📘 I²C Bus Control & Arbitration — Summary

1. Bus basics

- **Two lines:**
  - **SCL** (clock)
  - **SDA** (data)
- Both are **open-drain** → devices can only pull the line **LOW**; resistors pull it **HIGH** when released.
- This prevents short circuits and allows multiple devices to share the bus.

---

2. START & STOP conditions

- **START:** SDA goes HIGH → LOW *while SCL is HIGH* → "bus is busy."
- **STOP:** SDA goes LOW → HIGH *while SCL is HIGH* → "bus is free."
- All devices constantly monitor SDA+SCL to detect these events.

---

3. Data transfer rules

- Data bits are valid only while **SCL is HIGH**.
- Devices can change SDA only while **SCL is LOW**.
- After each byte (8 bits), the receiver must send an **ACK/NACK** bit by pulling SDA LOW or leaving it HIGH during the 9th clock pulse.

---

4. Arbitration (multiple masters)

- If two masters start at once, both transmit until a difference occurs.
- **Rule:** LOW dominates (wired-AND).
- Example:
  - Master A sends 0, Master B sends 1 → bus reads 0.
  - Master B sees a mismatch and stops transmitting.

- Only one master continues; no data corruption occurs.

---

5. Slave behavior

- Slaves **never generate START/STOP**.
- They only respond when their **address** is called by the master.
- They pull SDA low only when sending ACK/NACK or data (but always synchronized to master's clock).

---

6. Built-in logic

- The ATmega328P and other I²C chips have **hardware state machines (transistor logic)** that:
    - Detect START/STOP.
    - Shift data bits in/out.
    - Handle ACK/NACK timing.
    - Perform arbitration checks.
- As a programmer, you don't manually wiggle SDA/SCL.
- Instead, you control the **TWI registers** (TWBR, TWCR, TWSR, TWDR) and the hardware enforces protocol rules.

✅ **In one sentence:**
I²C bus control is enforced by built-in hardware logic: masters claim the bus by issuing a START, data is valid only when SCL is HIGH, LOW always wins during arbitration, and you as a bare-metal programmer just configure and step through the TWI registers while the hardware ensures correct protocol behavior.

📘 I²C Addressing — Notes

1. 7-bit addressing

- I²C uses a **7-bit address** → allows up to 127 possible addresses.

- In reality, only ~112 are usable (some reserved).
- The master sends:
- [A6 A5 A4 A3 A2 A1 A0 | R/W]
    - A6…A0 = 7-bit slave address
    - R/W = 1 bit → **Read (1)** or **Write (0)**

Example:

- Device address = 0x27 (binary 0100111)
- Write command byte = 01001110 (0x4E)
- Read command byte = 01001111 (0x4F)

---

2. Slave recognition (the "switches" analogy)

- Inside each slave, the 7-bit address is stored in **hardware comparator logic**.
- Think of it as **7 digital switches (XOR checks)** that must all match the master's bits.
- If all switches match → the slave:
    - Pulls SDA low during 9th clock → **ACK**
    - Gets ready to receive (write) or send (read) data depending on R/W bit
- If the address does not match → the slave ignores the transaction and keeps its SDA driver off (**high-impedance**).

---

3. Why this works

- Only the addressed slave enables its driver.
- All other slaves stay disconnected (high-Z), so there's no conflict.
- If two devices accidentally share the same address → both will ACK, causing bus errors.

---

4. Reserved addresses

- Some addresses are not available to normal devices:

- 0x00 = General Call (broadcast)
- 0x01–0x07 and some others = reserved

- This leaves ~112 usable unique addresses.

✅ **Key takeaways**

- The 7-bit address is not "just a register" — it's **hardware logic inside the slave**.
- The master always sends **7 bits + R/W bit** as the first byte.
- Only the slave with the matching internal "switch pattern" responds.
- This keeps the bus organized and prevents multiple slaves from talking at once.

🔄 **How I²C communication flows**

1. **Clock pulses (SCL)**
   - The master generates SCL pulses at a set speed (e.g. 100 kHz, 400 kHz).
   - These pulses give the "beat" — all devices stay in sync.
2. **Data line (SDA)**
   - During each clock cycle, the master sets SDA to 0 (pull low) or 1 (release, pulled high).
   - Rule: SDA must be **stable when SCL is high** → that's when receivers sample the bit.
3. **Address phase**
   - First byte = [7-bit address][R/W bit].
   - The master shifts it out, one bit per SCL pulse.
4. **ACK phase** (the 9th clock)
   - After 8 bits, the master **releases SDA** (lets it float high).
   - The addressed slave checks: "Do those 7 bits match my internal switches?"
     - If yes → it **pulls SDA low** during the 9th pulse = **ACK**.

