CE406 IoT Lab

ESP32–ESP32 I^{2}C Communication (Master/Slave)

With a 30-minute Comparative Stress Test

Department of CSE

November 3, 2025

Objectives

- Explain I^{2}C basics (SDA/SCL, addressing, speed modes, wiring).
- Wire two ESP32 DevKit boards for I^{2}C on default pins and common ground.
- Implement a **slave** with `onReceive` & `onRequest` callbacks, and a **master** that writes/reads.
- Run a **comparative stress test** over a reduced parameter grid ($\geq$ 20 packets per setting).
- Compute **Throughput**, **Message rate**, **Error rate** and select a recommended configuration.

1. I^{2}C Overview (ESP32)

I^{2}C is a synchronous, multi-master, multi-slave two-wire bus: **SDA** (data) and **SCL** (clock). ESP32 exposes flexible I^{2}C pin mapping; common DevKit defaults are **SDA=GPIO21**, **SCL=GPIO22**. Typical modes include Standard (100 kbit/s) and Fast (400 kbit/s). Keep wires short and, for robustness, add 4.7 kΩ to 10 kΩ pull-ups from SDA/SCL to 3.3 V when wiring is longer or multiple devices share the bus.[1]

2. Hardware & Wiring

Parts: 2× ESP32 DevKit, 3× female–female jumpers (SDA, SCL, GND), optional 2× 4.7 kΩ to 10 kΩ pull-ups to 3.3 V.

Connections (short jumpers):

- Master **GPIO21 (SDA)** → Slave **GPIO21 (SDA)**
- Master **GPIO22 (SCL)** → Slave **GPIO22 (SCL)**
- **GND** ↔ **GND** (common ground)

[1] Concepts, default pins and callback patterns adapted from Random Nerd Tutorials, *ESP32 I2C Master and Slave (Arduino IDE)*.

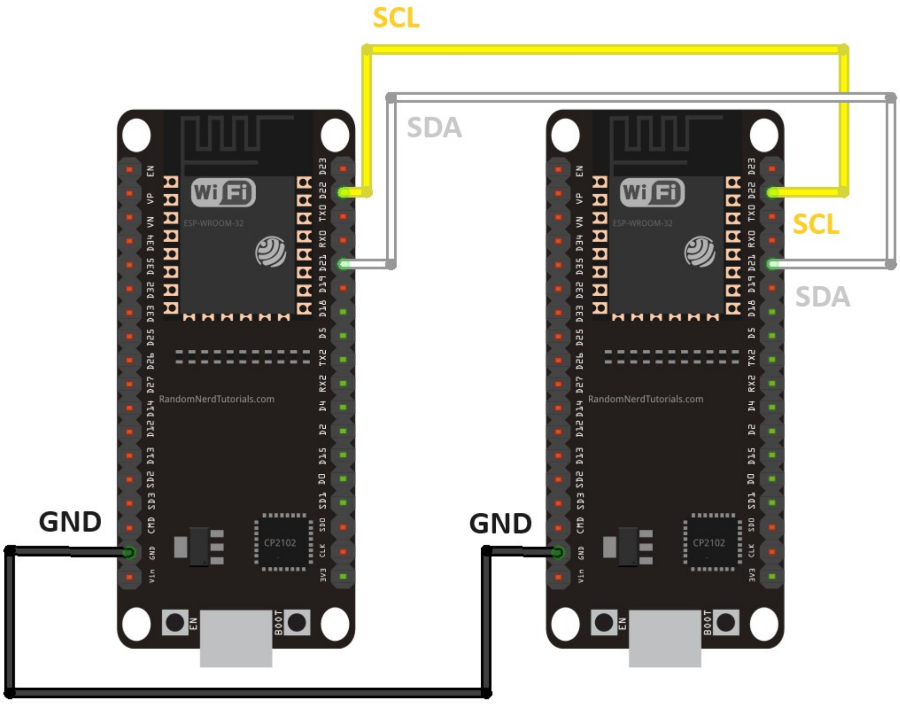

Figure 1: I2C wiring: SDA (GPIO21) to SDA, SCL (GPIO22) to SCL, and GND to GND.

If your board uses different defaults (e.g., C3/S3), remap with `Wire.setPins(sda,scl)` or `Wire.begin(addr, sda, scl, freq)` on the slave.

3. Software Setup

Arduino IDE with ESP32 core installed. Open two IDE windows to monitor `Serial` for both boards at 115 200 baud.

