EN.605.715.81.FA25 Project 2 - Arduino RT Temperature Transmission

Kelly “Scott” Sims

Github Link:
https://github.com/Mooseburger1/johns_hopkins_masters/tree/main/software_development_for_rt_embedded_systems_EN_605_715_81/project_2/temperature/src

YouTube Link: https://youtu.be/etQnZJuAaos

Requirements

Project Key Requirements

1. Using a Round Robin with Interrupts design, get the Arduino to capture the temperature and convert to Fahrenheit
2. After the temperature has stabilized, start recording the Arduino temperature at a periodic rate of around 10s at room temperature
3. Transmit the time and temp across a Serial bus like USB to your Host, others could be SPI or I2C if your Host had one of those buses.
4. Export as a comma separated value file, read into a spreadsheet program and plot the temperature vs time

Requirement Alterations

1. Consistent
2. Consistent
3. Data will be transferred via WiFi over UDP. This accounts for the consideration of working from a stationary desktop and the lack of a serial cable that reaches from cold sinks (freezer)
4. Data will be streamed and plotted in real time from the UDP client that establishes connection with the microcontroller

Hardware & Protocol Considerations

Microcontroller: renesas-ra platform (E.g. Arduino R4 Uno)
Communication: WiFi
Transport Protocol: UDP
Sensor: DHT 11 Temp and Humidity sensor

Design

WiFi Connection

WiFi credentials are statically persisted in configuration header as global constants. This allows them to be utilized anywhere throughout the codebase. These credentials are the SSID of the network and its password.

A WiFi Setup library is created that consumes these credentials via a configuration struct, attempts to connect, and reports the IP address of the microcontroller. The configuration struct is the following

```
struct WiFiConfigurations {
  const char* ssid;
  const char* password;
  const StartUpMessage* startup_message;
  const ConnectedMessage* connected_message;
};
```

Where startup_message and connected_message are user defined messages to be displayed when initializing WiFi connection

UDP Connection

After WiFi connection is established, a UDP listener will be initialized. It will listen for and transmit data to a client over port 1234

Clock Sync

The RTC library is utilized to sync the clock on the microcontroller. To keep the dependencies light, the microcontroller will not be responsible for setting the time. It is the client's responsibility to provide the unix timestamp as the first UDP packet to the microcontroller.

It is from this unix timestamp that subsequent data transmissions will reference.

Client Ready

Once clock sync is achieved, the microcontroller will ACK the client by providing two options:

1. Start Temperature Transmission
2. End Temperature Transmission

At any time, the client can send option 2 to sever the connection with the microcontroller. The microcontroller will not begin sending data until option 1 is selected. This grants the client enough time to configure any dependencies it may have in preparation of the RT transmission.

Application State

Application state is managed and controlled by AppState struct. This struct privately holds reference to the current state. This state is guarded by disabling interrupts and prevents concurrent reads and writes (i.e. guards against race conditions).

Valid states are:

```
enum class States {
    UNKNOWN,
    UNINITIALIZED,
    TRANSMITTING,
    CONNECTED,
    READY,
    DONE,
};
```

The microcontroller begins in UNINITIALIZED state. It is at this point that WiFi, UDP, Temperature sensor, and timers are configured and initialized. Any reentrant calls into the setup function will not reinitialize these instances.

State will not transition from UNINITIALIZED until a client has connected and provided a unix timestamp. Once the microcontroller is synced, it then transitions to CONNECTED. This is blocking. No other actions happen until a client connects and provides this timestamp.

Once connected, the microcontroller transmits the options to the client and transitions to a READY state. In this state, the microcontroller does nothing except waits for the client's option.

Once an option is received to start temperature transmission, the microcontroller transitions to the TRANSMITTING state and begins sampling the temp sensor at 10s intervals. These temps are transmitted across port 1234 in a comma separated format to the client. For example:

[UDP] Received: 2025-09-18T12:00:28, 20.90

If the client instead chooses to end the connection, the state transitions to DONE and reenters the setup function. Here, all logic is blocked until a unix timestamp is received from a new client.

