Summer Workshop on High-Altitude Platforms

Author: Tali Chan
Department of Applied Physics, National Pingtung University
In collaboration with: STSP Administration, National Sun Yat-sen University

Date: July 2025

Contents

Chapter 1

Hardware Components for Weather Balloon

In this chapter, we introduce the hardware setup required for the IoT balloon experiment. All essential devices and accessories are provided by the instructor to each student, while basic peripherals such as monitor, keyboard, and mouse are already available in the computer classroom.

Category	**Items**
Provided by Instructor	Raspberry Pi Zero 2 W UPS HAT (C) IoT Node (A) MicroSD Card HDMI Male to Mini HDMI Male Cable Micro USB Charging Cable Micro USB Male to USB-A Female Adapter USB Hub Fishing Line Latex Balloon Nitrogen Gas Cylinder
Available in Classroom	Monitor, Keyboard, Mouse

Table 1.1: Hardware used in the course

Chapter 2

Weather Balloon Basics

2.1 Fundamental Structure

A weather balloon (also known as a *scientific sounding balloon*) is a balloon designed to carry scientific instruments into the stratosphere to collect data on atmospheric pressure, temperature, humidity, and wind speed. It typically ascends to altitudes of 20–40 km, where it eventually bursts due to low pressure.

The fundamental structure of a weather balloon consists of:

- **Envelope:** The balloon itself, made of latex or synthetic material, filled with lifting gas.
- **Payload:** The onboard scientific instruments such as sensors, GPS modules, microcontrollers, and data loggers.
- **Parachute:** Ensures the safe descent of the payload after the balloon bursts.
- **Cut-down system:** An optional mechanism that allows controlled separation of the payload from the balloon.

2.2 Payload

The payload refers to the scientific instruments attached to the balloon. These may include environmental sensors, GPS modules, communication devices, and sometimes cameras or experimental equipment. Payload design must consider:

- **Weight:** The payload must remain lightweight to achieve sufficient altitude.
- **Power supply:** Typically provided by batteries or power modules.
- **Tracking:** Real-time GPS tracking is critical for recovery after descent.

Chapter 3

Raspberry Pi Family

The Raspberry Pi is a series of small single-board computers (SBCs) that are cost-effective and highly versatile. They are commonly used in educational projects, prototyping, and embedded systems. The Raspberry Pi family includes:

- Raspberry Pi 4 Model B (up to 8GB RAM)
- Raspberry Pi 3 Series
- Raspberry Pi 2 Series
- Raspberry Pi Zero / Zero W / Zero 2 W (ultra-small form factor, often used in balloon payloads)

3.1 Raspberry Pi Zero / Zero W / Zero 2 W

The Raspberry Pi Zero 2 W is the successor to the Raspberry Pi Zero series, released in 2021. It provides significantly improved performance and is well-suited for lightweight embedded applications, including high-altitude balloon payloads. In our experimental setup, the Raspberry Pi Zero 2 W is chosen as the main onboard computing unit.

Key Specifications

- SoC: Broadcom BCM2710A1 (same as Raspberry Pi 3, quad-core Cortex-A53, 1.0 GHz)
- Memory: 512 MB LPDDR2 RAM
- Wireless: 802.11 b/g/n Wi-Fi, Bluetooth 4.2 / BLE

Interfaces

- Mini HDMI
- Micro-USB OTG
- CSI camera interface
- 40-pin GPIO header (compatible with Raspberry Pi)
- Dimensions: 65 mm $\times$ 30 mm (same as original Zero)

Applications

- Suitable for lightweight IoT projects
- Embedded system prototyping
- Portable scientific and environmental data collection

3.2 Raspberry Pi Operating Systems

Since the Raspberry Pi Zero series comes with limited memory (512 MB), lightweight Linux distributions are generally recommended. Raspberry Pi OS is the official Debian-based distribution optimized for Raspberry Pi hardware.

