ARM Microcontroller Internship Programme

Trainer: *Kiran Jaybhave*

Sunbeam – Industrial Training Programme

Sunbeam Pune (Hinjawadi)

I2C Protocol

- **I²C** stands for **Inter-Integrated Circuit .**
- The **I²C bus** was **originally developed by Philips**
- Over time, it became a **widely accepted industry standard**
- Today, it is supported by **almost all semiconductor manufacturers**
- It is a **serial communication bus** used to connect **multiple digital devices** on the same board
- I²C is widely used in **embedded systems** because it requires **only two wires**
- Used to communicate with common peripherals like:
  - **Sensors** (Temperature, Light, Accelerometer)
  - **RTC** (Real Time Clock)
  - **EEPROM**
  - **LCD displays**
- Found in **microcontrollers**, **SoCs**, and **embedded boards** (Arduino, STM32, ESP32)

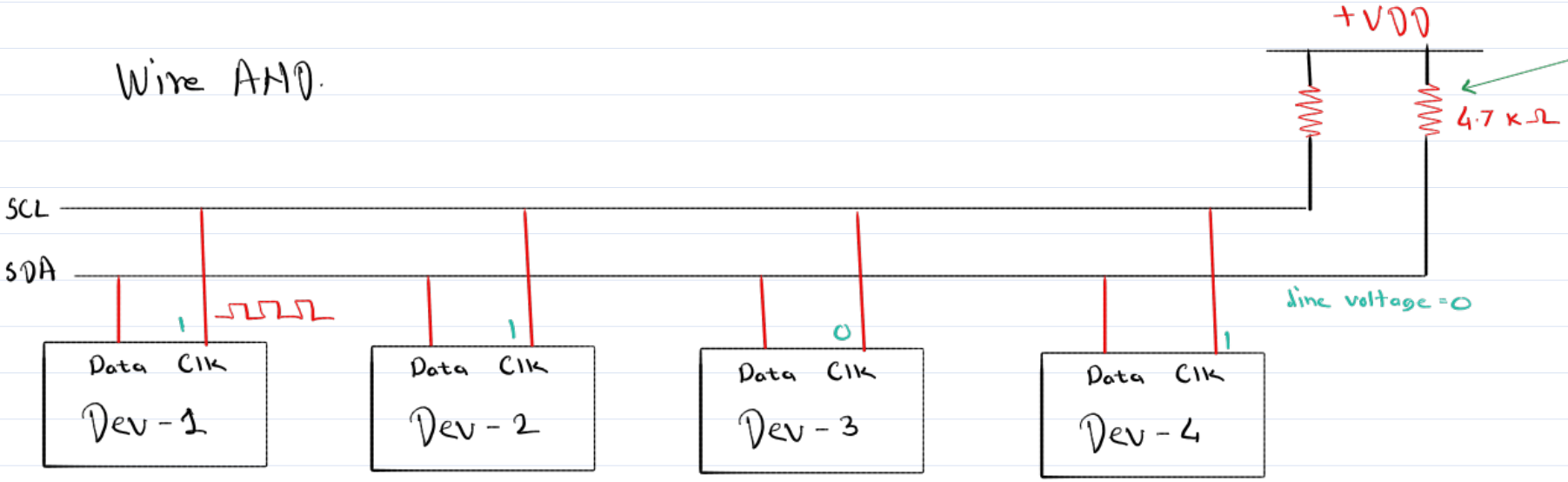
I²C – Physical Characteristics

- **Type**
  - I²C is a **bus protocol.**
  - One **Master device** controls the communication.
  - One or more **Slave devices** respond to the master , This is called **Master–Slave architecture**
  - **Example :**STM32 (Master) ↔ EEPROM / Sensor / LCD (Slave)
- **Communication Mode**
  - I²C is a **Half-Duplex protocol**
  - Data flows in **one direction at a time**
  - Either **Master → Slave** or **Slave → Master**
  - Data cannot be sent and received **simultaneously**
- **Wires / Connectivity**
  - I²C uses **only two wires**
  - These two wires are shared by **all devices on the bus**
  - This makes hardware:
    - Simple
    - Low cost
    - Easy to expand

I²C – Physical Characteristics

- **SDA – Serial Data Line**
  - SDA is used to **transfer data**
  - Data can move in **both directions**
  - Used for:
    - Sending address
    - Sending data
    - Receiving data
    - Acknowledgement (ACK/NACK)
- **SCL – Serial Clock Line**
  - SCL provides the **clock signal**
  - Clock is **generated by the Master**
  - All data transfer is **synchronized using this clock**
  - Without SCL, data timing cannot be controlled

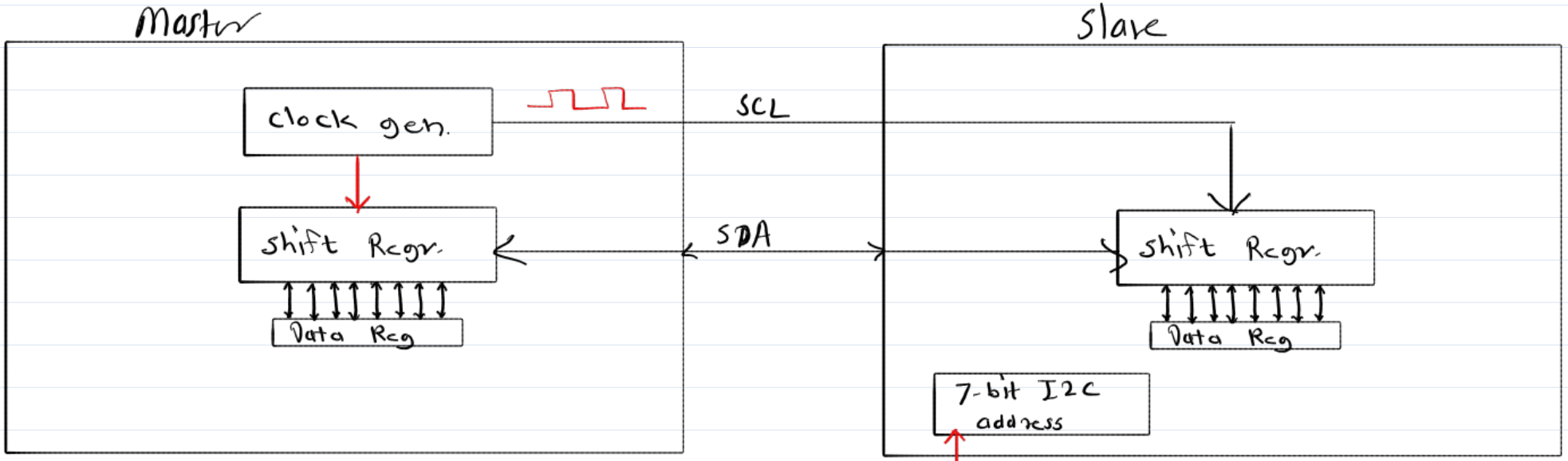
Wire AND.



