge screen. Long press again to return to whatever view you were on before.

Button D2 — Cycle Views and Brightness

Short press: Cycles through the display views in order: bar graph, LCARS diagnostics, CO2 graph, humidity graph, temperature graph, then back to bars. Each graph screen is built the first time you access it to save memory at boot.

Long press (2 seconds): Toggles display brightness between full (1.0) and dim (0.15). The dim mode is useful for indoor display or when the bright screen would be distracting. The brightness setting is remembered through sleep/wake cycles.



LCARS Diagnostics Screen

The stats screen shows a two-column layout with LCARS-style framing. The left column has labels in blue and values in gold on the right. It displays the current session uptime in hours and minutes, the number of history samples collected out of 120, and the current reading with session min/max range for CO2, humidity and temperature. Battery percentage is shown at the bottom.

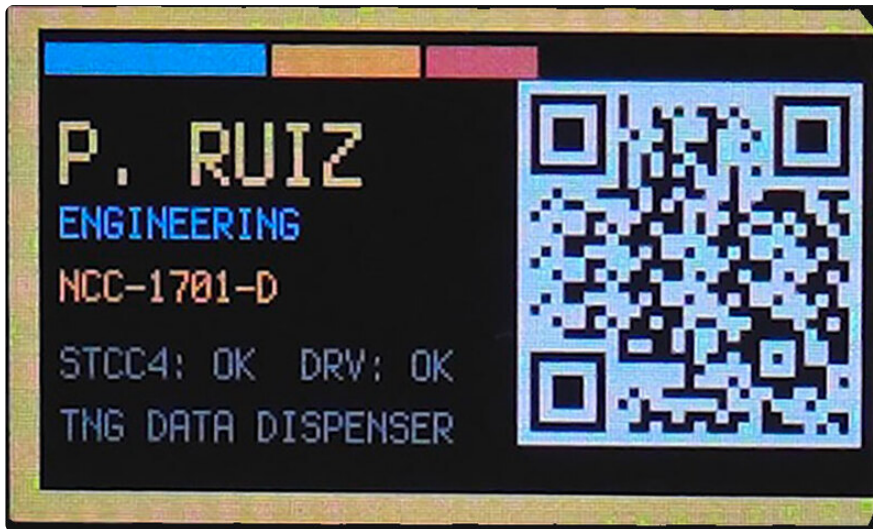
LCARS Graph Screens

Three separate graph screens show rolling history for CO2, humidity and temperature. Each graph is a bar histogram with the LCARS frame on the left showing the current value, unit, min/max range and a 2HR time window label.

The bars are color-coded by zone. For CO2, green is safe (under 600 ppm), yellow is elevated (600 to 800 ppm) and red is high (above 800 ppm). Humidity and temperature have their own zone thresholds tuned to typical comfort ranges.

Data is sampled every 60 seconds into a circular buffer of 120 entries, giving two hours of rolling history. The bar width adjusts dynamically — with just a few samples the bars are wide and centered, gradually slimming down as the buffer fills. Dashed threshold lines mark the zone boundaries.

Press D1 on any graph screen to toggle zoom. The zoomed view shows the last 15 minutes with wide bars for a detailed close-up of recent changes.



LCARS Name Badge

The badge screen shows your personalized information in the LCARS style with a gold frame and colored accent blocks. Your name appears at double size, followed by your title, registry ID, sensor status and project name. A QR code on the right side links to the URL you specify — scannable directly from the display at conventions or meetups.

All badge text is editable in the tunables section at the top of `code.py`: `BADGE_NAME`, `BADGE_TITLE`, `BADGE_ID`, `BADGE_PROJECT` and `BADGE_QR_URL`. Set the QR URL to an empty string to hide the QR code.