3.2.1 Raspberry Pi OS Lite

The Lite version of Raspberry Pi OS is recommended for the Zero series.

Features:

- Command-line only (no desktop environment)
- Based on Debian Linux
- Small memory footprint, optimized for embedded devices
- Provides access to package management (APT)

Usage:

- Suitable for Python programming, SSH access, and GPIO control
- Commonly used in IoT and LoRa-based applications
- Ideal for lightweight scientific experiments

3.2.2 Raspberry Pi OS with Desktop

The desktop version of Raspberry Pi OS is not recommended for the Zero series due to limited memory resources. It includes the LXDE desktop environment and is more suitable for Raspberry Pi 3, 4, or 5, where higher RAM capacity and performance are available. The graphical interface allows for easier interaction but comes at the cost of increased system overhead.

3.2.3 PiCore (Tiny Core Linux for Pi)

PiCore is an extremely lightweight Linux distribution (approximately 50 MB). It is designed for embedded applications and offers a minimal footprint.

Features:

- Minimal system resource usage
- Extremely fast boot time
- Highly modular, allowing users to add only the required components

3.2.4 Installing Raspberry Pi OS

All Raspberry Pi OS variants can be installed using the **Raspberry Pi Imager** tool. The steps are as follows:

1. Download and install the Raspberry Pi Imager (choose the Raspberry Pi Zero 2 option if applicable).
2. Prepare a microSD card (Class 10 recommended).
3. Flash the Raspberry Pi OS image onto the card using the Imager.
4. Insert the microSD card into the Raspberry Pi and power it on.

3.2.5 Running Linux on a Windows Host

If only a Windows computer is available, Linux can still be run in a virtualized environment using **VirtualBox**.

Key points about VirtualBox:

- Supports multiple host systems (Windows, macOS, Solaris).
- Can run various guest systems including Linux, BSD, Android x86, DOS, etc.
- Provides flexibility to simulate a Linux environment without needing dedicated hardware.

Chapter 4

Software and System Commands

4.1 Linux Commands

Raspberry Pi commonly uses Debian-based Linux distributions such as Raspberry Pi OS. Understanding basic Linux commands is essential for operating and configuring the system.

Useful References

- Linux file system structure
- Linux common commands
- Package management with `apt`

Examples

- `ls` List files and directories
- `pwd` Print working directory
- `cd` Change directory
- `apt update && apt upgrade` Update packages

4.2 System Permissions and Commands

Certain commands require administrative privileges. On Linux, this is done with `sudo`, which is equivalent to "Run as Administrator" in Windows.

Examples

- `sudo reboot` Restart the system
- `sudo shutdown now` Shut down immediately
- `chmod +x script.sh` Make a script executable
- `chown user file` Change file ownership

4.3 Raspberry Pi GPIO

The General Purpose Input/Output (GPIO) pins allow the Raspberry Pi to interface with sensors, actuators, and communication modules.

4.3.1 Key Notes

- Operates at **3.3V logic level** (not 5V tolerant).
- Use external driver circuits (MOSFETs, relays) for high-power devices.
- Always call `GPIO.cleanup()` to reset pins after use.

4.3.2 Electrical Limitations

- Maximum current per pin: **16 mA**
- Maximum total current across all GPIO pins: **50 mA**
- Exceeding these limits may cause permanent hardware damage.

4.3.3 Enabling I2C and Preparation for IoT Node(A)

Many sensors (e.g., GPS modules, LCD displays) communicate via the I2C interface. Before using IoT node(A), the Raspberry Pi must be configured to enable I2C.

Step 1: Enable I2C

Open the terminal and run:

```
sudo raspi-config
```

Navigate to `Interfacing Options`, then enable I2C.

Step 2: Modify Configuration File

Edit the system configuration:

```
sudo nano /boot/firmware/config.txt
```

Add the following lines:

```
[all]
dtoverlay=sc16is752-i2c
```

Save and exit (`Ctrl+X`, confirm with `Y`, then Enter).