I^{2}C Slave (ESP32): what it does and why

Key ideas for students:

- `Wire.begin(address)` puts the ESP32 into *slave mode* at the given 7-bit address.
- `onReceive()` is a *callback* that runs automatically whenever the master writes to this slave.
- `onRequest()` is a *callback* that runs automatically when the master calls `requestFrom()`.
- We compute a simple XOR checksum of received bytes and store it as `last_checksum`. When the master asks for data, we send that one byte back as an *ACK token*. This lets the master check if the payload it sent arrived intact.
- `Serial.printf` lines show live status on the Serial Monitor, which is useful for debugging.

ESP32 I2C Slave (Arduino):

```
// ESP32 I2C Slave (Arduino) - minimal demo + 1-byte ACK
#include <Wire.h>

#define I2C_DEV_ADDR 0x55 // 7-bit address (match on master)
```

```

volatile unsigned long rx_count = 0;
volatile uint8_t last_checksum = 0; // for ACK in onRequest

void onReceive(int len) {
  // Read all bytes and compute a simple XOR checksum
  uint8_t sum = 0;
  Serial.printf("onReceive[%d]: ", len);
  while (Wire.available()) {
    uint8_t b = Wire.read();
    sum ^= b;
    Serial.write(b); // show raw data for learning/debug
  }
  Serial.println();
  last_checksum = sum; // cache for 1-byte ACK echo
  rx_count++;
}

void onRequest() {
  // Reply to master when it calls requestFrom()
  // Send the last checksum as a 1-byte ACK
  Wire.write(last_checksum);
  Serial.println("onRequest fired (ACK sent)");
}

void setup() {
  Serial.begin(115200);
  Wire.onReceive(onReceive);
  Wire.onRequest(onRequest);
  // Initialize I2C on default pins (DevKit: SDA=21, SCL=22)
  Wire.begin((uint8_t)I2C_DEV_ADDR);
  Serial.println("I2C Slave ready");
}

void loop() {
  // optional status every ~2s
  static uint32_t t0 = 0;
  if (millis() - t0 > 2000) {
    t0 = millis();
    Serial.printf("rx_count=%lu\n", rx_count);
  }
}
```

How to verify it works Open the slave Serial Monitor at 115 200 baud. You should see `I2C Slave ready`. When the master runs, you'll see `onReceive[. . . ]` lines with the bytes printed and periodic `onRequest` messages.

I^{2}C Master (ESP32) demo: what to look for

Key ideas for students:

- `Wire.begin()` puts this ESP32 into *master mode* (default pins).
- A master write is: `beginTransmission(addr)` → `write/printf(...)` → `endTransmission()`.
- A master read is: `requestFrom(addr, n)` then `Wire.read()/readBytes()`.

- The master prints any I²C error code from `endTransmission()` and shows the bytes it reads back from the slave.

ESP32 I2C Master (Arduino):

```
// ESP32 I2C Master (Arduino) - minimal demo
#include <Wire.h>
#define I2C_DEV_ADDR 0x55

uint32_t i = 0;

void setup() {
  Serial.begin(115200);
  Wire.begin(); // default SDA/SCL for this board
  Serial.println("I2C Master ready");
}

void loop() {
  delay(1000);

  // Send message to slave
  Wire.beginTransmission(I2C_DEV_ADDR);
  Wire.printf("Hello World! %lu", i++);
  uint8_t error = Wire.endTransmission(true); // true = send STOP
  Serial.printf("endTransmission err=%u\n", error);

  // Request up to 16 bytes from slave
  uint8_t n = Wire.requestFrom(I2C_DEV_ADDR, (uint8_t)16);
  Serial.printf("requestFrom bytes=%u\n", n);
  if (n) {
    uint8_t buf[32];
    if (n > sizeof(buf)) n = sizeof(buf);
    Wire.readBytes(buf, n);
    Serial.write(buf, n);
    Serial.println();
  }
}
```

How to verify it works Open the master Serial Monitor at 115 200 baud. You should see `I2C Master ready`, periodic `endTransmission err=0`, and a line showing how many bytes were read. The characters printed are whatever the slave returned (in our slave we return dynamic text or the single-byte ACK in other variants).

Comparative Stress Test

Goal: Run multiple settings automatically and measure performance.