Flagging Transmission

The main loop will continue to monitor application state and take the appropriate action. While the state is in a TRANSMITTING state, the main loop continues to monitor a global volatile flag ***readyToReadTemp***. If true, the main loop sets it to false, reads the temperature sensor, and transmits it via UDP to the client.

This global flag is set to true every 10s by an ISR. This ISR is controlled via a FspTimer with a GPT_TIMER type.

Timer Caveats

The sampling interval of 10s is too long for the GPT_TIMER. This is because it is a 16-bit counter. The amount of counts induced by the clock cycles greatly exceeds the 16-bit maximum value of 65k. This is true even with the largest prescaler of 1024. It can however successfully achieve 1s intervals.

Because of this, the ISR will be triggered every 1s and increments a counter. Once this counter reaches 10, it is reset to 0, and sets the global volatile flag ***readyToReadTemp*** to true.

UDP Client

The UDP client is a simple python client that connects to the microcontroller, transmits the unix timestamp, listens for the options, and provides the user input.

Once the option for transmission is selected, the client starts two background threads

1. One for receiving data across the UDP connection
2. One for listening for user inputs to terminate the connection

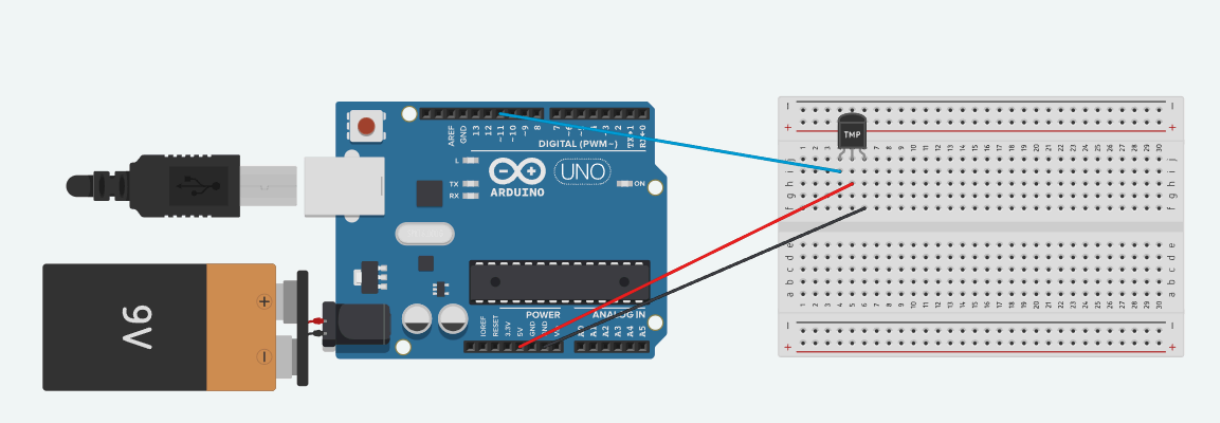
This leaves the main thread free to plot, update, and render the real-time graph when new data arrives.