Step 3: Shutdown and Prepare IoT Node(A)

Finally, shut down safely:

```
sudo shutdown now
```

After shutdown, connect the required IoT node(A) hardware to the Raspberry Pi. This ensures that the system is correctly configured and ready for sensor applications.

Chapter 5

Applications and Extensions

5.1 UPS HAT(C)

To ensure stable operation of the Raspberry Pi Zero 2 W in remote or mobile environments, an Uninterruptible Power Supply (UPS) HAT can be used to provide continuous power, often in combination with solar panels.

Power Consumption of Raspberry Pi Zero 2 W

The table below summarizes typical power consumption:

Operating Mode	**Current (mA)**	**Power (W)**
Idle (No Load)	∼260 mA	1.3 W
Normal Operation (Light Load)	350–400 mA	1.75–2.0 W
Medium Load (Wi-Fi + Peripherals)	500–550 mA	2.5–2.75 W
Shutdown State	< 100 mA	0.5 W

Solar Panel Supply Example

When powered by solar panels, the following configuration can be used:

Panel Count	**Voltage / Current**	**Power**
1 Panel	5V / 200 mA	1 W
2 Panels	5V / 400 mA	2 W
3 Panels	5V / 600 mA	3 W

In practice, a single 2.5 W solar panel can support Raspberry Pi Zero W operation in standby or low-load conditions.

Power Management with UPS HAT(C)

To optimize energy usage, the following commands can be applied:

- Disable HDMI output:

```
sudo /usr/bin/tvservice -o
```

- Turn off onboard LEDs:

  ```
  sudo echo none | sudo tee /sys/class/leds/led0/trigger
  ```

- Disable Wi-Fi (if not required):

  ```
  sudo ifconfig wlan0 down
  ```

By reducing unnecessary power consumption, UPS HAT(C) ensures longer runtime and stability of the Raspberry Pi system in field applications.

Monitoring UPS HAT(C) Information

In addition to power management, the UPS HAT(C) can also report real-time voltage, current, and battery percentage. This allows monitoring of system power status during field deployment.

Step 1: Install Required Tools

Open terminal and install the necessary packages:

```
sudo apt-get install p7zip
wget https://files.waveshare.com/upload/4/40/UPS_HAT_C.7z
7zr x UPS_HAT_C.7z -r -o./
cd UPS_HAT_C
python3 INA219.py
```

Step 2: Example Output

After running the script, the output may look as follows:

```
voltage : 4.98 V
current : 0.60 A
power   : 2.99 W
percent : 80 %
```

Step 3: Subsequent Usage

For later usage, simply enter the UPS HAT(C) directory and run the script:

```
cd UPS_HAT_C
python3 INA219.py
```

5.2 GPS Applications

5.2.1 Installing and Configuring GPSD

To use a GPS module with Raspberry Pi, install and configure the GPS daemon (`gpsd`). This service exposes GPS data via a Unix domain socket so that user programs can read it easily.

Step 1: Reset GPS (Optional)

Some GPS modules require a reset via I^{2}C:

```
i2cset -y 1 0x16 0x23 0x40
```

Step 2: Install GPSD

```
sudo apt-get update
sudo apt-get install gpsd gpsd-clients
```

Step 3: Configure GPSD

Edit the configuration file to point to the correct serial device and socket:

```
sudo nano /etc/default/gpsd
```

Use settings similar to:

```
DEVICES="/dev/ttyS0"
GPSD_OPTIONS="-F /var/run/gpsd.sock"
```

5.2.2 Displaying GPS Data on LCD1602

The LCD1602 module connects via the I^{2}C bus to display GPS information (coordinates, time, satellites).