Parameter grid (send ≥ 20 packets per setting)

- I²C clock (Hz): {100000, 400000}
- Message size (bytes): {10, 50}
- Interval (ms between packets): {0, 10}
- Total combinations: $2 \times 2 \times 2 = 8 \Rightarrow 8 \times 20 = 160$ packets overall

Metrics (for test window duration T s)

- **Throughput (B/s):** $\dfrac{\text{total payload bytes received}}{T}$
- **Message rate (msg/s):** $\dfrac{\text{messages received}}{T}$
- **Error rate (%):** $100 \times \dfrac{\text{sent} - \text{valid received}}{\text{sent}}$

Master *Test Program* (automates grid & metrics)

What this code adds for students:

- A small `struct Cfg` holds one row of the parameter grid: bus speed, payload size, and gap between packets.
- We prefill a payload with ASCII letters (`'A'..'Z'`) to make it deterministic and readable.
- Before each packet, we compute an XOR **checksum** over the payload; the slave echoes this value as a 1-byte ACK. If the master receives the same byte it sent as checksum, we count it as a *valid* packet.
- After sending `NUM_PACKETS` for one setting, we compute throughput, message rate, and error rate and print a one-line summary that you can paste into a spreadsheet.

DHT11 Example Code:

*(Same label; this block is the ESP32 I^2C **Master Test Program**.)*

```
// ESP32 I2C Master -- test program for comparative stress test
#include <Wire.h>
#define I2C_DEV_ADDR 0x55
#define NUM_PACKETS 20 // >= 20 per setting

struct Cfg { uint32_t hz; uint8_t size; uint16_t gap_ms; };
Cfg grid[] = {
  {100000,10,0}, {100000,10,10},
  {100000,50,0}, {100000,50,10},
  {400000,10,0}, {400000,10,10},
  {400000,50,0}, {400000,50,10},
};

uint8_t payload[128];

static inline uint8_t checksum(const uint8_t* p, uint8_t n){
  uint8_t s=0; for(uint8_t i=0;i<n;i++) s^=p[i]; return s;
}

void run_one(const Cfg& c){
  Wire.setClock(c.hz);
  for (uint8_t i=0;iG<c.size;i++) payload[i] = 'A' + (i % 26);

  uint32_t sent=0, valid=0;
  uint32_t t0 = millis();

  for (uint16_t k=0;k<NUM_PACKETS;k++){
    uint8_t chk = checksum(payload, c.size);

    Wire.beginTransmission(I2C_DEV_ADDR);
```

```
    Wire.write((uint8_t)(k>>8)); Wire.write((uint8_t)k); // seq
    Wire.write(c.size);
    Wire.write(payload, c.size);
    Wire.write(chk);
    uint8_t err = Wire.endTransmission(true);
    if (err==0) sent++;

    // 1-byte ACK from slave (should echo checksum)
    if (Wire.requestFrom(I2C_DEV_ADDR,(uint8_t)1)==1) {
      uint8_t ack = Wire.read();
      if (ack==chk && err==0) valid++;
    }

    if (c.gap_ms) delay(c.gap_ms); else delayMicroseconds(500);
  }

  float T = (millis()-t0)/1000.0f;
  float thr = (valid * c.size) / T; // B/s
  float rate = (float)valid / T; // msg/s
  float errp = sent? (100.0f*(sent-valid)/sent) : 100.0f;

  Serial.printf("F=%luHz size=%uB gap=%ums | sent=%lu valid=%lu T=%.2fs | thr=%.1fB/s rate
      =%.2f/s err=%.1f%%\n",
    (unsigned long)c.hz, c.size, c.gap_ms,
    (unsigned long)sent, (unsigned long)valid, T, thr, rate, errp);
}

void setup(){
  Serial.begin(115200);
  Wire.begin();
  Serial.println("I2C Master (test program) ready");
  for (auto &c : grid) {
    Serial.printf("=== %lu Hz, %u B, %u ms ===\n",
      (unsigned long)c.hz, c.size, c.gap_ms);
    run_one(c);
    delay(300);
  }
  Serial.println("All tests done.");
}

void loop(){}
```

7. Data Sheets (fill these from Serial output)

Table 1: ESP32 I^2C results

Freq (Hz)	Size (B)	Gap (ms)	Throughput (B/s)	Msg/s	Error (%)
100000	10	0			
100000	10	10			
100000	50	0			
100000	50	10			
400000	10	0			
400000	10	10			
400000	50	0			
400000	50	10			

Table 2: Recommended configuration (justify briefly)

Freq (Hz)	Size (B)	Gap (ms)	Rationale

9. What to Submit

1. Final **Master & Slave** sketches.
2. Serial logs for all 8 settings.
3. Completed tables and a 1–2 page discussion selecting and justifying a "sweet spot" configuration.

Reference

Random Nerd Tutorials, *ESP32 I2C Master and Slave (Arduino IDE)* — wiring, default pins, and callback patterns adapted for this lab handout.[2]

[2] https://randomnerdtutorials.com/esp32-i2c-master-slave-arduino/