Wiring

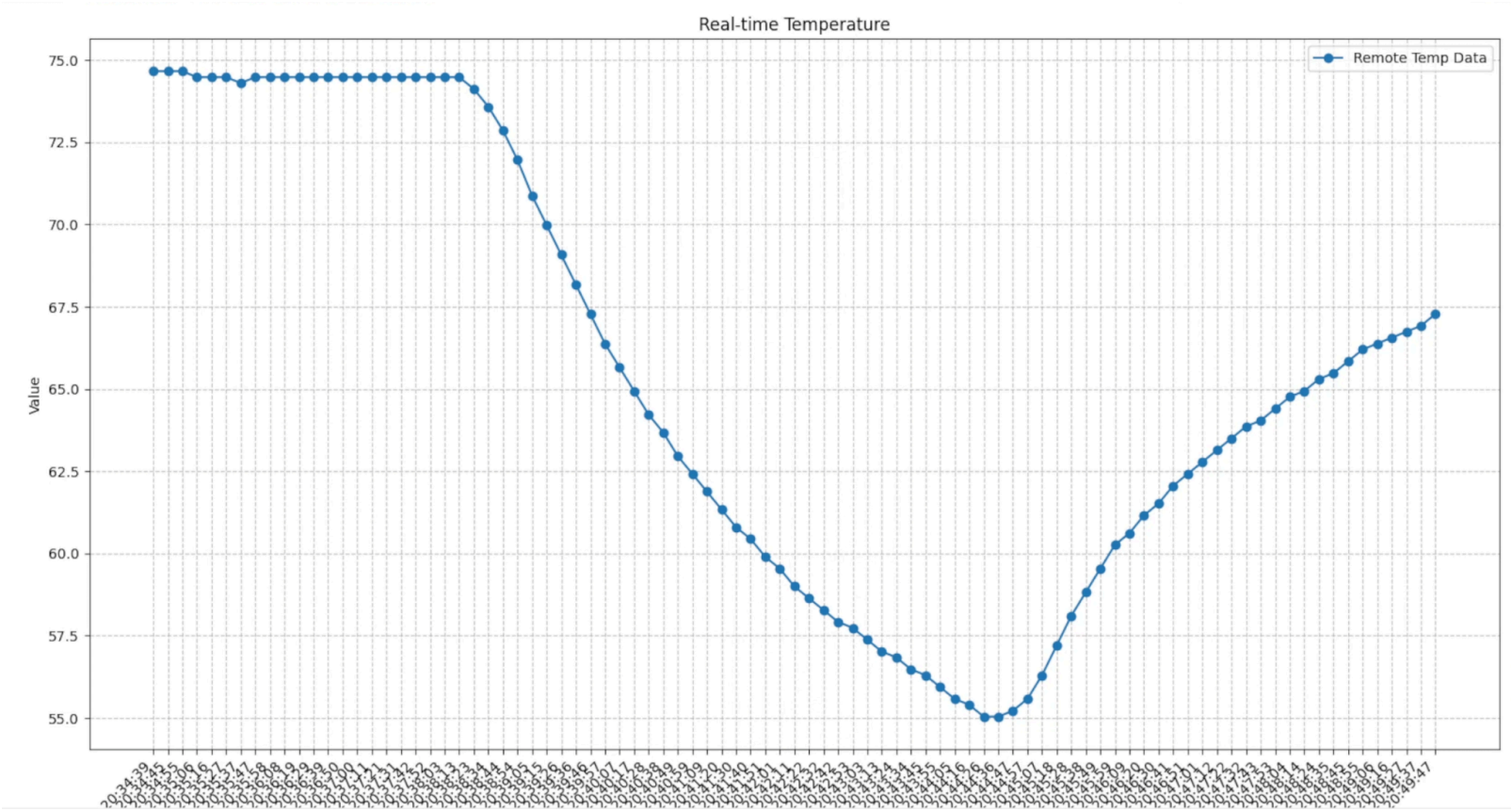
The sensor is a DHT11 temperature and humidity sensor with 3 pins. Where pin…

1. Data pin connected to pin 11 on the microcontroller
2. VCC pin connected to 5v pin on the microcontroller
3. Ground pin connected to the ground pin on the microcontroller

No other wiring is needed since the DHT 11 comes with its own pull-up resistor.