Wiring

- VCC → 3.3V or 5V (Pin 1 or 2)
- GND → Ground (Pin 6)
- SDA → GPIO2 (Pin 3, I^{2}C Data)
- SCL → GPIO3 (Pin 5, I^{2}C Clock)

Step 1: Enable I^{2}C

```
sudo raspi-config
# Interfacing Options -> I2C -> Enable
```

Step 2: Install Dependencies

```
sudo apt update
sudo apt install -y i2c-tools python3-smbus
pip3 install RPLCD --break-system-packages
```

Step 3: Detect I^{2}C Address

Confirm the LCD backpack address (commonly 0x27 or 0x3f):

```
i2cdetect -y 1
```

Step 4: LCD Test Script (Placeholder)

Create a Python script to verify LCD operation:

```
from time import sleep
from RPLCD.i2c import CharLCD

# Adjust address to your module (0x27 or 0x3f)
lcd = CharLCD('PCF8574', address=0x27, port=1, cols=16, rows=2, charmap='A00')

lcd.write_string("Hello, Pi!")
sleep(2)
lcd.clear()
lcd.write_string("LCD1602 I2C OK!")
sleep(2)
lcd.clear()
```

Step 5: GPS + LCD Integration (Placeholder)

Write a Python program that reads GPS data (via gpsd/serial) and updates the LCD1602:

```
import serial, time
from datetime import datetime, timezone, timedelta
from RPLCD.i2c import CharLCD

# --- LCD setup (change address if needed) ---
lcd = CharLCD('PCF8574', address=0x27, port=1, cols=16, rows=2, charmap='A00')

# --- GPS serial (adjust device/baud to your module) ---
ser = serial.Serial('/dev/ttyS0', 115200, timeout=1)

def dm_to_decimal(dm, hemi):
    """Convert NMEA ddmm.mmmm (lat) / dddmm.mmmm (lon) to decimal degrees."""
    if not dm:
        return None
    dm = float(dm)
    deg = int(dm // 100)
    minutes = dm - 100 * deg
    val = deg + minutes / 60.0
    if hemi in ('S', 'W'):
        val = -val
    return val

def read_fix(timeout_s=15):
    """Read lines for up to timeout_s, return (lat, lon, local_time_str, sats).
   """
```

```
    end = time.time() + timeout_s
    lat = lon = None
    local_time_str = "--:--:--"
    sats = "0"

    while time.time() < end:
        line = ser.readline().decode(errors='ignore').strip()
        if not line:
            continue

        if line.startswith(("$GNRMC", "$GPRMC")):
            p = line.split(',')
            # RMC: p[2]=A=valid, p[1]=hhmmss.sss, p[9]=ddmmyy
            if len(p) > 10 and p[2] == "A":
                lat = dm_to_decimal(p[3], p[4])
                lon = dm_to_decimal(p[5], p[6])
                # Local time (UTC+8 example; adjust as needed)
                try:
                    utc_t = datetime.strptime(p[1][:6] + p[9], "%H%M%S%d%m%y").
    replace(tzinfo=timezone.utc)
                    local_t = utc_t.astimezone(timezone(timedelta(hours=8)))
                    local_time_str = local_t.strftime("%H:%M:%S")
                except Exception:
                    pass

        elif line.startswith(("$GPGSV", "$GNGSV")):
            # GSV sentence contains satellite-in-view count at field 3
            parts = line.split(',')
            if len(parts) > 3 and parts[3].isdigit():
                sats = parts[3]

        if lat is not None and lon is not None:
            return lat, lon, local_time_str, sats

    return None, None, local_time_str, sats

def main():
    lcd.clear()
    lcd.write_string("GPS Initializing")
    time.sleep(2)

    while True:
        lat, lon, tstr, sats = read_fix()
        lcd.clear()
        if lat is None or lon is None:
            lcd.write_string("NO FIX  Sats:" + str(sats))
        else:
            lcd.write_string(f"Lat:{lat:.4f}")
            lcd.cursor_pos = (1, 0)
            # 2nd line can include time or sats
            lcd.write_string(f"Lon:{lon:.4f}")
        time.sleep(2)

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        lcd.clear()
```

Setting Auto-Execution at Startup (systemd)

