

Project Task 1: Preparing the Embedded Devices

Gabriel Frank, Ryan Botwinick, Jacob Frankel

[GitHub Repository](#)

Part A

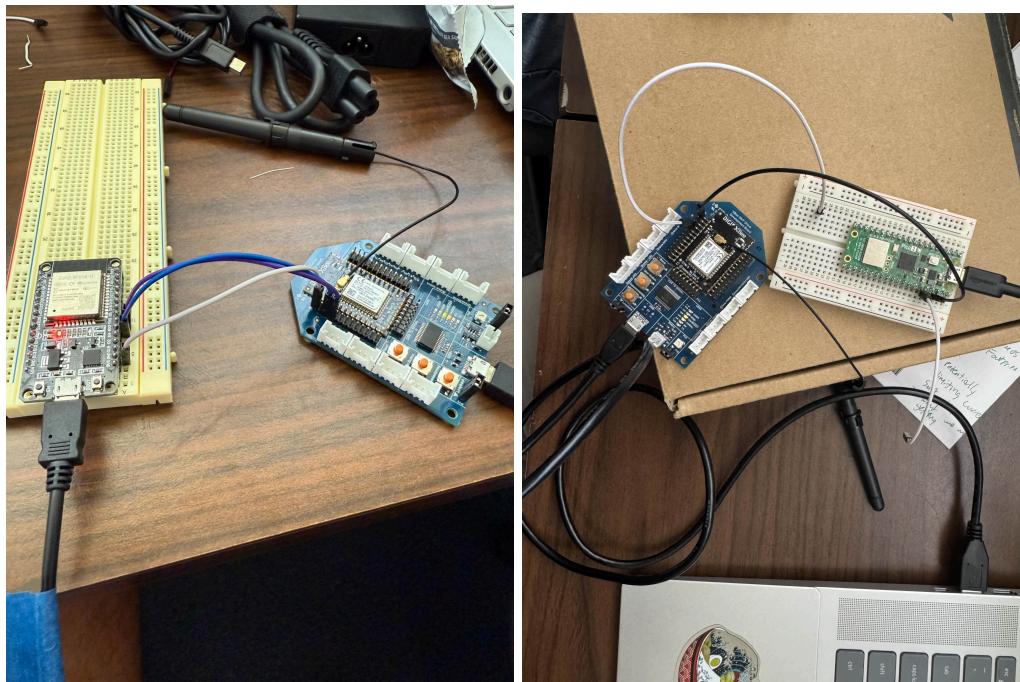
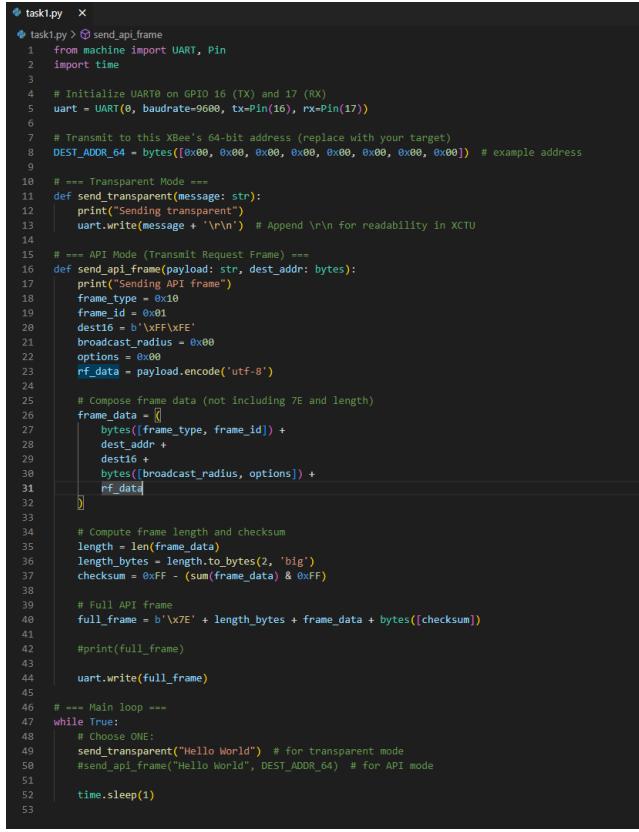