Pull up Resistor
- Use to Hold line voltage High.
- Resister value use for High speed comm^n
- 4.7 KΩ gives 100 KHz data transfer., for want High speed Reduce the Resister value.

* We can connect upto 128 device.

Wire AND
- Similar like AND Gate
- If any line make Data/Clk pull line (zero) Then effective line voltage is zero (vtg grounded)



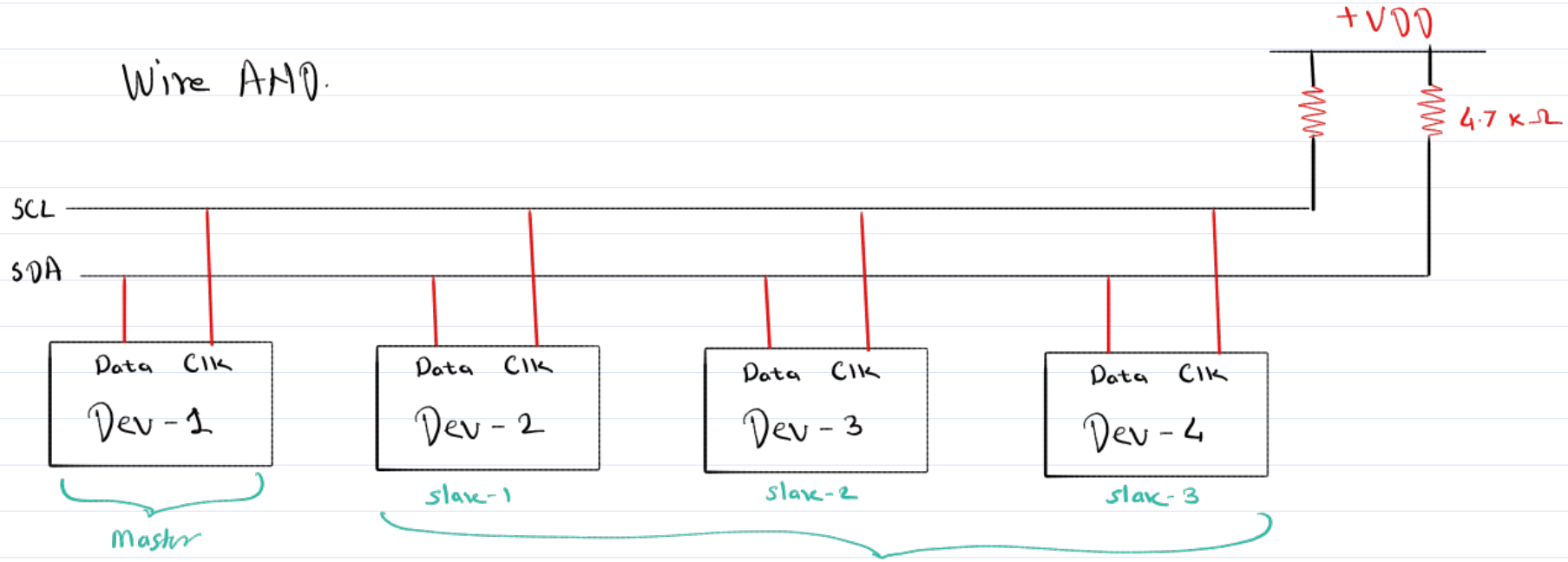
- Every slave has own unique Address (given in data sheet)
- Address given by I²C global commitee.

- 2^7 = 128 unique Address

few address reserved for special purpose

like 0x00 → genral Call . (gik call all device - for check, or send data all)

Wire AND.



- Generate clock
- Start / Stop comm"
- Select the slave

e.g. STM32, AVR...

- follow order of master.
- Ack when it address is selected
- data transfer (Wrik / Read)

e.g. EEPROM, LCD

I²C – Physical Characteristics

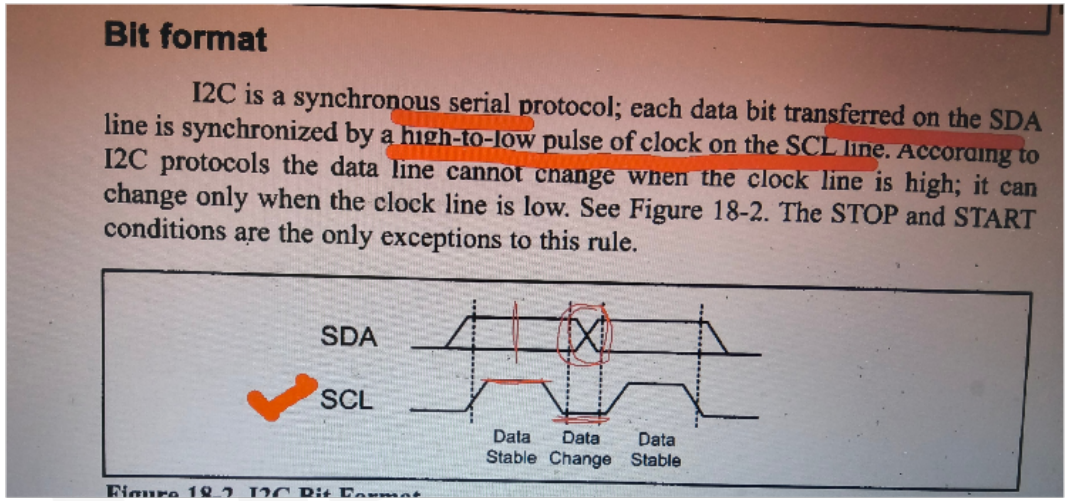
- **Voltage Levels**
  - I²C uses **TTL logic levels**
  - Logic levels are typically:
    - **0 V** → Logic LOW
    - **3.3 V or 5 V** → Logic HIGH
  - Depends on the microcontroller and devices used
- **I²C Operating Frequency**
  - I²C supports different speed modes:
  - **100 kHz → Standard Mode**
  - **400 kHz → Fast Mode**
  - **1 MHz → Fast-Plus Mode**
  - Higher frequency = Faster data transfer