To automatically run the LCD1602 GPS display program at boot using `systemd`:

1) Place the program Assume that the Python script is located at:

```
/home/pi/gps_lcd1602_display.py
```

Make it executable:

```
sudo chmod +x /home/pi/gps_lcd1602_display.py
```

2) Create a systemd service

```
sudo nano /etc/systemd/system/gps_lcd.service
```

Paste the following content:

```
[Unit]
Description=GPS LCD1602 Display Service
After=network.target

[Service]
ExecStart=/usr/bin/python /home/pi/gps_lcd1602_display.py
Restart=always
User=pi
Environment=PYTHONUNBUFFERED=1

[Install]
WantedBy=multi-user.target
```

3) Enable the service

```
sudo systemctl daemon-reload
sudo systemctl enable gps_lcd.service
sudo systemctl start gps_lcd.service
```

5.3 LoRa Applications

5.3.1 LoRa Quick Test: Sender and Receiver

LoRa modules on the Raspberry Pi Zero 2 W can be accessed via I^{2}C (`smbus`). Below are minimal test scripts to verify basic sending/receiving of several bytes.

Sender

Create the file `lora-send.py`:

```python
import time
import smbus
import sys

bus = smbus.SMBus(1)

try:
    # Example payload
    data_list = [170, 85, 165, 90]  # 0xAA, 0x55, 0xA5, 0x5A

    # Write bytes into consecutive registers 0x01..0x04
    for index in range(1, len(data_list) + 1):
        bus.write_byte_data(0x16, index, data_list[index - 1])
        print(f"LORA send data to reg 0x{index:02X}: {data_list[index - 1]}")

    # Trigger TX (example control register)
    bus.write_byte_data(0x16, 0x23, 0x01)

except KeyboardInterrupt:
    sys.exit()
```

Receiver

Create the file `lora-receive.py`:

```python
import time
import smbus
import sys

bus = smbus.SMBus(1)
recv_data = []

try:
    # Check RX flag in status/control register 0x23 (bit 1 as example)
    if bus.read_byte_data(0x16, 0x23) & 0x02:
        # Optionally clear RX flag
        # bus.write_byte_data(0x16, 0x23, 0x00)

        # Read back data from 0x11..0x14
        register_list = [0x11, 0x12, 0x13, 0x14]
        for reg in register_list:
            recv_data.append(bus.read_byte_data(0x16, reg))

        print("Received data:")
        print(recv_data)
    else:
        print("No data Received yet-")

```

```
except KeyboardInterrupt:
    sys.exit()
```

Run the test

Open two terminals, one for each device:

```
# On the sender
python3 lora-send.py

# On the receiver
python3 lora-receive.py
```

Expected output on the receiver (example):

```
Received data: [170, 85, 165, 90]
```

If no frame is captured, you may see the following:

```
No data Received yet-
```

5.3.2 Camera Module 2 Usage

To capture images/videos for later transmission over LoRa:

Install libcamera apps

```
sudo apt install -y libcamera-apps
```

Basic checks

```
# Quick preview
libcamera-hello
```

Capture a still image

```
libcamera-jpeg -o image.jpg
```

Record a short video

```
libcamera-vid -t 10000 -o video.h264
```

5.3.3 Using LoRa to Transmit 160x160 Grayscale Images

The following subsections reserve space for the complete sender/receiver programs. Paste your finalized code into the blocks below when ready.