Figure 1: Embedded devices and XBee radio connections

Part B


```

❶ task1.py > ⌂ send_api.frame
❷ task1.py > 
1   from machine import UART, Pin
2   import time
3
4   # Initialize UART0 on GPIO 16 (TX) and 17 (RX)
5   uart = UART(0, baudrate=9600, tx=Pin(16), rx=Pin(17))
6
7   # Transmit to this XBee's 64-bit address (replace with your target)
8   DEST_ADDR_64 = bytes([0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00])
9
10  # === Transparent Mode ===
11  def send_transparent(message: str):
12      print("Sending transparent")
13      uart.write(message + '\r\n') # Append \r\n for readability in XCTU
14
15  # === API Mode (Transmit Request Frame) ===
16  def send_api_frame(payload: str, dest_addr: bytes):
17      print("Sending API frame")
18      frame_type = 0x10
19      frame_id = 0x01
20      dest16 = b'\xFF\xFE'
21      broadcast_radius = 0x00
22      options = 0x00
23      rf_data = payload.encode('utf-8')
24
25      # Compose frame data (not including 7E and length)
26      frame_data = [
27          bytes([frame_type, frame_id]) +
28          dest_addr +
29          dest16 +
30          bytes([broadcast_radius, options]) +
31          rf_data
32      ]
33
34      # Compute frame length and checksum
35      length = len(frame_data)
36      length_bytes = length.to_bytes(2, 'big')
37      checksum = 0xFF - (sum(frame_data) & 0xFF)
38
39      # Full API frame
40      full_frame = b'\x7E' + length_bytes + frame_data + bytes([checksum])
41
42      #print(full_frame)
43
44      uart.write(full_frame)
45
46  # === Main loop ===
47  while True:
48      # Choose ONE:
49      #send_transparent("Hello World") # for transparent mode
50      #send_api_frame("Hello World", DEST_ADDR_64) # for API mode
51
52      time.sleep(1)
53

```

Figure 2: Screenshot of the script for sending the “Hello World” text

Part C

```

1 #define TXD1 17
2 #define RXD1 16
3 #define USE_API_MODE true // change to change the mode sending in
4
5 uint8_t dest_addr[8] = { 0x00, 0x13, 0xA2, 0x00, 0x41, 0x0A, 0x0C, 0xF1 }; // MAC for sink node
6
7 // function to configure and send the API mode, used packet from Zigbee
8 void send_api_frame(const char* payload) {
9     uint8_t frame_type = 0x10;
10    uint8_t frame_id = 0x01;
11    uint8_t dest16[2] = { 0xFF, 0xFF };
12    uint8_t broadcast_radius = 0x00;
13    uint8_t options = 0x00;
14
15    size_t payload_len = strlen(payload);
16    size_t frame_data_len = 14 + payload_len;
17
18    uint8_t frame[100]; // large enough for a big frame
19
20    // header information
21    frame[0] = 0x7E;
22    frame[1] = (frame_data_len >> 8) & 0xFF;
23    frame[2] = frame_data_len & 0xFF;
24
25    size_t i = 3;
26    frame[i++] = frame_type;
27    frame[i++] = frame_id;
28
29    // address information
30    for (int j = 0; j < 8; j++) frame[i++] = dest_addr[j];
31    frame[i++] = dest16[0];
32    frame[i++] = dest16[1];
33    frame[i++] = broadcast_radius;
34    frame[i++] = options;
35
36    // writes the message to the frame
37    for (int j = 0; j < payload_len; j++) frame[i++] = payload[j];
38
39    // help with error control
40    uint8_t checksum = 0;
41    for (int j = 3; j < i; j++) checksum += frame[j];
42    frame[i++] = 0xFF - checksum;
43
44    // writes message
45    Serial1.write(frame, i);
46 }
47
48 void send_transparent(const char* message) {
49     // sends in transparent mode
50     Serial1.println(message);
51 }
52
53
54 void setup() {
55     // sets up baud rate and serial communication
56     Serial1.begin(9600, SERIAL_NORM, RXD1, TXD1); // 1 stop and start bit, 8 data, no parity, set Rx and Tx pins
57     delay(1000);
58 }
59
60 // CHANGE MODE AT THE TOP
61 void loop() {
62     if (USE_API_MODE) {
63         send_api_frame("2025"); // case for api frame
64     } else {
65         send_transparent("2025"); // case for transparent mode
66     }
67     delay(1000); // delay for visibility
68 }
69
70