Implementation



```
// main.cpp

#define DHTPIN 11
#define DHTTYPE DHT11

#include <Arduino.h>
#include <DHT.h>
#include <WiFiS3.h>

#include "configurations.h"
#include "RTC.h"
#include "rtc_config.h"
#include "transmit.h"
#include "wifi_setup.h"
#include "FspTimer.h"

const int PORT = 12345;

volatile bool readyToReadTemp = false;
volatile bool stableTemp = false;
volatile uint8_t tickCount = 0;

state::AppState app_state;
DHT dht(DHTPIN, DHTTYPE);
FspTimer temp_timer;
WiFiUDP udp;

wifi_utils::WiFiConfigurations wifi_configs = {
    .ssid=config::SSID,
    .password=config::PASSWORD,
    .startup_message=[](){
      Serial.print("Connecting to SSID: ");
      Serial.println(config::SSID);
    },
    .connected_message=[](){
        Serial.print("Arduino WiFi IP Address: ");
        Serial.println(WiFi.localIP());;
    }
};

void timer_callback(timer_callback_args_t __attribute__((unused)) *p_args) {
  tickCount++;
  if (tickCount >= 10) {
    tickCount = 0;
    readyToReadTemp = true;
  }
}

bool BeginTimer(float rate_hz) {
  uint8_t timer_type = GPT_TIMER;
```

```
  int8_t tindex = FspTimer::get_available_timer(timer_type, true);

  if (tindex < 0) {
    return false;
  }

  if (!temp_timer.begin(TIMER_MODE_PERIODIC, timer_type, tindex, rate_hz, 0.0f,
timer_callback)) {
    Serial.println("begin() failed");
    return false;
  }

  if (!temp_timer.setup_overflow_irq() || !temp_timer.open() || !temp_timer.start()) {
    Serial.println("failed to start timer");
    return false;
  }

  return true;
}


void setup() {
  if (app_state.GetState() == state::States::UNINITIALIZED) {
    Serial.begin(9600);
    dht.begin();

    while (wifi_utils::SetupWiFi(wifi_configs)) {
      Serial.println("Unable to connect to Wifi..."
      "You are currently trapped in an infinite loop!!!");
    }

    udp.begin(PORT);
    Serial.print("UDP Server started on port: ");
    Serial.println(PORT);

    if (!BeginTimer(1.0)) {
    Serial.println("Timer failed to start");
    }
  }

  rtc_config::WaitForClockConfiguration(udp, app_state, state::States::CONNECTED);

}

void loop() {

  if (state::ReadyToTransmitOptions(app_state)) {
    app_state.UpdateState(state::States::READY);
    transmit::TransmitOptions(udp);
    return;
```

```
  }

  if (state::ReadyToTransmitTemp(app_state)) {
    transmit::ListenForOption(udp, app_state);
    return;
  }

  if (state::ClientSignaledDone(app_state)) {
    setup();
    return;
  }

  if (app_state.GetState() == state::States::TRANSMITTING) {
    if (readyToReadTemp) {
      readyToReadTemp = false;

      // Setting readTemperature to true automatically converts temperature from C to F
      float tempF = dht.readTemperature(true);
      Serial.print("tempF ");
      Serial.println(tempF);

      RTCTime curr_time;
      RTC.getTime(curr_time);
      auto payload = curr_time.toString() + ", " + String(tempF, 2);

      transmit::Transmit(udp, {payload.c_str()});
    }
  }
}
```

```
// rtc_config.cc

#include <Arduino.h>
#include <WiFiS3.h>

#include "RTC.h"
#include "state.h"

namespace rtc_config {
namespace {
char udp_packet[255];
int dataLen;

time_t ParseTimeFromUdp(const char* packet) {
  char* end_ptr;
  unsigned long epoch = strtoul(packet, &end_ptr, 10);
  Serial.print("Converted unix timestamp: ");
  Serial.println(epoch);
```

```
  if (*end_ptr != '\0') {
    Serial.println("Error: Non-numeric characters in timestamp");
    return 0;
  }

  if (epoch < 946684800UL || epoch > 4102444800UL) {
    // sanity check ~2000-01-01 to ~2100-01-01
    Serial.println("Error: Epoch timestamp out of valid range");
    return 0;
  }

  return (time_t)epoch;
}

} // namespace

void WaitForClockConfiguration(WiFiUDP& udp,
                    state::AppState& app_state, state::States transition_state) {

  char udp_packet[256];

  auto curr_state = app_state.GetState();
  while (curr_state == state::States::UNINITIALIZED || curr_state == state::States::DONE) {
    Serial.println("Waiting for UDP Connection...");

    if (udp.parsePacket()) {
      int dataLen = udp.available();
      udp.read(udp_packet, 255);
      udp_packet[dataLen] = '\0';
      Serial.print("Received epoch: ");
