

PROJECT 1

Industrial Object Counter

Build a miniature production-line sensing station with reliability logic and a live monitoring dashboard.

Audience First- and second-year engineering students | Format Guided 90-minute build

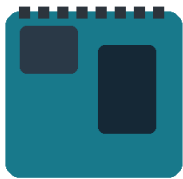
Learning model Predict → Wire → Code → Test → Explain → Extend

Name: _____ Group: _____ Date: _____

VISUAL QUICK START

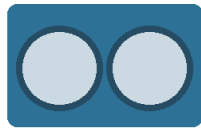
Identify the components before wiring. The Arduino is the controller, the HC-SR04 is the input sensor, and LEDs or the piezo are outputs.

MEET YOUR COMPONENTS



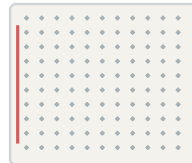
ARDUINO UNO

Reads inputs + runs code



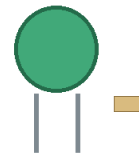
HC-SR04

Measures distance



BREADBOARD

Connects parts without solder



LED + RESISTOR

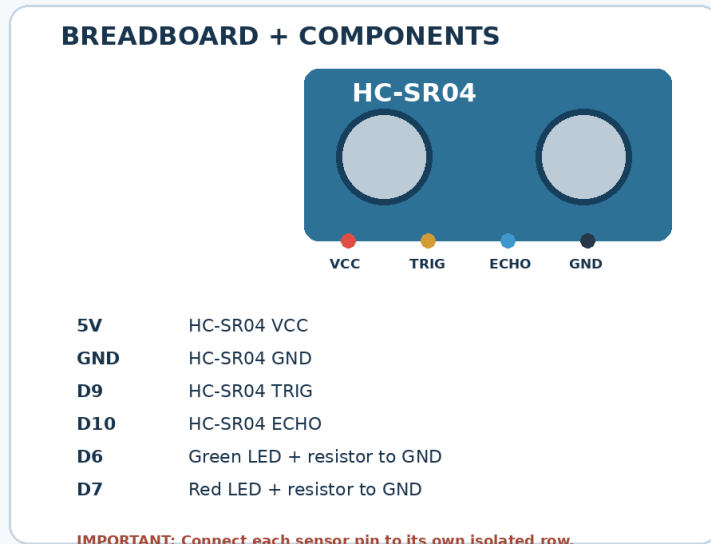
Shows READY / ACTIVE

WIRING MAP

Read the wiring table and follow the labeled Arduino pins. Sensor VCC goes to 5V, sensor GND to GND, TRIG to D9 and ECHO to D10. Use separate breadboard rows for each component leg.

PROJECT 1 | WIRING AT A GLANCE

Conceptual layout - follow labels and pin numbers, not physical spacing.



Breadboard rule: A-E holes in one numbered row connect together; F-J holes in that row connect together, but NOT across the centre channel. Power rails are independent and may be split along their length.

Power off before rewiring. Every LED needs its own 220-330 ohm series resistor. For Project 2, use a passive low-current piezo (not a high-current alarm module).

PRACTICE CHECKPOINT

- ☐ I can identify VCC, TRIG, ECHO and GND. ☐ Each sensor pin has a separate breadboard connection.
- ☐ I checked the Arduino pin numbers. ☐ I verified LED resistors / piezo polarity.
- ☐ I unplugged USB while changing wires.

1 Mission and learning outcomes

A factory needs to count objects as they move through a detection zone. Your job is to build a microcontroller that senses each arrival, counts it once, rejects short glitches, indicates operating state, and reports production data to a laptop.

- Explain the roles of sensors, microcontrollers, actuators/indicators, and telemetry.
- Program an HC-SR04 to measure distance using ultrasound time-of-flight.
- Build a two-state finite-state machine (READY ↔ OCCUPIED).
- Use hysteresis and consecutive-reading confirmation to improve reliability.
- Interpret a CSV-like serial protocol and connect to a ready-to-run Python dashboard.