```

Figure 3: Screenshot of the script for sending the “2025” number**Part D**

To create the script to send text our team used a PiPico with MicroPython. The functions used to send in transparent mode used the string parameter. Transparent mode is the simpler of the two focusing on writing the raw data to the Xbee which allows for the direct writing of the string. When working in API mode we passed both the message as a string and the MAC address of the sink node to properly create a packet for the Xbee. When working with the ESP32 similar processes were applied for the transparent and API modes, but the data was sent in character format. This allowed for easier string manipulation in the Arduino C++ code. In both implementations UART was used to communicate data from the ESP32 and PiPico to the sink Xbee.

Part E

In transparent mode, the script sends raw data directly over UART without any additional formatting or structure. Whatever string is passed (e.g., "Hello World") is transmitted byte-for-byte as-is. This mode behaves like a basic serial passthrough, making it simple but limited in terms of control and addressing. In contrast, API mode requires the data to be wrapped in a well-defined frame structure that the XBee understands. This includes a start delimiter ('0x7E'), a two-byte length field, a frame type ('0x10' for Transmit Request), a frame ID, a 64-bit destination address, a 16-bit address (typically '0xFFFF' if unknown), broadcast radius, options byte, the payload itself, and a final checksum byte calculated as '0xFF' minus the 8-bit sum of all bytes from the frame type to the end of the payload. This structured approach allows for features like addressing, acknowledgments, and delivery status, which are not possible in transparent mode.