      Serial.println(udp_packet);

      time_t config_epoch = ParseTimeFromUdp(udp_packet);

      if (config_epoch != 0) {
        RTC.begin();
        RTCTime config_time_from_udp(config_epoch);
        RTC.setTime(config_time_from_udp);
        Serial.println("RTC time has been set.");

        udp.beginPacket(udp.remoteIP(), udp.remotePort());
        udp.print("OK");
        udp.endPacket();

        app_state.UpdateState(transition_state);
        return;
      }
    }
    delay(500);
  }
```

```
}

} // rtc_config
```

```
// state.h

#ifndef STATE_H
#define STATE_H

#include <Arduino.h>

namespace state {

enum class States {
    UNKNOWN,
    UNINITIALIZED,
    TRANSMITTING,
    CONNECTED,
    READY,
    DONE,
};

class AppState {
  public:
   explicit AppState() = default;

   void UpdateState(States state) {
    noInterrupts();
    state_ = state;
    interrupts();
   }

   void UpdateTransmitInterval(int interval) {
    noInterrupts();
    transmit_interval_ = interval;
    interrupts();
   }

   int GetTransmitInterval() const {
    int curr_interval;
    noInterrupts();
    curr_interval = transmit_interval_;
    interrupts();

    return curr_interval;
   }

   States GetState() {
```

```
    States return_state;
    noInterrupts();
    return_state = state_;
    interrupts();

    return return_state;
   }

   static char* GetStateString(States state) {
    switch (state) {
      case States::CONNECTED:
        return "CONNECTED";
      case States::DONE:
        return "DONE";
      case States::READY:
       return "READY";
      case States::TRANSMITTING:
        return "TRANSMITTING";
      case States::UNINITIALIZED:
        return "UNINITIALIZED";
      default:
        return "UNKNOWN";
    }
   }

   char* GetStateString() {
    return GetStateString(GetState());
   }

 private:
  States state_ = States::UNINITIALIZED;
  int transmit_interval_ = 10;
};

inline bool ReadyToTransmitOptions(AppState& app_state) {
  return app_state.GetState() == States::CONNECTED;
};

inline bool ReadyToTransmitTemp(AppState& app_state) {
  return app_state.GetState() == States::READY;
};

inline bool ClientSignaledDone(AppState& app_state) {
  return app_state.GetState() == States::DONE;
}

} // state

#endif STATE_H
```

```cpp
// transmit.cc

#include <Arduino.h>
#include <WiFiS3.h>

#include "state.h"

namespace transmit {

void Transmit(WiFiUDP& udp, std::initializer_list<const char*> messages) {
    udp.beginPacket(udp.remoteIP(), udp.remotePort());
    for (const char* msg : messages) {
        udp.print(msg);
    }
    udp.endPacket();
}

void TransmitOptions(WiFiUDP& udp) {
    Serial.println("Transmitting options");
    Transmit(udp, {"Menu Options:\n",
                   "1. Start Temperature Transmission\n",
                   "2. End Temperature Transmission\n"});
};

void ListenForOption(WiFiUDP& udp, state::AppState& app_state) {
    state::States curr_state = app_state.GetState();
    if (curr_state != state::States::READY) {
        Serial.print("Current state: ");
        Serial.print(app_state.GetStateString(curr_state));
        Serial.println(" is not valid state for listening for options");

        return;
    }

    char udp_packet[256];

    if (udp.parsePacket()) {
        int dataLen = udp.available();
        udp.read(udp_packet, 255);
        udp_packet[dataLen] = '\0';

        Serial.print("Received option ");
        Serial.println(udp_packet);

        char* end_ptr;
        unsigned int option = strtol(udp_packet, &end_ptr, 10);

        if (option == 1) {
            app_state.UpdateState(state::States::TRANSMITTING);
```

```
            Transmit(udp, {"Option 1 Accepted"});
            Serial.println("Option 1 Selected");
            return;
        }

        if (option == 2) {
            app_state.UpdateState(state::States::DONE);
            Transmit(udp, {"Option 2 Accepted"});
            Serial.println("Option 2 Selected");
            return;
        }

        Transmit(udp, {"Invalid option provided: ", udp_packet});
        return;
    }
};

} // transmit
```

```
# client_app.py

import threading
import time
import socket
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation


from data_handler import DataManager
from networking import setup_socket, send_unix_time, receive_data
from plotting import setup_plot, update_plot

# NOTE: SET THE IP ADDRESS TO WHATEVER THE MICROCONTROLLER OUTPUTS
HOST = "192.168.1.37"
PORT = 12345
PLOT_INTERVAL_MS = 1000

stop_receiving = threading.Event() # Use a thread-safe Event for stopping
user_input_value = None

def input_thread(stop_event):
    """Thread to get user input (e.g., '2' to stop) without blocking the plot."""
    global user_input_value
    print("\n--- Input Thread Started ---")
    while not stop_event.is_set():
        try:
            user_input_value = input().strip()
            if user_input_value == "2":
                stop_event.set()
```

```
                print("Stopping data reception and closing plot.")
                break
        except EOFError:
            stop_event.set()
            break
        except Exception as e:
            if not stop_event.is_set():
                print(f"Input thread error: {e}")
                stop_event.set()
                break

def main():
    # 1. Initialize data manager (handles lists and lock)
    data_manager = DataManager()

    # 2. Setup socket
    my_socket = setup_socket(PORT)
    if not my_socket:
        return

    # 3. Initial Communication Workflow
    print("Starting initial communication...")
    my_socket.settimeout(1)
    while True:
        if send_unix_time(my_socket, HOST, PORT):
            break
        else:
            print("Retrying time sync...")
            time.sleep(2)

    print("Waiting for menu options from server...")
    my_socket.settimeout(None) # Temporarily block until menu is received
    try:
        data, _ = my_socket.recvfrom(1024)
        response = data.decode().strip()
        print(f"Received response from server: {response}")
    except Exception as e:
        print(f"Error receiving menu: {e}")
        my_socket.close()
        return

    my_socket.settimeout(1) # Restore timeout

    option_input = input("Provide option choice (e.g., '1' for data, '2' to exit): ").strip()
    my_socket.sendto(option_input.encode(), (HOST, PORT))
    if option_input == "2":
        my_socket.close()
        return

    # 4. Start threads
```

```
    receiver_thread = threading.Thread(
        target=receive_data,
        args=(my_socket, data_manager, stop_receiving),
        daemon=True
    )
    receiver_thread.start()

    input_thread_obj = threading.Thread(
        target=input_thread,
        args=(stop_receiving,),
        daemon=True
    )
    input_thread_obj.start()

    # 5. Setup Matplotlib plot
    fig, ax, line = setup_plot()

    print("\n--- Real-time Plotting Started ---")
    print(f"Receiving data from {HOST}:{PORT}. Type '2' + Enter to stop.")

    # 6. Start animation
    # Use lambda to pass data_manager into the update_plot function
    ani = FuncAnimation(
        fig,
        lambda frame: update_plot(frame, ax, line, data_manager),
        interval=PLOT_INTERVAL_MS,
        cache_frame_data=False
    )

    # 7. Show plot and cleanup
    try:
        plt.show()
    except Exception as e:
        print(f"Plotting error: {e}")
    finally:
        stop_receiving.set()
        print("Plotting ended. Cleaning up resources.")
        if receiver_thread.is_alive():
            receiver_thread.join(timeout=2)
        my_socket.close()

if __name__ == "__main__":
    main()
```

```
# data_handler.py

import threading
from datetime import datetime
```

```
class DataManager:
    """Manages the shared data points and timestamps in a thread-safe manner."""
    def __init__(self):
        self.data_points = []
        self.timestamps = []
        self.lock = threading.Lock()

    def parse_and_add(self, csv_string):
        """Parses the 'date, value' string and adds data to the lists."""
        try:
            # 1. Split the string
            parts = [p.strip() for p in csv_string.split(',')]
            if len(parts) != 2:
                print(f"Warning: Data format error. Expected 'DATE, VALUE', got:
{csv_string}")
                return

            date_str, value_str = parts[0], parts[1]

            # 2. Convert value to float