I²C – Logical Characteristics

- **I²C Device Modes**
  - An I²C device can work in **four logical modes** depending on its role:
  1. **Master Transmitter**
     - Master sends data to a slave
     - Example: STM32 writing data to EEPROM
  2. **Master Receiver**
     - Master receives data from a slave
     - Example: STM32 reading temperature from a sensor
  3. **Slave Transmitter**
     - Slave sends data when master requests it
     - Example: Sensor sending data to STM32
  4. **Slave Receiver**
     - Slave receives data sent by master
     - Example: EEPROM receiving data from STM32
  - Same device can act as **transmitter or receiver** depending on operation

I²C – Logical Characteristics

- **Data Bit Transfer**
  - Data on I²C is transferred **bit by bit**
  - One data byte = **8 bits**
  - After every byte, an **ACK (Acknowledge)** is sent
  - Clock (SCL) controls when data is sampled
- **Data Frame Format**
  - Each data transfer follows this format:
  - **8 Data Bits + 1 ACK Bit**
    - **[D7 D6 D5 D4 D3 D2 D1 D0] + ACK**
  - **ACK (Acknowledge) Concept**
    - **ACK = 0 (LOW)** → Data received successfully
    - **NACK = 1 (HIGH)** → Data not received / not accepted
    - **Receiver always sends ACK**
  - **Data Frame Communication**
    - **Transmitter** sends 8 data bits
    - **Receiver** sends ACK after receiving data
  - **Transmitter ---> 8 Data Bits ---> Receiver**
  - **Receiver ---> ACK (0) ---> Transmitter**



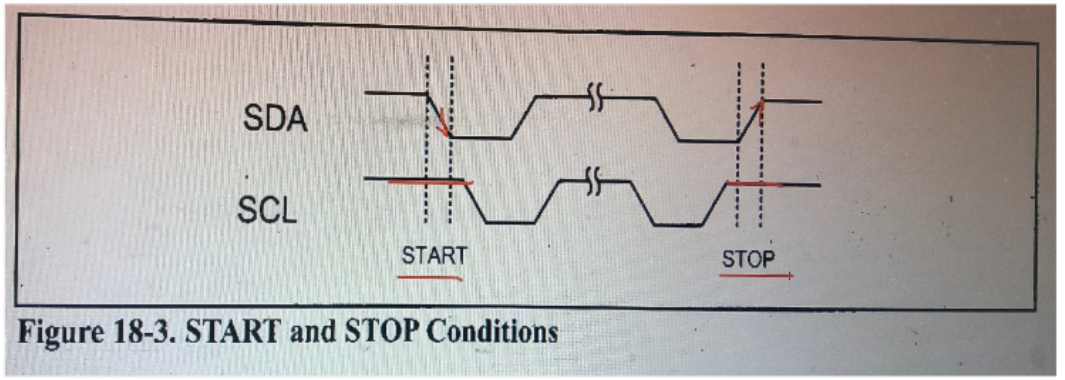


Figure 18-3. START and STOP Conditions

I²C – Logical Characteristics

- **Address Frame Format**
  - Before data transfer, **address frame** is sent:
    - [A6 A5 A4 A3 A2 A1 A0] [R/W] + ACK
      - **R/W = 0** → Write operation
      - **R/W = 1** → Read operation
  - **Address Frame Communication**
    - **Master** sends slave address + R/W bit
    - **Slave** responds with ACK if address matches
    - Master ---> 7-bit Address + R/W ---> Slave
    - Slave ---> ACK (0) ---> Master
    - If slave does **not respond**, ACK is not received

I²C – Single Byte Data Write Operation

➢ **START (M) + Slave Address + W (MT) + ACK (SR) + Internal Address (MT) + ACK (SR) + Data Byte (MT) + ACK (SR) + STOP (M)**

- **Steps**

1. **START Condition (M)**
   - Generated by the **Master**
   - Indicates the **beginning of I²C communication**
   - All slaves get ready to listen
2. **Slave Address + Write Bit (MT)**
   - Master sends **7-bit slave address**
   - **Write bit (W = 0)** indicates write operation
   - Only the addressed slave responds
3. **ACK from Slave (SR)**
   - Slave pulls SDA **LOW**
   - Confirms: *"Yes, this address is mine"*
4. **Internal Address (MT)**
   - Master sends **internal register/memory address**
   - Used in devices like:
     - EEPROM , Sensors ,RTC
   - Tells the slave **where to write the data**
5. **ACK from Slave (SR)**
   - Slave confirms it received the internal address
6. **Data Byte (MT)**
   - Master sends **one byte of actual data**
   - This data is written into the specified internal location.
7. **ACK from Slave (SR)**
   - Slave confirms data is received successfully
8. **STOP Condition (M)**
   - Generated by the **Master**
   - Indicates **end of communication**
   - Bus becomes free for next operation

I²C – Multi-Byte Data Write Operation

- **START (M) + Slave Address + W (MT) + ACK (SR) + Internal Address (MT) + ACK (SR) + Data Byte 1 (MT) + ACK (SR) + Data Byte 2 (MT) + ACK (SR) + . . . + STOP (M)**
- **Step Explanation**