Sender Program (placeholder)

```python
# sender_with_chunk_counter.py
# Capture 160x160 grayscale, PNG-compress, prepend 4-byte length header,
# then transmit in 4-byte chunks via I2C.
from picamera2 import Picamera2
from PIL import Image
import io, time
import smbus

I2C_ADDR = 0x16
bus = smbus.SMBus(1)

def capture_gray_160():
    picam2 = Picamera2()
    picam2.start()
    time.sleep(2)
    # Save a quick still then convert to 160x160 L (grayscale)
    picam2.capture_file("raw.jpg")
    img = Image.open("raw.jpg").convert("L").resize((160, 160))
    return img

def png_bytes(img):
    bio = io.BytesIO()
    img.save(bio, format="PNG", optimize=True)
    return bio.getvalue()

def make_header_le(n):
    # 4-byte little-endian length header
    return bytes([(n >> (8*i)) & 0xFF for i in range(4)])

def wait_txe_clear():
    # TXE bit clears when ready to accept next frame (example bit 0x01)
    while True:
        status = bus.read_byte_data(I2C_ADDR, 0x23)
        if (status & 0x01) == 0:
            break
        time.sleep(0.05)

def send_chunk4(chunk4):
    # Write up to 4 bytes to regs 0x01..0x04
    for i in range(4):
        val = chunk4[i] if i < len(chunk4) else 0
        bus.write_byte_data(I2C_ADDR, 0x01 + i, val)
        time.sleep(0.005)
    # Trigger TX
    bus.write_byte_data(I2C_ADDR, 0x23, 0x01)

def main():
    img = capture_gray_160()
    payload = png_bytes(img)
    header = make_header_le(len(payload))
    data = header + payload

    # Split to 4-byte chunks
    chunks = [data[i:i+4] for i in range(0, len(data), 4)]
    print(f"[Sender] Total chunks: {len(chunks)}")

    # Send all chunks
    for idx, ch in enumerate(chunks, 1):
        wait_txe_clear()
        send_chunk4(ch)
        print(f"[Sender] Sent chunk {idx}/{len(chunks)}")
        time.sleep(0.01)

    # Reliability: repeat last chunk a few times to help receiver finalize
    last = chunks[-1]
    print("[Sender] Re-sending last chunk for reliability...")
    for r in range(5):
        wait_txe_clear()
        send_chunk4(last)
        time.sleep(0.05)

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

Receiver Program (placeholder)

```
# receiver_stable.py
# Read 4-byte chunks from I2C when RXNE indicates data available.
# First 4 bytes = little-endian payload length. Accumulate and reconstruct PNG.
import io, time
from PIL import Image
import smbus

I2C_ADDR = 0x16
bus = smbus.SMBus(1)

def rx_ready():
    # RXNE bit set means a frame is available (example bit 0x02)
    status = bus.read_byte_data(I2C_ADDR, 0x23)
    return (status & 0x02) != 0

def read_chunk4():
    # Read 4 bytes from regs 0x11..0x14
    regs = [0x11, 0x12, 0x13, 0x14]
    data = bytes(bus.read_byte_data(I2C_ADDR, r) for r in regs)
    # Optionally clear RX flag (module-dependent)
    try:
        bus.write_byte_data(I2C_ADDR, 0x23, 0x00)
    except Exception:
        pass
    return data

def le_u32(b4):
    return b4[0] | (b4[1] << 8) | (b4[2] << 16) | (b4[3] << 24)

def main():
    print("[Receiver] Waiting for data...")
    buf = bytearray()
    expected_len = None
    idle = 0
    IDLE_MAX = 50  # ~ a few seconds

    while True:
        if rx_ready():
            ch = read_chunk4()
            buf.extend(ch)
            idle = 0

            # Parse header as soon as we have >= 4 bytes
            if expected_len is None and len(buf) >= 4:
                expected_len = le_u32(buf[:4])
                print(f"[Receiver] Expecting image size: {expected_len} bytes")

            # If we already received everything (header + payload), stop
            if expected_len is not None and len(buf) >= expected_len + 4:
                print("[Receiver] All data received.")
                break
        else:
            idle += 1
            if idle > IDLE_MAX and expected_len is not None:
                print("[Receiver] No more data coming. Ending.")
                break
            time.sleep(0.06)

    if expected_len is None or len(buf) < expected_len + 4:
        print("[Receiver] ERROR: Incomplete or missing header.")
        with open("debug_output.raw", "wb") as f:
            f.write(buf)
        return

    image_bytes = bytes(buf[4:4+expected_len])

    # Try to decode PNG and save result
    try:
        stream = io.BytesIO(image_bytes)
        img = Image.open(stream)
        img.save("received_final.png")
        print("[Receiver] Image saved: received_final.png")
    except Exception as e:
        print("[Receiver] Failed to decode PNG:", e)
        with open("debug_output.raw", "wb") as f:
            f.write(image_bytes)
        print("[Receiver] Raw image data saved for debug.")