2 Kit and workshop timeline

Part	Quantity / notes
Arduino Uno + USB cable	1

HC-SR04 ultrasonic module	1
Breadboard + male-to-male jumpers	1 breadboard, 4+ wires
Green LED and red LED	1 of each
220–330 Ω resistors	2, one per LED
Laptop	Arduino IDE + Python 3 and PySerial installed
Minutes	Build milestone
0–10	Introduction, safe wiring, predict readings
10–25	Read distance and test sensor
25–45	Count object entry using states
45–60	Confirm detection and add hysteresis
60–75	Wire and control green/red LEDs
75–90	Open supplied dashboard; test and document

Before you begin: breadboard basics and safety

- Disconnect USB power before moving wires. Double-check the Arduino 5V and GND connections before reconnecting.
- In a typical breadboard, the five holes in each numbered row on one side of the center gap share an electrical connection. The center gap separates the two sides; side power rails may be split.
- Place each HC-SR04 pin in a different electrically connected row. Do not accidentally join VCC, TRIG, ECHO, and GND.
- LEDs require a series current-limiting resistor (220–330 ohms is suitable for the specified workshop LEDs). The long LED leg is normally the anode (+).
- The projects use USB-powered Arduino Uno with low-current sensors, LEDs and a small passive piezo. Do not attach a high-torque servo or bare DC motor directly to Arduino I/O pins.

Core Arduino coding vocabulary

Term	Meaning
setup()	Runs once after power-up or reset; configure pins and serial communication.
loop()	Repeats continuously while the board is running.
pinMode()	Configures a pin as an input or output.
digitalWrite()	Drives an output HIGH or LOW.
pulseIn()	Measures the duration of a digital pulse; optional timeout prevents indefinite blocking.
Serial.begin(9600)	Starts USB serial communication at 9600 baud.
millis()	Reports elapsed milliseconds since reset, useful for timing.

3 Wiring: from a blank breadboard

Begin with the Arduino USB disconnected. Place the HC-SR04 in electrically separate rows and connect as shown. Each jumper is a direct electrical connection between its sensor row and Arduino header.

Component terminal	Arduino pin	Purpose
HC-SR04 VCC	5V	Sensor power
HC-SR04 GND	GND	Common return
HC-SR04 TRIG	D9	Start an ultrasonic pulse
HC-SR04 ECHO	D10	Read echo duration
Green LED anode (+)	D6	READY status
Green LED cathode (–)	220–330 Ω resistor → GND	Limit current
Red LED anode (+)	D7	OCCUPIED status
Red LED cathode (–)	220–330 Ω resistor → GND	Limit current

The LEDs may be added later at Milestone 4. Ground may be distributed via the breadboard negative rail, provided that rail is joined to Arduino GND.

CHECKPOINT Before power-on: are VCC and GND correct, and is each LED in series with its own resistor?

4 Guided coding milestones

Milestone 1 — Make the sensor report a distance

A 10-microsecond trigger pulse starts measurement. The echo pulse length is the round-trip travel time. With the speed of sound approximated as 0.0343 cm/ μ s, distance = duration $\times$ 0.0343 $\div$ 2. We divide by two because sound travels to the object and back.

```
const int trigPin = 9, echoPin = 10;
void setup() {
  Serial.begin(9600);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}
void loop() {
  digitalWrite(trigPin, LOW); delayMicroseconds(2);
  digitalWrite(trigPin, HIGH); delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  unsigned long us = pulseIn(echoPin, HIGH, 25000UL);
  if (us == 0) Serial.println("No echo");
  else Serial.println(us * 0.0343 / 2.0);
  delay(200);
}
```

CHECKPOINT Open Serial Monitor at 9600 baud. Does the reading increase when you move a flat object farther away?

Milestone 2 — Count transitions, not samples

Naively incrementing the counter whenever distance $\leq$ 5 cm would count the same stationary box repeatedly. Instead, a READY state changes to OCCUPIED only when a new object enters. On that transition, increment once. Stay OCCUPIED until the object leaves.

```
bool objectPresent = false;
unsigned long objectCount = 0;
// After reading a valid distance into distance:
if (!objectPresent && distance <= 5.0) {
  objectPresent = true;
```

```

objectCount++;
}
else if (objectPresent && distance >= 40.0) {
  objectPresent = false;
}

```

The 5 cm entry / 40 cm release thresholds are the values used in our tested prototype. This large gap requires moving the object far away before the system can count another pass; teams can tune it for a real conveyor geometry.

CHECKPOINT Hold a box near the sensor for five seconds: does the count remain unchanged after its first detection?