1. **START Condition (M)**
   - Generated by the **Master**
   - Signals the **beginning of communication**
2. **Slave Address + Write Bit (MT)**
   - Master sends **7-bit slave address**
   - **W = 0** → Write operation
   - Only the matched slave responds
3. **ACK from Slave (SR)**
   - Slave confirms address reception
   - Communication continues
4. **Internal Address (MT)**
   - Master sends **starting internal register/memory address**
   - Indicates **where the first data byte will be written**
5. **ACK from Slave (SR)**
   - Slave accepts the internal address
6. **Multiple Data Bytes (MT)**
   - Master sends **DataByte1, DataByte2, …**
   - After **each data byte**, slave sends **ACK**
   - Data is written **sequentially**
   - Internal address **auto-increments** (device dependent)
7. **STOP Condition (M)**
   - Master ends the transmission
   - Bus becomes **idle**

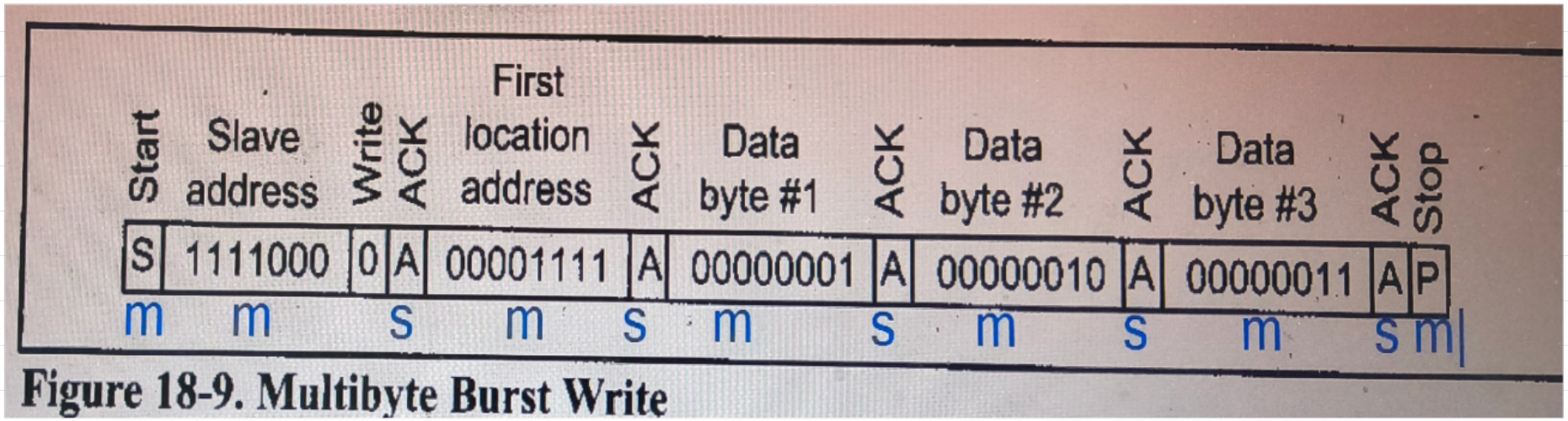


Figure 18-9. Multibyte Burst Write

I²C – Single Byte Read Operation

- **Write internal address first, then read data)**
- **START (M) + Slave Address + W (MT) + ACK (SR) + Internal Address (MT) + ACK (SR) + REPEATED START (M) + Slave Address + R (MR) + ACK (SR) + Data Byte (ST) + NACK (MR) + STOP (M)**
- **Steps:**

1. **START Condition (M)**
   - Generated by the **Master**
   - Begins I²C communication
2. **Slave Address + Write Bit (MT)**
   - Master sends **7-bit slave address**
   - **W = 0** → Write mode
   - Purpose: to send **internal address**
3. **ACK from Slave (SR)**
   - Slave confirms it is selected
4. **Internal Address (MT)**
   - Master sends **register / memory address**
   - Tells slave **which location to read from**
5. **ACK from Slave (SR)**
   - Slave accepts internal address
6. **REPEATED START (M)**
   - Master does **NOT release the bus**
   - Changes direction from **write → read**
   - Prevents other masters from taking control
7. **Slave Address + Read Bit (MR)**
   - Same slave address is sent again
   - **R = 1** → Read mode
8. **ACK from Slave (SR)**
   - Slave is ready to send data
9. **Data Byte from Slave (ST)**
   - Slave sends **one byte of data**
   - Master receives it
10. **NACK from Master (MR)**
    - Master sends **NACK**
    - Indicates: *"I want only one byte"*
11. **STOP Condition (M)**
    - Master ends communication
    - Bus becomes free

* Address Packet alway send After start.