Part F

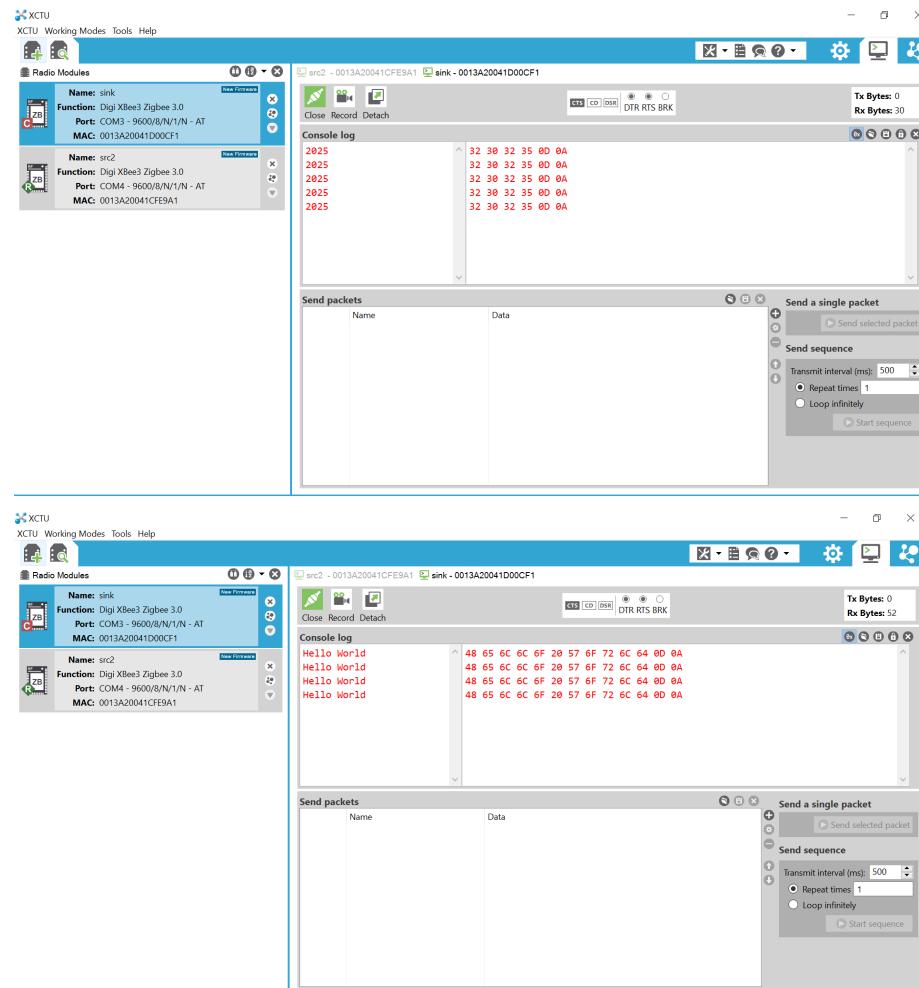


Figure 4: Screenshots of XCTU software receiving data from the embedded devices in Transparent mode