Milestone 3 — Reject glitches with consecutive readings

Require three consecutive valid measurements for each state change. A single false short-range reflection no longer counts as a product. Invalid echoes clear the confirmation counters.

```

const int REQUIRED_READINGS = 3;
int detectionReadings = 0, releaseReadings = 0;
// READY: each distance <= 5 cm increments detectionReadings.
// Any distance > 5 cm resets detectionReadings to 0.
// At 3 consecutive matches: count once and enter OCCUPIED.
// OCCUPIED: each distance >= 40 cm increments releaseReadings.
// Any distance < 40 cm resets releaseReadings to 0.
// At 3 matches: return to READY.

```

CHECKPOINT Wave your hand quickly through the zone. Do brief appearances get rejected more often than deliberate passes?

Milestone 4 — Add physical feedback and structured telemetry

Green = READY; red = OCCUPIED. The Arduino then sends a DATA line every 500 ms for the supplied dashboard. Keep the format consistent: DATA,elapsed_ms,distance_cm,total_count,state,average_rate.

```

digitalWrite(GREEN_LED, objectPresent ? LOW : HIGH);
digitalWrite(RED_LED, objectPresent ? HIGH : LOW);
// Example dashboard message:
// DATA,12500,4.32,7,OCCUPIED,33.60

```

The rate sent in this workshop firmware is the running average since startup, not an instantaneous production rate. The enhanced desktop application also calculates a rolling-window rate.

5 Verification and troubleshooting

Test	Expected result
No object in detection zone	Green on, red off, state READY
Object held at ≤ 5 cm	After three valid samples: count +1, red on
Object still present	No repeated count
Object at ≥ 40 cm	After three valid samples: green on again
Echo timeout	No new object count; look for timeout notice
Dashboard open	Counts, graph and state respond to the Arduino

If every measurement times out: inspect TRIG/ECHO pins and check for a stable 5V/GND. If readings fluctuate, face a broad flat object toward the sensor. If the Python port is busy, close Serial Monitor/Serial Plotter. On macOS the port name may change when reconnecting USB.

6 Install and run the dashboard

1. Install Python 3 and ensure Tkinter is available. In Terminal run: `python3 -m pip install pyserial`
2. Upload the complete Arduino sketch from Appendix A.
3. Close Arduino Serial Monitor and Serial Plotter. Only one program can own the serial port.
4. Save Appendix B as `industrial_dashboard_project1.py` (or use the separately supplied source file).
5. Run: `python3 industrial_dashboard_project1.py`; choose the Arduino port and click Connect.
6. Verify that the dashboard updates, then optionally start CSV recording and test simulated production passes.

Default prototype Mac port: `/dev/cu.usbmodem11201`. Different Macs may use different ports. Connecting through serial may reset the Uno and restart its counter.

7 Think like an embedded engineer

- Why does the code detect a transition rather than measure proximity alone?
- What is the trade-off between three-reading confirmation and counting fast-moving boxes?
- What happens when the sensor times out while OCCUPIED? Would your intended factory consider that a fault?
- How would you measure actual throughput over the most recent minute?
- Extension: create a FAULT state, or display counts/min with a rolling window.

Workshop submission

Show your instructor: (1) a working counter, (2) one five-second stationary-object test, (3) the laptop dashboard, and (4) a sentence explaining hysteresis. Record your chosen threshold and an example measurement.