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

5.3.4 Using LoRa to Transmit CPU Temperature

In this section, we demonstrate how to capture the CPU temperature from the Raspberry Pi and transmit it using LoRa. The process consists of two main parts: (1) logging the CPU temperature locally and (2) sending it over LoRa to a receiver.

Step 1: Reading CPU Temperature

On Linux-based systems, CPU temperature can be accessed from the `/sys/class/thermal/thermal_zo` file. Run the following command in terminal to check the value:

```
cat /sys/class/thermal/thermal_zone0/temp >> temprature.dat
```

Step 2: Logging Temperature with a Shell Script

Create a script `log_temperature.sh`:

```
#!/bin/bash
DATAFILE="temperature.dat"

while true; do
  TEMP_RAW=$(cat /sys/class/thermal/thermal_zone0/temp)
  TEMP_C=$(echo "scale=1; $TEMP_RAW / 1000" | bc)
  TIMESTAMP=$(date '+%Y-%m-%d %H:%M:%S')
  echo "$TIMESTAMP - $TEMP_C C" | tee -a $DATAFILE
  sleep 20
done
```

Make the script executable:

```
sudo chmod +x log_temperature.sh
```

Run it:

```
./log_temperature.sh
```

This will record the CPU temperature every 20 seconds.

Step 3: Sending CPU Temperature via LoRa (Sender)

On the sender device, create a Python script `lora-cpu.py`. This script will read CPU temperature, prepare the data, and transmit it via LoRa.

```
# sender_send_temp_chunked.py
# Read CPU temperature and send as text lines via LoRa in 4-byte chunks.
import smbus
import time
```

```python
from datetime import datetime

I2C_ADDR = 0x16
bus = smbus.SMBus(1)

def read_cpu_temp():
    with open("/sys/class/thermal/thermal_zone0/temp", "r") as f:
        raw = int(f.read())
    return raw / 1000.0  # Celsius

def wait_txe_clear():
    # TXE bit cleared (0) => ready to accept next frame
    while True:
        status = bus.read_byte_data(I2C_ADDR, 0x23)
        if (status & 0x01) == 0:
            break
        time.sleep(0.05)

def send_lora_msg(msg):
    data = msg.encode("utf-8")
    chunks = [data[i:i+4] for i in range(0, len(data), 4)]

    for idx, chunk in enumerate(chunks, 1):
        print(f"[Sender] Sending chunk {idx}: {chunk!r}")

        wait_txe_clear()
        for i in range(4):
            val = chunk[i] if i < len(chunk) else 0
            bus.write_byte_data(I2C_ADDR, 0x01 + i, val)
            time.sleep(0.005)

        bus.write_byte_data(I2C_ADDR, 0x23, 0x01)  # trigger send
        time.sleep(0.1)

if __name__ == "__main__":
    while True:
        temp = read_cpu_temp()
        ts = datetime.now().strftime("%H:%M:%S")
        # Add newline as a delimiter so receiver can split lines
        msg = f"{ts} {temp:.1f}C\n"
        send_lora_msg(msg)
        time.sleep(20)
```

Step 4: Receiving CPU Temperature via LoRa (Receiver)