        - If no → it leaves SDA high = **NACK**.

5. **Master checks**
    - Master looks at SDA during the 9th pulse.
    - If LOW → "slave exists and is listening."
    - If HIGH → "nobody answered."

✅ **Summary in plain words**

- The master drives the **data (SDA)**, timed by its **own clock pulses (SCL)**.
- After sending the address, the master "lets go" of SDA for one clock.
- If the right slave is present, it **pulls SDA low** → an ACK handshake saying *"Yes, I exist, keep talking."*
- If nobody answers, SDA stays high → NACK.

So yes — **SCL is the heartbeat**, **SDA is the spoken word**, and **ACK is the slave raising its hand saying "I heard you."**

📘 **1602A LCD with I²C Backpack (PCF8574) — Notes**

1. Raw 1602A LCD (16 pins)

1 VSS → GND

2 VDD → +5V

3 V0 → Contrast (potentiometer)

4 RS → Register Select (0=command, 1=data)

5 RW → Read/Write (0=write, 1=read)

6 E → Enable (latches command/data)

7–14 D0–D7 → Data lines (can use 4 or 8 bits)

15 A → Backlight Anode (+)

16 K → Backlight Cathode (–)

- Normally you'd need 10+ Arduino pins to control this.

---

2. With I²C backpack (PCF8574)

- The backpack physically connects to **all 16 pins** of the LCD.
- But the **PCF8574 only has 8 outputs** → it can't drive all 16.
- The backpack designer hardwires the "static" pins and only routes the **essential control/data lines** through the I²C expander.

---

3. Which pins are hardwired

- **VSS (GND)** → directly to ground.
- **VDD (+5V)** → directly to power.
- **V0 (contrast)** → to potentiometer on the backpack.
- **RW** → tied LOW (always write mode).
- **Backlight A/K** → tied to +5V/GND, sometimes via jumper/resistor. (Some backpacks allow software backlight control through PCF8574.)

👉 These pins never need to be changed in software.

---

4. Which pins are under I²C control

- **RS** (selects command/data).
- **E** (Enable strobe to latch data).
- **D4–D7** (4-bit data bus).
- (Optional) Backlight control if wired to PCF8574.

That's 6–7 signals → easily handled by the 8 available PCF8574 outputs.

---

5. How it works

1. Master sends **1 byte over I²C**.
2. PCF8574 updates its 8 output pins according to that byte.
3. Those pins drive the LCD's control + data lines.
4. The LCD controller (HD44780) interprets the signals as either a **command** or **data** depending on RS and the bits on D4–D7.

✅ **Key takeaways**

- The backpack touches all 16 pins, but **only 6–7 are "dynamic."**
- The rest are permanently tied to power, ground, or the contrast pot.
- You **cannot control all 16 pins via I²C** → only the ones routed through PCF8574.
- Communication always happens in **4-bit mode** (D4–D7), so you often send two nibbles for one full byte command/data.

👉 This is why you send a single I²C byte, but it ends up as a whole "LCD instruction" after the PCF8574 + HD44780 logic chain.

🔑 **General rule for I²C slaves**

- I²C itself is just a **pipe**:
    - The bus only guarantees bytes are delivered (START → address → data → STOP).
    - It does **not** describe what those bytes *mean*.
- Every **slave device needs its own internal logic / controller** to:
    - Recognize its address.
    - Interpret incoming bytes according to its design.
    - Decide whether they mean "write to register," "send me sensor data," "turn on an output," etc.

📖 Examples

1. **PCF8574 I/O expander (your LCD backpack)**
    - Bytes received = directly mapped to 8 output pins.
    - No “registers,” just output latches.
2. **Temperature sensor (e.g. TMP102)**
    - First byte after address = register pointer (temperature, config, etc.).
    - Following bytes = data (read or write).
    - The internal state machine knows: “If I see 0x00, that means the temperature register.”
3. **EEPROM (e.g. 24LC256)**
    - First two bytes = memory address.
    - Next bytes = data to write.
    - Internal logic knows those bytes are not literal output pins but memory writes.

✅ Key point

- The **meaning of the data** is **not defined by I^2C**.
- It’s defined by the **slave’s chip design and datasheet**.
- That’s why:
    - You always read the slave’s datasheet.
    - Master code has to “speak the protocol” that specific chip expects.

🚦 Summary for notes

- I^2C provides the **addressing + data transport mechanism**.
- Each slave has its own **internal logic** (a small controller, state machine, or just latches).

- That logic decides what to do with the data — whether it's a command, register index, or raw value.
- As a programmer, you must follow the **datasheet protocol** for each slave device.