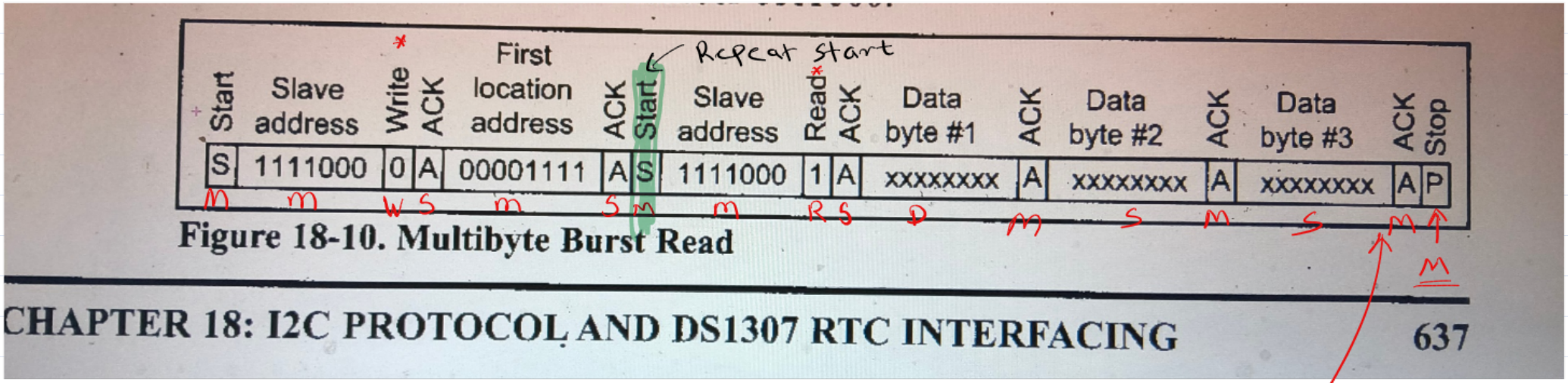


Figure 18-10. Multibyte Burst Read

After Read last data, Master's Not Send Ack signal, it send stop

I²C – Multi-Byte Data Read Operation

- **Write internal address once, then read multiple bytes.**
- **START (M) + Slave Address + W (MT) + ACK (SR) + Internal Address (MT) + ACK (SR) + REPEATED START (M) + Slave Address + R (MR) + ACK (SR) + Data Byte 1 (ST) + ACK (MR) + Data Byte 2 (ST) + ACK (MR) + ... + Last Data Byte (ST) + NACK (MR) + STOP (M)**
- **Steps**

1. **START Condition (M)**
   - Generated by the **Master**
   - Begins I²C communication
2. **Slave Address + Write Bit (MT)**
   - Master selects the slave
   - **W = 0** → Write mode (to send internal address)
3. **ACK from Slave (SR)**
   - Slave confirms address match
4. **Internal Address (MT)**
   - Master sends starting register / memory address
   - Slave prepares data from this location
5. **ACK from Slave (SR)**
   - Internal address accepted
6. **REPEATED START (M)** ✡
   - Master keeps control of the bus
   - Changes direction from **write → read**
7. **Slave Address + Read Bit (MR)**
   - Same slave address
   - **R = 1** → Read mode
8. **ACK from Slave (SR)**
   - Slave ready to transmit data
9. **Multiple Data Bytes from Slave (ST)**
   - Slave sends **DataByte1, DataByte2, …**
   - Master sends **ACK after each byte**
   - Internal address auto-increments
10. **NACK from Master (MR) – Last Byte**
    - Master sends **NACK** after final byte
    - Indicates: *"No more data required"*
11. **STOP Condition (M)**
    - Master ends communication
    - Bus becomes idle

* Address Packet alway send After Start

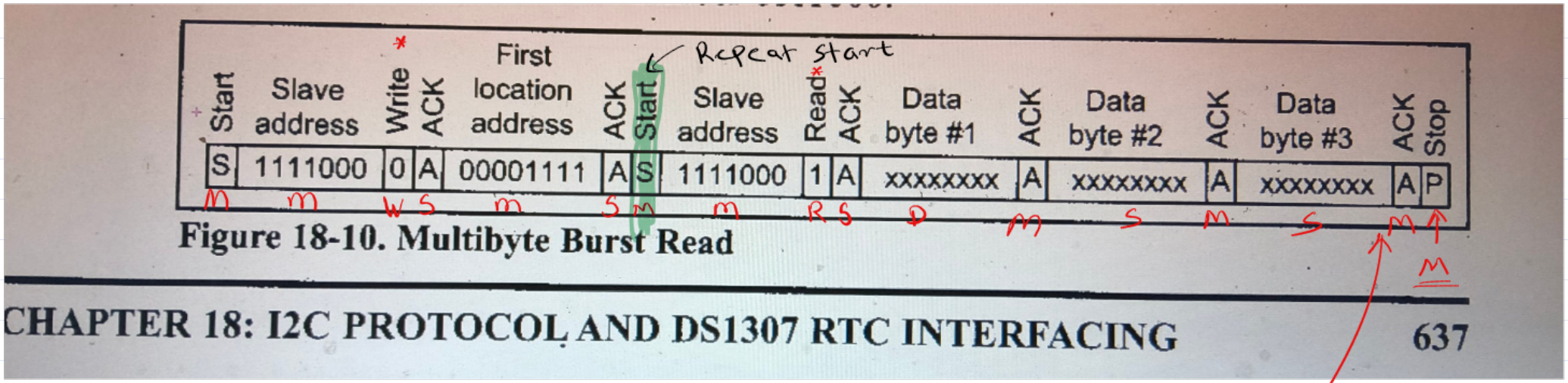


Figure 18-10. Multibyte Burst Read

After Read last data, Master's Not Send Ack signal, it send stop

I²C – Clock Stretching

- **Clock stretching** is a feature of I²C where the **slave can pause communication**
- If the slave is **busy processing data**, it can **hold the clock line (SCL) LOW**
- As long as SCL is LOW, the **clock is effectively disabled.**
- Clock stretching is **controlled by the slave**
- Master must **wait** until SCL goes HIGH again
- This mechanism prevents **data loss** ,Not all masters support clock stretching (implementation dependent)

❖Why Clock Stretching is Needed

- Some slave devices are **slower** than the master
- Slave may need time to:
  - Process received data
  - Read internal memory
  - Prepare data to send
- Clock stretching allows **safe synchronization** between master and slave

❖How Clock Stretching Works

- Master generates clock pulses on **SCL**
- Slave needs more time → it **pulls SCL LOW**
- Because SCL is LOW:
  - Clock stops
  - **No data transfer occurs on SDA**
- When slave is ready:
  - It **releases SCL**
  - Master continues clock generation

* Clock stretching.

One of the features of the I²C protocol is clock stretching. It is a kind of flow control. If an addressed slave device is not ready to process more data it will stretch the clock by holding the clock line (SCL) low after receiving (or sending) a bit of data. Thus the master will not be able to raise the clock line (because devices are wire-ANDed) and will wait until the slave releases the SCL line to show it is ready to transfer the next bit. See Figure 18-8.