On the receiver device, create a script `lora-cpu-recv.py`. This script will listen on the LoRa bus, capture the transmitted temperature, and display it.

```python
# receiver_receive_temp.py
# Receive 4-byte chunks via LoRa I2C and print text lines split by '\n'.
import smbus
import time

I2C_ADDR = 0x16
bus = smbus.SMBus(1)

def rx_ready():
    # RXNE bit set (1) => a frame is available
    status = bus.read_byte_data(I2C_ADDR, 0x23)
    return (status & 0x02) != 0

def read_chunk4():
    regs = [0x11, 0x12, 0x13, 0x14]
    chunk = bytes(bus.read_byte_data(I2C_ADDR, r) for r in regs)
    # clear RX flag if required by module
    try:
        bus.write_byte_data(I2C_ADDR, 0x23, 0x00)
    except Exception:
        pass
    return chunk

if __name__ == "__main__":
    print("[Receiver] Waiting for temperature chunks...")
    received = bytearray()
    idle = 0
    IDLE_MAX = 64

    while True:
        try:
            if rx_ready():
                chunk = read_chunk4()
                received.extend(chunk)
                idle = 0

                if b'\n' in received:
                    line, _, rest = received.partition(b'\n')
                    try:
                        print(f"[Receiver] Received: {line.decode('utf-8').strip
   ()}")
                    except Exception:
                        print(f"[Receiver] Received (raw): {line!r}")
                    received = bytearray(rest)

                # safety: prevent unbounded growth
                if len(received) >= 64:
                    received = bytearray()

            else:
                idle += 1
                if idle > IDLE_MAX:
                    idle = 0
                    if received:
                        # flush partial buffer (optional)
                        try:
                            print(f"[Receiver] Partial: {received.decode('utf
-8','ignore')}")
                        except Exception:
                            print(f"[Receiver] Partial (raw): {received!r}")
                        received = bytearray()
                time.sleep(0.05)

        except KeyboardInterrupt:
            break
        except Exception as e:
            print(f"[Receiver] Error: {e}")
            time.sleep(1)
```

Chapter 6

System Testing and Debugging

To ensure that the I^{2}C interface is working correctly, we provide a simple Python script that writes and reads test data from the IoT Node (default I^{2}C address: 0x16). This allows students to quickly verify whether communication with the device is functional before proceeding with more advanced experiments.

I^{2}C Test Script

```
# i2c_test_tool.py
import time
import smbus

address = 0x16  # 52Pi IoT Node(A) default I2C address
bus = smbus.SMBus(1)

# Arbitrary test data to write
test_data = [10, 20, 30, 40]
registers = [0x11, 0x12, 0x13, 0x14]

print("=== I2C WRITE TEST ===")

# Write to registers
for i in range(4):
    bus.write_byte_data(address, registers[i], test_data[i])
    print(f"Wrote {test_data[i]} to register 0x{registers[i]:02X}")
    time.sleep(0.05)

print("\n=== I2C READ BACK ===")

# Read back and verify
passed = True
for i in range(4):
    read_val = bus.read_byte_data(address, registers[i])
    print(f"Read from register 0x{registers[i]:02X}: {read_val}")
    if read_val != test_data[i]:
        print(f"Mismatch at register 0x{registers[i]:02X}: "
              f"expected {test_data[i]}, got {read_val}")
        passed = False

if passed:
    print("\nResult: I2C communication PASS :)")
else:
    print("\nResult: I2C communication FAILED :(")
```

References

- Raspberry Pi Official Documentation: `https://www.raspberrypi.com/documentation/`
- Raspberry Pi OS Guide: `https://www.raspberrypi.org/software/`
- GPSD Project: `http://www.catb.org/gpsd/`
- Python Official Documentation: `https://docs.python.org/3/`
- LoRa Development Resources: `https://lora-developers.semtech.com/`
- IoT Node(A) Wiki: `https://wiki.52pi.com/index.php?title=EP-0105`

—

Acknowledgements

This project and teaching material were made possible through the support of:

- Southern Taiwan Science Park Administration (STSPA)
- National Sun Yat-sen University (NSYSU)
- National Pingtung University (NPTU)

Special thanks to the instructors, assistants, and students who contributed to the preparation and testing of the course materials.