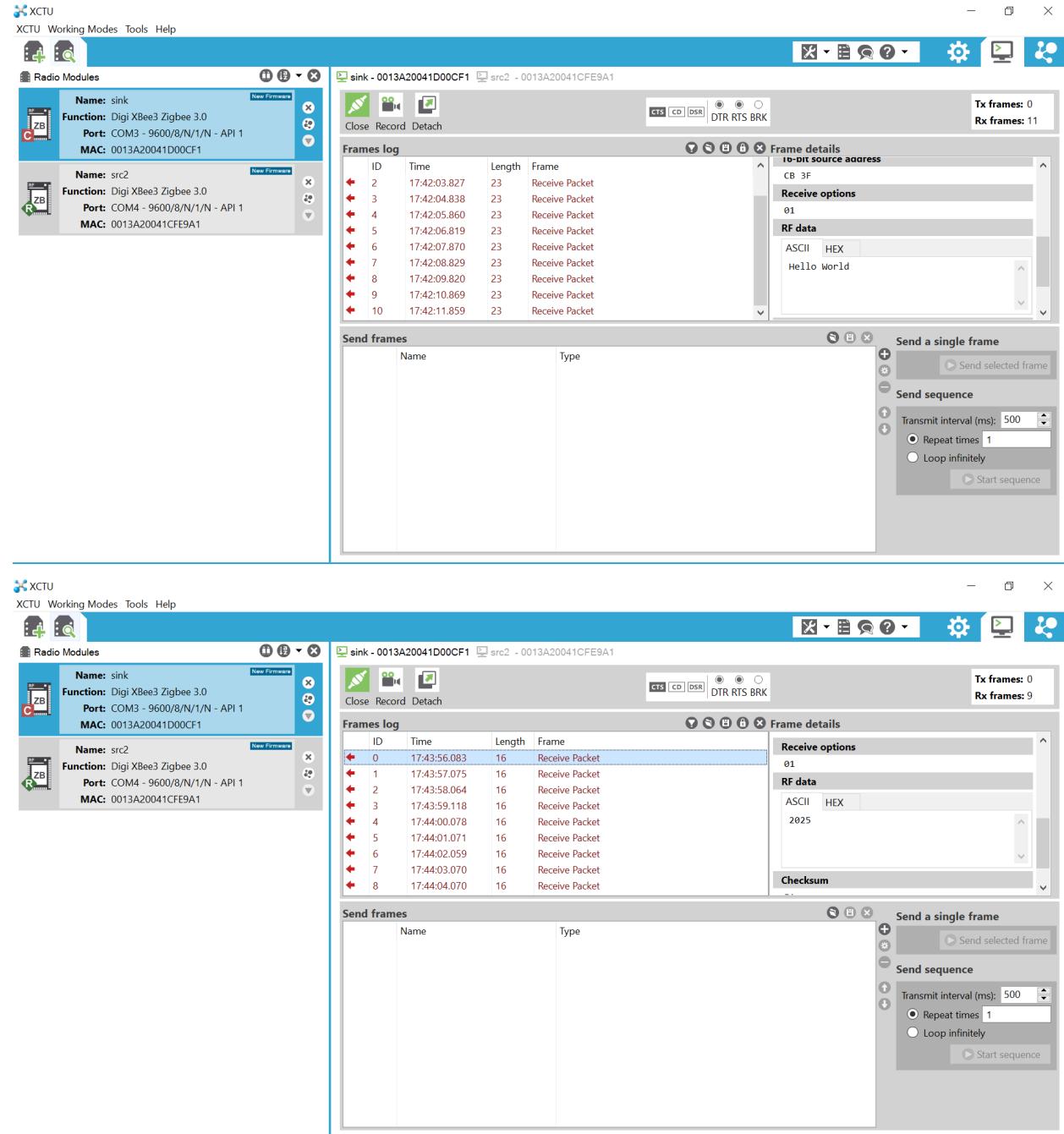
Part G

Figure 5: Screenshots of XCTU software receiving data from the embedded devices in API mode

Part H

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows open files: Welcome, transparent_logger.py, and api_logger.py.
- Editor:** Displays the code for `transparent_logger.py`. The code reads from `COM3` and prints received messages. It includes a `main()` function and a check for the `__name__` variable.
- Terminal:** Shows the output of running the script in terminal mode, indicating it's listening on `COM3` in transparent mode. Subsequent lines show the script receiving and printing "Hello World" messages.
- Status Bar:** Shows the current file is `transparent_logger.py`, line 23, column 1, and other status indicators like `Spaces: 4`, `UTF-8`, and `Python 3.11.0`.