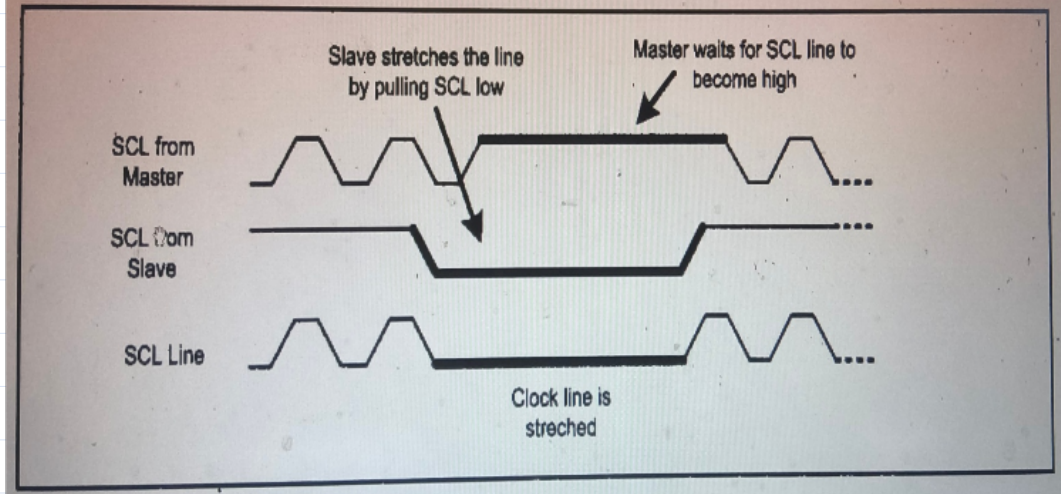
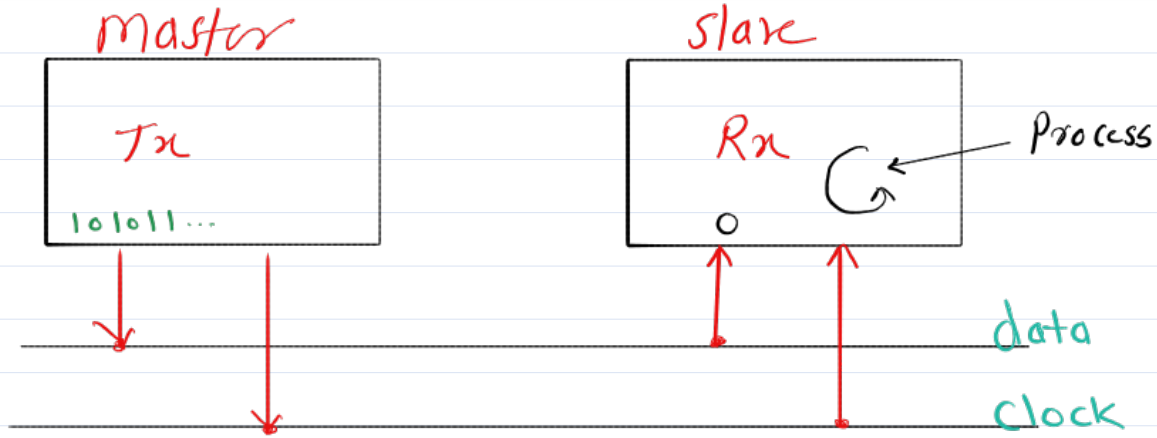


Figure 18-8. Clock Stretching



Master (Tx) : 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0

Slave (Rx) : 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

Clock 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0

↓

no clock

↓

shift Register

↓

No data Transfer

I²C – Arbitration

- **Arbitration** occurs when **more than one device tries to become Master**
- This can happen in a **multi-master I²C bus**
- Arbitration ensures that **only one master controls the bus at a time**
- It is handled **automatically by hardware**
- **When Does Arbitration Occur?**
  - When **two devices generate START condition at the same time**
  - Both devices believe they are the master
  - They start transmitting address/data together
- **How Arbitration Works**
  - I²C uses **open-drain lines** (wired-AND logic)
  - **LOW (0) always dominates HIGH (1)** on the bus
  - **Rule:**
  - The device that sends **LOW (0) first wins arbitration**

* Bus Arbitration

Comment View Form Protect Share Help Tell me...

stm32f407_user_manu... | ipc1768_user_manual.pdf | AVR Microcontroller and E...

Arbitration

Happens when 2 dev try to master at same time

I2C protocol supports a multimaster bus system. This doesn't mean that more than one master can use the bus at the same time. Rather, each master waits for the current transmission to finish and then starts to use the bus. But it is possible that two or more masters initiate a transmission at about the same time. In this case the arbitration happens.

Each transmitter has to check the level of the bus and compare it with the level it expects; if it doesn't match, that transmitter has lost the arbitration, and will switch to slave mode. In the case of arbitration, the winning master will continue its job. Notice that neither the bus is corrupted nor the data is lost. See Example 18-5.

Example 18-5

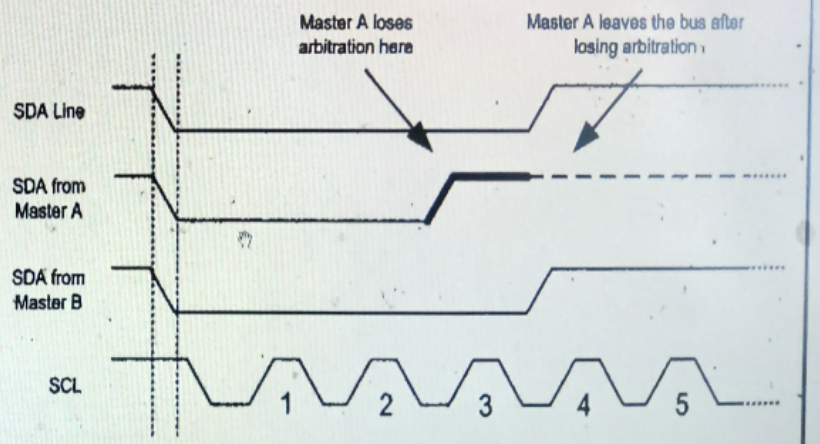
Two masters, A and B, start at about the same time. What happens if master A wants write to slave 0010 000 and master B wants to write to slave 0001 111?