Appendix A — Complete Arduino sketch (upload this)

```
const int trigPin = 9;
const int echoPin = 10;
const int GREEN_LED = 6;
const int RED_LED = 7;

const float DETECT_THRESHOLD = 5.0;
const float RELEASE_THRESHOLD = 40.0;
const int REQUIRED_READINGS = 3;

bool objectPresent = false;
unsigned long objectCount = 0;
int detectionReadings = 0;
int releaseReadings = 0;
unsigned long systemStartTime = 0;
unsigned long lastReportTime = 0;

void setup() {
  Serial.begin(9600);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(GREEN_LED, OUTPUT);
  pinMode(RED_LED, OUTPUT);
  digitalWrite(trigPin, LOW);
  digitalWrite(GREEN_LED, HIGH);
  digitalWrite(RED_LED, LOW);
  systemStartTime = millis();
  Serial.println("INDUSTRIAL OBJECT COUNTER");
```

```

Serial.println("System initialized");
}

float measureDistance() {
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  unsigned long duration = pulseIn(echoPin, HIGH, 25000UL);
  if (duration == 0) return -1.0;
  return duration * 0.0343 / 2.0;
}

void loop() {
  float distance = measureDistance();
  if (distance < 0) {
    detectionReadings = 0;
    releaseReadings = 0;
    Serial.println("Sensor timeout");
  } else {
    if (!objectPresent) {
      if (distance <= DETECT_THRESHOLD) {
        detectionReadings++;
        if (detectionReadings >= REQUIRED_READINGS) {
          objectPresent = true;
          objectCount++;
          detectionReadings = 0;
          Serial.println("OBJECT COUNTED");
        }
      } else {
        detectionReadings = 0;
      }
    } else {
      if (distance >= RELEASE_THRESHOLD) {
        releaseReadings++;
        if (releaseReadings >= REQUIRED_READINGS) {
          objectPresent = false;
          releaseReadings = 0;
        }
      } else {
        releaseReadings = 0;
      }
    }
  }
  digitalWrite(GREEN_LED, objectPresent ? LOW : HIGH);
  digitalWrite(RED_LED, objectPresent ? HIGH : LOW);
  unsigned long now = millis();
  if (now - lastReportTime >= 500) {
    lastReportTime = now;
    float elapsedMinutes = (now - systemStartTime) / 60000.0;
    float rate = elapsedMinutes > 0 ? objectCount / elapsedMinutes : 0;
    Serial.print("DATA,");
    Serial.print(now);
    Serial.print(",");
    Serial.print(distance);
    Serial.print(",");
    Serial.print(objectCount);
    Serial.print(",");
    Serial.print(objectPresent ? "OCCUPIED" : "READY");
    Serial.print(",");
    Serial.println(rate, 2);
  }
  delay(100);
}

```

Appendix B — Complete Python monitoring dashboard

Save the following code into `industrial_dashboard_project1.py`. This full program is provided for students; understanding the Arduino portion is the primary workshop coding goal.

```

"""IEEE Project 1: Industrial Object Counter dashboard.
Requires: pip install pyserial
Arduino sends: DATA,millis,distance_cm,total_count,READY|OCCUPIED,avg_rate
"""

import csv
import math
import time
from collections import deque
from datetime import datetime
from pathlib import Path
import tkinter as tk
from tkinter import ttk, filedialog, messagebox

import serial
from serial.tools import list_ports

PORT = '/dev/cu.usbmodem11201'
BAUD = 9600
ROLLING_WINDOW_MS = 60_000
HISTORY_SECONDS = 60

def parse_telemetry(line):
    parts = line.strip().split(',')
    if len(parts) != 6 or parts[0] != 'DATA':
        return None
    try:
        ms = int(parts[1]); distance = float(parts[2]); count = int(parts[3])
        state = parts[4]; avg_rate = float(parts[5])
        if ms < 0 or count < 0 or state not in ('READY', 'OCCUPIED'):
            return None
        if not math.isfinite(distance) or not math.isfinite(avg_rate):
            return None
        return {'ms': ms, 'distance': distance, 'count': count,
                'state': state, 'avg_rate': avg_rate}
    except ValueError:
        return None

class Dashboard:
    def __init__(self, root):
        self.root = root
        root.title('IEEE | Industrial Production Monitoring — Project 1')
        root.geometry('1050x780')
        root.minsize(860, 690)
        self.serial_port = None
        self.buffer = ""
        self.last_packet_at = None
        self.previous = None
        self.points = deque(maxlen=240)
        self.events = deque()
        self.last_count_at = None
        self.last_cycle_seconds = None
        self.occupied_since = None
        self.invalid_samples = 0
        self.session_events = 0
        self.logging = False
        self.log_file = None
        self.log_writer = None
        self.last_render_at = 0

        outer = ttk.Frame(root, padding=16)
        outer.pack(fill='both', expand=True)