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. The top menu bar includes File, Edit, Selection, View, Go, Run, and others. The left sidebar has sections for Explorer, Open Editors, and Final, with files like transparent_logger.py, api_logger.py, and api_logger.py listed. The main editor area displays the code for transparent_logger.py:

```
def main():
    while True:
        try:
            line = ser.readline().decode('utf-8').strip()
            if line:
                print(f"[RECV] {line}")
        except KeyboardInterrupt:
            print("\n[INFO] Stopped by user.")
            break
        except Exception as e:
            print(f"[ERROR] {e}")

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

Below the editor, the terminal tab is active, showing the command line output:

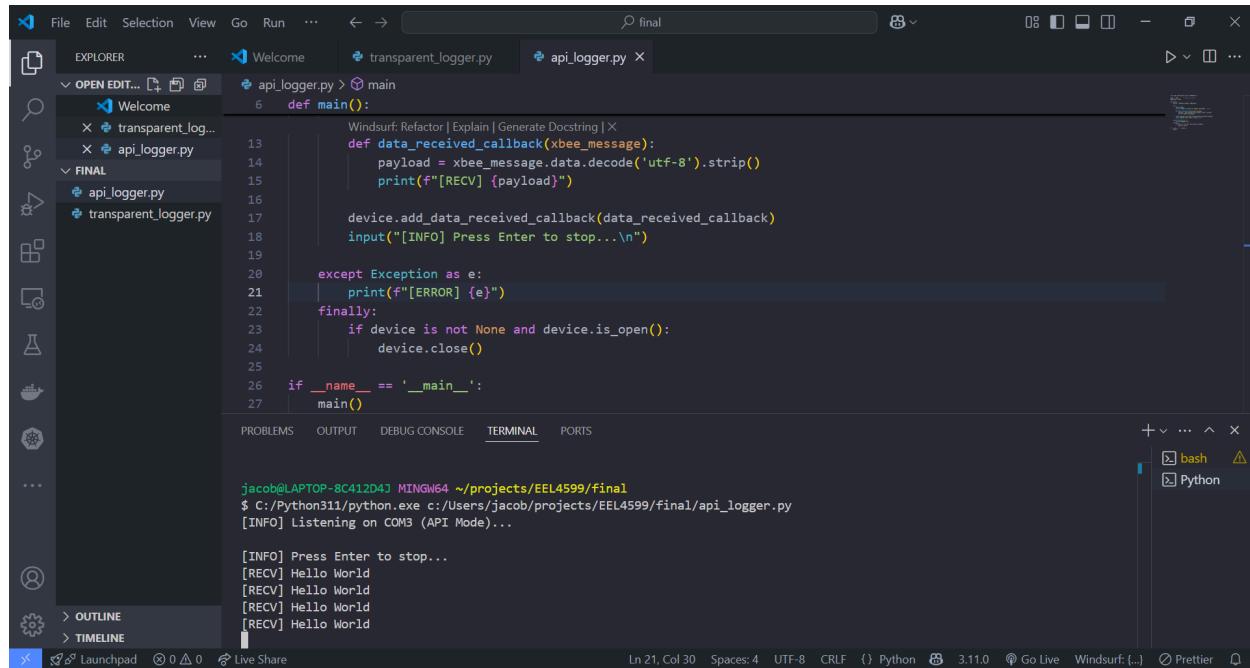
```
jacob@LAPTOP-8C412D43 MINGW64 ~/projects/EEL4599/final
$ c:/Python311/python.exe c:/Users/jacob/projects/EEL4599/final/transparent_logger.py
[INFO] Listening on COM3 (Transparent Mode)...

[RECV] 2025
[RECV] 2025
[RECV] 2025
[RECV] 2025
[RECV] 2025
[RECV] 2025
```

The bottom status bar indicates the code is at Line 23, Column 1, with 3.11.0 Python and Windurf (...) status icons.

Figure 6: Screenshots of the Python script receiving data from embedded devices in Transparent mode

Part I



The screenshot shows the Visual Studio Code interface with two windows side-by-side. Both windows display the same Python script, `api_logger.py`, which is designed to receive data from an Xbee module in API mode.

```

def main():
    Windsurf: Refactor | Explain | Generate Docstring | X
    def data_received_callback(xbee_message):
        payload = xbee_message.data.decode('utf-8').strip()
        print(f"[RECV] {payload}")

        device.add_data_received_callback(data_received_callback)
        input("[INFO] Press Enter to stop...\n")

    except Exception as e:
        print(f"[ERROR] {e}")
    finally:
        if device is not None and device.is_open():
            device.close()

    if __name__ == '__main__':
        main()

```

The terminal output in both windows shows the script running and receiving multiple "Hello World" messages from the Xbee device:

```

jacob@LAPTOP-8C412D4J MINGW64 ~/projects/EEL4599/final
$ C:/Python311/python.exe c:/Users/jacob/projects/EEL4599/final/api_logger.py
[INFO] Listening on COM3 (API Mode)...
[INFO] Press Enter to stop...
[RECV] Hello World

```

Figure 7: Screenshots of the Python script receiving data from embedded devices in API mode

Part J

We used only a single script for receiving text AND numbers. This is because the data transmitted from the embedded devices to the source nodes is done through UART, where raw serial data is sent to the Xbees. At this stage there is no difference between text and numbers. In essence, all text is numbers. Once the raw, non-decoded numerical data is sent to the sink, the Python script we use decodes that numerical data using UTF-8, which is a Unicode standard to convert these numbers into text. Every symbol has a UTF-8 counterpart, even numbers. Thus, regardless of whether text or numbers were sent, they get decoded all the same by the Python script.