Example 18-5

Two masters, A and B, start at about the same time. What happens if master A wants to write to slave 0010 000 and master B wants to write to slave 0001 111?

Solution:

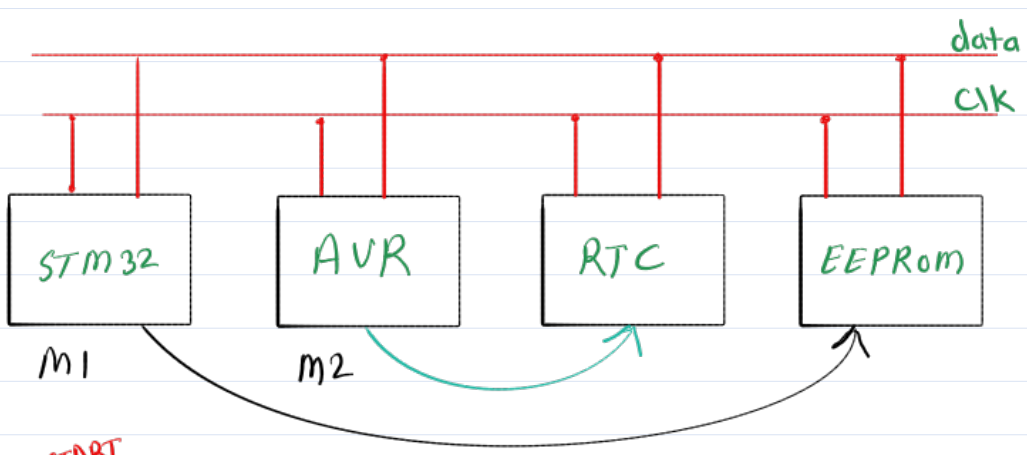
Master A will lose the arbitration in the third clock because the SDA line is different from the output of master A at the third clock. Master A switches to slave mode and leaves the bus after losing the arbitration.



Bus Arbritation

R|$\overline{W}$ =>
1 0

* when 2 devisess try to Transmitt data to ~~two~~ diff device - Then who win Aribritaction He become master



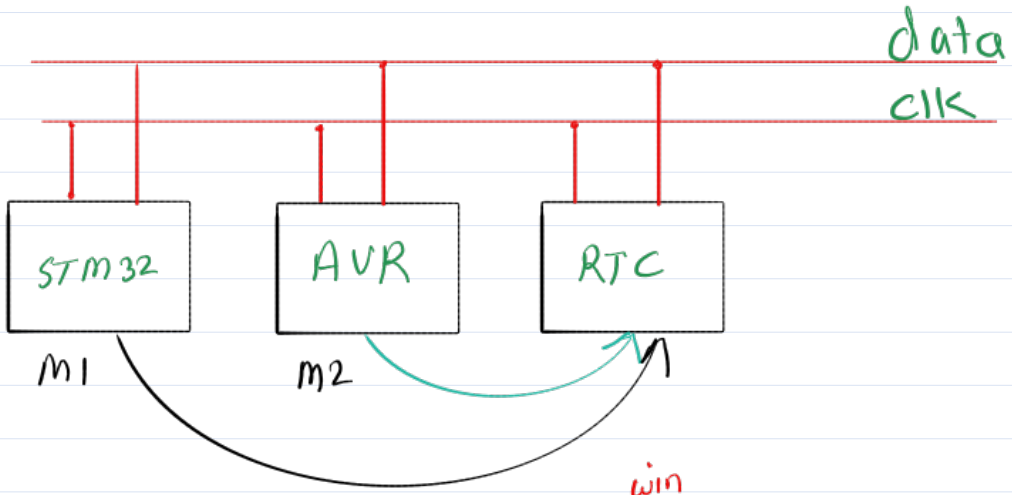
START

M1 S 1 0 1 0 0 0 0 1 0 ..... } Two master write
M2 S 1 0 1 0 0 1 → lost Arbritation } Data

x
0 → Stop further Comm^n

master 2 write 1st
1 so it lost bus Arbritation
Then master 1 continue

* when 2 devisess try to Transmitt data to same device - Then who win Aribritaction He become master



win

M1 : S 1 0 1 0 1 0 0 0 0 0 1

M2 : S 1 0 1 0 1 0 0 1

lost arbritation

- **Arbitration Process (Step-by-Step)**
  1. Two devices send **START condition simultaneously**
  2. Both begin transmitting bits on SDA
  3. One device sends **HIGH (1)**, other sends **LOW (0)**
  4. Bus level becomes **LOW**
  5. Device that sent **HIGH** detects mismatch
  6. That device **loses arbitration**

- **Result of Arbitration**
  - **Winning device**
    - Continues as **Master**
    - Completes communication
  - **Losing device**
    - Stops transmitting
    - **Falls back to Slave mode**
    - Waits for bus to become free

I²C – Error Conditions

1. Bus Error

- Occurs when a **START or STOP condition appears while the bus is already busy**
- This indicates **illegal bus activity**
- **Usually caused by:**
  - Noise on the bus
  - Improper timing
  - Software error
- **In Master mode:**
  - Master **continues owning the bus**
  - Bus lines are **not released automatically**
  - Software must handle recovery
  - **Meaning:** Communication sequence is violated

I²C – Error Conditions

2. **ACK Error (Acknowledge Error)**
   - After sending **address or data**, transmitter releases SDA
   - Receiver must pull SDA **LOW** to send ACK
   - If **no device responds**, SDA remains HIGH
   - This condition is detected as **ACK error**
   - **Indicates:**
     - No slave present
     - Wrong slave address
     - Slave not ready or powered OFF

3. **Arbitration Lost Error**
   - Happens in **multi-master systems**
   - When another master sends **LOW (0)** while current master sends **HIGH (1)**
   - Current master **loses arbitration**
   - **Result:**
     - Loser stops transmission
     - Automatically switches to **slave mode**
     - Winning master continues communication
     - This error is **non-destructive**

I²C – Error Conditions

4. Read Overrun Error

- Occurs when:
  - New data is received
  - Previous data was **not read from the receive register**
- New data **overwrites old data**
- **Cause:**
  - Slow software response
  - Interrupt not serviced in time
  - Results in **data loss**