```

```

    ttk.Label(outer, text='IEEE | INDUSTRIAL MONITORING STATION',
              font=('Helvetica', 22, 'bold')).pack(anchor='w')
    ttk.Label(outer, text='Project 1 • Ultrasonic object counter • Live embedded telemetry',
              font=('Helvetica', 11)).pack(anchor='w', pady=(2, 12))

    controls = ttk.Frame(outer)
    controls.pack(fill='x', pady=(0, 12))
    ttk.Label(controls, text='Serial port:').pack(side='left')
    self.port_var = tk.StringVar(value=PORT)
    self.port_combo = ttk.Combobox(controls, textvariable=self.port_var, width=29,
                                   values=self.available_ports())
    self.port_combo.pack(side='left', padx=7)
    ttk.Button(controls, text='Refresh ports', command=self.refresh_ports).pack(side='left', padx=3)
    ttk.Button(controls, text='Connect', command=self.connect).pack(side='left', padx=3)
    ttk.Button(controls, text='Disconnect', command=self.disconnect).pack(side='left', padx=3)
    self.log_button = ttk.Button(controls, text='Start CSV log', command=self.toggle_log)
    self.log_button.pack(side='right')

    self.connection_text = tk.StringVar(value='DISCONNECTED')
    self.warning_text = tk.StringVar(value='Connect to Arduino to begin receiving data.')
    ttk.Label(outer, textvariable=self.connection_text, font=('Helvetica', 12, 'bold')).pack(anchor='w')
    ttk.Label(outer, textvariable=self.warning_text, wraplength=980).pack(anchor='w', pady=(2, 12))

    metrics = ttk.Frame(outer)
    metrics.pack(fill='x', pady=(0, 13))
    for i in range(4):
        metrics.columnconfigure(i, weight=1)
    self.count_value = self.metric(metrics, 'TOTAL COUNT', '—', 0)
    self.distance_value = self.metric(metrics, 'DISTANCE', '—', 1)
    self.state_value = self.metric(metrics, 'SENSOR STATE', '—', 2)
    self.rate_value = self.metric(metrics, 'LAST 60 s RATE', '—', 3)

    stats = ttk.Frame(outer)
    stats.pack(fill='x', pady=(0, 12))
    self.avg_var = tk.StringVar(value='Average since boot: —')
    self.cycle_var = tk.StringVar(value='Time between counts: —')
    self.uptime_var = tk.StringVar(value='Arduino uptime: —')
    self.events_var = tk.StringVar(value='This dashboard session: 0 events')
    for value in (self.avg_var, self.cycle_var, self.uptime_var, self.events_var):
        ttk.Label(stats, textvariable=value).pack(side='left', padx=(0, 25))

    graph_frame = ttk.LabelFrame(outer, text='LIVE SENSOR DISTANCE • last 60 seconds')
    graph_frame.pack(fill='both', expand=True, pady=(0, 12))
    self.graph = tk.Canvas(graph_frame, bg='#101c2a', highlightthickness=0, height=250)
    self.graph.pack(fill='both', expand=True, padx=8, pady=8)
    self.graph.bind('<Configure>', lambda _event: self.draw_graph())

    bottom = ttk.Frame(outer)
    bottom.pack(fill='both', expand=False)
    self.event_log = tk.Text(bottom, height=8, state='disabled', wrap='word', font=('Menlo', 10))
    self.event_log.pack(side='left', fill='both', expand=True)
    scrollbar = ttk.Scrollbar(bottom, command=self.event_log.yview)
    scrollbar.pack(side='right', fill='y')
    self.event_log.configure(yscrollcommand=scrollbar.set)
    self.append_event('Application ready. Arduino firmware stays unchanged.')
    self.root.after(50, self.poll)
    self.root.protocol('WM_DELETE_WINDOW', self.close)

    def metric(self, parent, heading, initial, column):
        frame = ttk.LabelFrame(parent, text=heading, padding=13)
        frame.grid(row=0, column=column, sticky='nsew', padx=4)
        value = tk.StringVar(value=initial)
        ttk.Label(frame, textvariable=value, font=('Helvetica', 23, 'bold')).pack(anchor='w')
        return value

    def available_ports(self):
        return [p.device for p in list_ports.comports()]