            numeric_value = float(value_str)

            # 3. Parse timestamp and format for graph label
            dt_object = datetime.fromisoformat(date_str)
            time_label = dt_object.strftime("%H:%M:%S")

            # Acquire lock before modifying shared lists
            with self.lock:
                self.data_points.append(numeric_value)
                self.timestamps.append(time_label)

        except ValueError as e:
            print(f"Warning: Data parsing error ({e}). Ignoring message: {csv_string}")
        except Exception as e:
             print(f"Unexpected error during data parsing: {e}")

    def get_data(self):
        """Returns copies of the data lists in a thread-safe manner."""
        with self.lock:
            # Return copies to prevent external modification during plot drawing
            return self.data_points[:], self.timestamps[:]
```

```
# networking.py

import socket
import time
from data_handler import DataManager
from datetime import datetime
```

```python
def setup_socket(port):
    """Initializes and binds the UDP socket."""
    my_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        # Bind the socket to the local port to listen for incoming UDP data
        my_socket.bind(('', port))
        return my_socket
    except OSError as e:
        print(f"Error binding socket: {e}. Check if port {port} is already in use.")
        return None

def send_unix_time(my_socket, host, port):
    """Sends current Unix time to the server for synchronization and waits for 'OK'."""
    unix_time = int(time.time())
    payload = str(unix_time).encode()
    print(f"Sending current Unix time: {unix_time}")
    my_socket.sendto(payload, (host, port))

    try:
        # Wait for "OK" response from the server
        data, _ = my_socket.recvfrom(1024)
        response = data.decode().strip()
        print(f"Received response from server: {response}")
        return response == "OK"
    except socket.timeout:
        # This socket.timeout is handled in the calling function (client_app.py)
        return False

def receive_data(my_socket, data_manager, stop_event):
    """Thread to continuously receive UDP data and parse the 'date, value' CSV string."""
    while not stop_event.is_set():
        try:
            # Set a small timeout for the thread to check the stop_event periodically
            my_socket.settimeout(0.1)
            data, _ = my_socket.recvfrom(1024)
            if data:
                csv_string = data.decode().strip()
                print(f"[UDP] Received: {csv_string}")
                data_manager.parse_and_add(csv_string)

        except socket.timeout:
            continue
        except Exception as e:
            if not stop_event.is_set():
                print(f"Error in receive_data: {e}")
            break
```

```python
# plotting.py

import matplotlib.pyplot as plt
from data_handler import DataManager # Not strictly needed here, but good practice

def setup_plot():
    """Initializes and configures the Matplotlib plot."""
    fig, ax = plt.subplots(figsize=(15, 8))

    # 'line,' unpacks the single Line2D object returned by ax.plot
    line, = ax.plot([], [], marker='o', linestyle='-', label='Remote Data')

    ax.set_xlabel("Time")
    ax.set_ylabel("Value")
    ax.set_title("Real-time Temperature")
    ax.legend()
    ax.grid(True, linestyle='--', alpha=0.7)
    plt.tight_layout()

    return fig, ax, line

def update_plot(frame, ax, line, data_manager: DataManager):
    """Function called periodically by FuncAnimation to update the plot."""

    # Get thread-safe copies of data
    data_points, timestamps = data_manager.get_data()

    if not data_points:
        return line,

    # X-data is the index (0, 1, 2, ...), Y-data is the collected values
    x_data = list(range(len(data_points)))
    y_data = data_points

    line.set_data(x_data, y_data)

    # Rescale axes automatically
    ax.relim()
    ax.autoscale_view()

    # Update x-axis labels to show timestamps
    ax.set_xticks(x_data)
    ax.set_xticklabels(timestamps, rotation=45, ha='right')

    # Return the line artist (required by FuncAnimation)
    return line,
```