STM32 I2C

- Multi-master capability: the same interface can act as Master or Slave
- I2C Master features: Clock generation, Start and Stop generation
- I2C Slave features: Dual Addressing Capability, Stop bit detection
- Generation and detection of 7-bit addressing and General Call
- Supports different communication speeds: 100 kHz and 400 kHz
- Status flags: Transmitter/Receiver mode flag, End-of-Byte transmission flag, I2C busy
- Error flags:
  - Arbitration lost condition for master mode
  - Acknowledgment failure after address/ data transmission
  - Detection of misplaced start or stop condition
  - Overrun/Underrun if clock stretching is disabled
- 2 Interrupt vectors: Successful Address/Data communication, Error
- Clock stretching

PCF8574 – I²C I/O Expander

- **What is PCF8574?**
  - **PCF8574** is an **I²C-based I/O Expander**
  - It is used to **increase the number of GPIO pins**
  - Commonly used with:
    - LCD displays (I²C LCD module)
    - LEDs
    - Switches
    - Relays
  - Very useful when microcontroller GPIO pins are limited

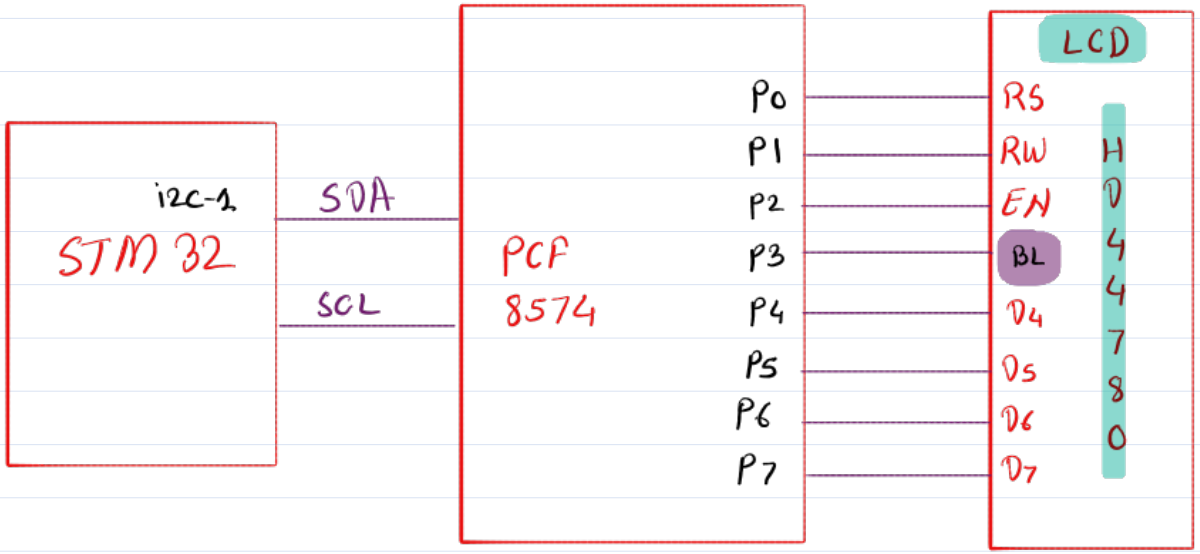
- **Communication Type**
  - Works as an **I²C Slave device**
  - Controlled by an **I²C Master** (STM32, Arduino, etc.)
  - Communication uses **SDA and SCL only**

- **Slave Address**
  - PCF8574 has **two possible slave addresses**:
  - 0x4E → Write operation 0x4F → Read operation
  - These are **8-bit I²C addresses** (including R/$\overline{W}$ bit)

2 I o

- **No Internal Address**
  - PCF8574 **does NOT have internal registers**
  - **No internal address byte is required**
  - Data sent by master directly controls the **I/O pins**
- **Data Meaning**
  - Each bit of the data byte corresponds to **one I/O pin**
  - Example:
    - Data Byte: 1010 1100
    - Pin State: P7 P6 P5 P4 P3 P2 P1 P0
      - 1 → Pin HIGH / Input
      - 0 → Pin LOW / Output
- **CF8574 is Easy to Use**
  - No register addressing
  - Simple write and read
  - Ideal for beginners
  - Perfect for **I²C LCD interfacing**
- STM32 controls a **16x2 LCD** using only **2 I²C pins** via PCF8574
- PCF8574 expands GPIO using I²C and needs only slave address no internal register addressing.

* Connection



* To write a 4-bit data
1. RS = 0 (Instr) | RS = 1 (Data)
2. RW̄ = 0 (write) | RW̄ = 1 (Read)
3. EN = 1
4. Write 4-bits of data (D_4-D_7)
   - data write in upper nibble (D_4-D_7)
   - Cmd write in lower nibble (D_0-D_3)
5. Delay 1 ms / No operation
6. EN = 0
   ↳ Enable high → low for LCD Read Reg. data.

e.g. Val = 0x28
- pass these value in 4 bit mode
- 4 bit (D_4-D_7) only for data / cmd

high_nibble = Val & 0xF0;

```
  0 0 1 0  1 0 0 0
& 1 1 1 1  0 0 0 0
-----------------
  0 0 1 0  0 0 0 0
```

lower_nibble = Val << 4;
lower_nibble = lower_nibble & 0xF0

```
  0 0 1 0  1 0 0 0   <<4
  1 0 0 0  0 0 0 0
& 1 1 1 1  0 0 0 0
-----------------
  1 0 0 0  0 0 0 0
```

* To write a 8 bit data eg 0x28 (instr)
- divid into 2 nibble
  high = Val & 0xF0 | 0x28
  low = Val << 4
  low = low & 0xF0; | 0x28
  ↳ data ↳ instr
- writ high nibble
- writ low nibble
- Wait for busy flag (Not Needed for low Speed MC)

Thank you!

Kiran Jaybhave

email – kiran.jaybhave@sunbeaminfo.com