```

```

def refresh_ports(self):
    self.port_combo['values'] = self.available_ports()

def append_event(self, text):
    stamp = datetime.now().strftime('%H:%M:%S')
    self.event_log.configure(state='normal')
    self.event_log.insert('end', f'[{stamp}] {text}\n')
    self.event_log.see('end')
    self.event_log.configure(state='disabled')

def reset_session(self):
    self.previous = None
    self.points.clear()
    self.events.clear()
    self.session_events = 0
    self.last_count_at = None
    self.last_cycle_seconds = None
    self.occupied_since = None
    self.invalid_samples = 0
    self.last_packet_at = None
    self.events_var.set("This dashboard session: 0 events")
    self.cycle_var.set("Time between counts: —")
    self.draw_graph()

def connect(self):
    self.disconnect(silent=True)
    try:
        self.serial_port = serial.Serial(self.port_var.get().strip(), BAUD, timeout=0)
        self.buffer = ""
        self.reset_session()
        self.connection_text.set('CONNECTED • waiting for telemetry')
        self.warning_text.set('Opening the port may reset the Arduino count to zero.')
        self.append_event(f'Connected to {self.port_var.get()} at {BAUD} baud')
    except (serial.SerialException, ValueError, OSError) as exc:
        self.serial_port = None
        self.connection_text.set('CONNECTION FAILED')
        self.warning_text.set(str(exc))
        self.append_event(f'Connection failed: {exc}')

def disconnect(self, silent=False):
    if self.serial_port is not None:
        try:
            self.serial_port.close()
        except (serial.SerialException, OSError):
            pass
    self.serial_port = None
    self.buffer = ""
    if not silent:
        self.connection_text.set('DISCONNECTED')
        self.warning_text.set('Telemetry paused. Arduino continues counting on its own.')
        self.append_event('Disconnected')

def toggle_log(self):
    if self.logging:
        self.stop_log()
        return
    path = filedialog.asksaveasfilename(
        title='Save production telemetry', defaultextension='.csv',
        initialfile=f'ieee_counter_{datetime.now():%Y%m%d_%H%M%S}.csv',
        filetypes=[('CSV files', '*.csv')])
    if not path:
        return
    try:
        self.log_file = open(Path(path), 'w', newline="", encoding='utf-8')
        self.log_writer = csv.writer(self.log_file)
        self.log_writer.writerow(['computer_time', 'arduino_millis', 'distance_cm',
                                  'total_count', 'state', 'average_rate_per_min',
                                  'rolling_60s_rate_per_min'])
        self.logging = True
        self.log_button.configure(text='Stop CSV log')

```

```

        self.append_event(f'CSV logging started: {path}')
    except OSError as exc:
        self.warning_text.set(f'Could not start logging: {exc}')
        messagebox.showerror('Logging error', str(exc))

def stop_log(self):
    if self.log_file:
        self.log_file.close()
    self.log_file = None
    self.log_writer = None
    self.logging = False
    self.log_button.configure(text='Start CSV log')
    self.append_event('CSV logging stopped')

def process_packet(self, data):
    now = time.monotonic()
    ms, count, state = data['ms'], data['count'], data['state']
    prev = self.previous
    if prev is not None and (ms < prev['ms'] or count < prev['count']):
        self.append_event('Arduino reset or counter restarted; cleared rate history.')
        self.reset_session()
        prev = None

    if prev is not None and count > prev['count']:
        increment = count - prev['count']
        for _ in range(min(increment, 10000)):
            self.events.append(ms)
        self.session_events += increment
        self.events_var.set(f'This dashboard session: {self.session_events} events')
        if self.last_count_at is not None:
            self.last_cycle_seconds = (ms - self.last_count_at) / 1000.0
            self.cycle_var.set(f'Time between counts: {self.last_cycle_seconds:.2f} s')
        self.last_count_at = ms
        self.append_event(f'+{increment} object(s) detected • total {count}')

    while self.events and ms - self.events[0] > ROLLING_WINDOW_MS:
        self.events.popleft()
    # Rate is a 60-second count normalized per minute, not a prediction.
    rolling_rate = len(self.events)
    self.count_value.set(str(count))
    self.distance_value.set('TIMEOUT' if data['distance'] < 0 else f'{data["distance"]:.1f} cm')
    self.state_value.set(state)
    self.rate_value.set(f'{rolling_rate}/min')
    self.avg_var.set(f'Average since boot: {data["avg_rate"]:.2f}/min')
    self.uptime_var.set(f'Arduino uptime: {ms / 1000:.1f} s')
    self.last_packet_at = now
    self.connection_text.set('LIVE • RECEIVING TELEMETRY')
    if data['distance'] < 0:
        self.invalid_samples += 1
        self.warning_text.set('SENSOR TIMEOUT: ultrasonic echo unavailable. Check alignment and wiring.')
    elif state == 'OCCUPIED':
        self.warning_text.set('OBJECT PRESENT • detection zone occupied')
    else:
        self.warning_text.set('SYSTEM READY • waiting for next object')
    self.points.append((ms, data['distance']))
    if self.logging:
        try:
            self.log_writer.writerow([datetime.now().isoformat(timespec='milliseconds'),
                                     ms, data['distance'], count, state, data['avg_rate'], rolling_rate])
            self.log_file.flush()
        except OSError as exc:
            self.warning_text.set(f'CSV write failed: {exc}')
            self.stop_log()
    self.previous = data
    if now - self.last_render_at > 0.12:
        self.draw_graph()
        self.last_render_at = now

def draw_graph(self):
    canvas = self.graph

```

```

canvas.delete('all')
width = max(canvas.winfo_width(), 350)
height = max(canvas.winfo_height(), 200)
left, right, top, bottom = 58, width - 20, 20, height - 32
valid = [(t, d) for t, d in self.points if d >= 0]
max_d = max(50, math.ceil(max((d for _, d in valid), default=50) / 10) * 10)
for i in range(5):
    y = top + (bottom - top) * i / 4
    canvas.create_line(left, y, right, y, fill='#30455c')
    canvas.create_text(left - 8, y, text=f'{max_d * (4-i)/4:.0f}',
        anchor='e', fill='#c0cad5', font=('Arial', 9))
canvas.create_text(12, 9, text='cm', fill='#c0cad5', anchor='nw')
canvas.create_text(right, height-10, text='NOW', fill='#c0cad5', anchor='e')
canvas.create_text(left, height-10, text='-60 s', fill='#c0cad5', anchor='w')
if not valid:
    canvas.create_text(width/2, height/2, text='Waiting for sensor data', fill='#becbd9')
    return
latest = self.points[-1][0]
coords = []
for ms, d in valid:
    if latest - ms > HISTORY_SECONDS * 1000:
        continue
    x = right - (latest - ms) / (HISTORY_SECONDS * 1000) * (right-left)
    y = bottom - min(d, max_d) / max_d * (bottom-top)
    coords.extend([x, y])
if len(coords) >= 4:
    canvas.create_line(*coords, fill='#56dfad', width=2, smooth=False)
elif len(coords) == 2:
    x, y = coords
    canvas.create_oval(x-2, y-2, x+2, y+2, fill='#56dfad', outline='')
for threshold, color, name in [(5, '#e66a62', 'detect 5cm'), (40, '#e1b55b', 'release 40cm')]:
    if threshold <= max_d:
        y = bottom - threshold / max_d * (bottom-top)
        canvas.create_line(left, y, right, y, fill=color, dash=(5, 5))
        canvas.create_text(right - 5, y - 8, text=name, fill=color,
            anchor='e', font=('Arial', 9))

def poll(self):
    if self.serial_port:
        try:
            # Limit bytes processed per UI update to keep controls responsive.
            available = self.serial_port.in_waiting
            if available:
                data = self.serial_port.read(min(available, 4096))
                self.buffer += data.decode('ascii', errors='ignore')
                if len(self.buffer) > 16384:
                    self.buffer = self.buffer[-8192:]
                lines = self.buffer.split("\n")
                self.buffer = lines.pop()
                for line in lines:
                    packet = parse_telemetry(line)
                    if packet:
                        self.process_packet(packet)
            if self.last_packet_at is not None and time.monotonic() - self.last_packet_at > 3:
                self.connection_text.set("WARNING • TELEMETRY STALE")
                self.warning_text.set('No valid DATA packet received for 3 seconds.')
        except (serial.SerialException, OSError) as exc:
            self.disconnect(silent=True)
            self.connection_text.set('CONNECTION LOST')
            self.warning_text.set(str(exc))
            self.append_event(f'Serial error: {exc}')
    self.root.after(50, self.poll)

def close(self):
    if self.logging:
        self.stop_log()
    self.disconnect(silent=True)
    self.root.destroy()

```

```
if __name__ == '__main__':  
    app = tk.Tk()  
    Dashboard(app)  
    app.mainloop()
```