I2C PROTOCOL

Understanding I2C Protocol and How It Differs from SPI

- What is I2C?
  I^2C (Inter-Integrated Circuit) is a synchronous, multi-master, multi-slave, serial communication protocol used for short-distance communication between integrated circuits. It was developed by Philips (now NXP Semiconductors) and is commonly used in embedded systems.
- Key Differences Between I2C and SPI

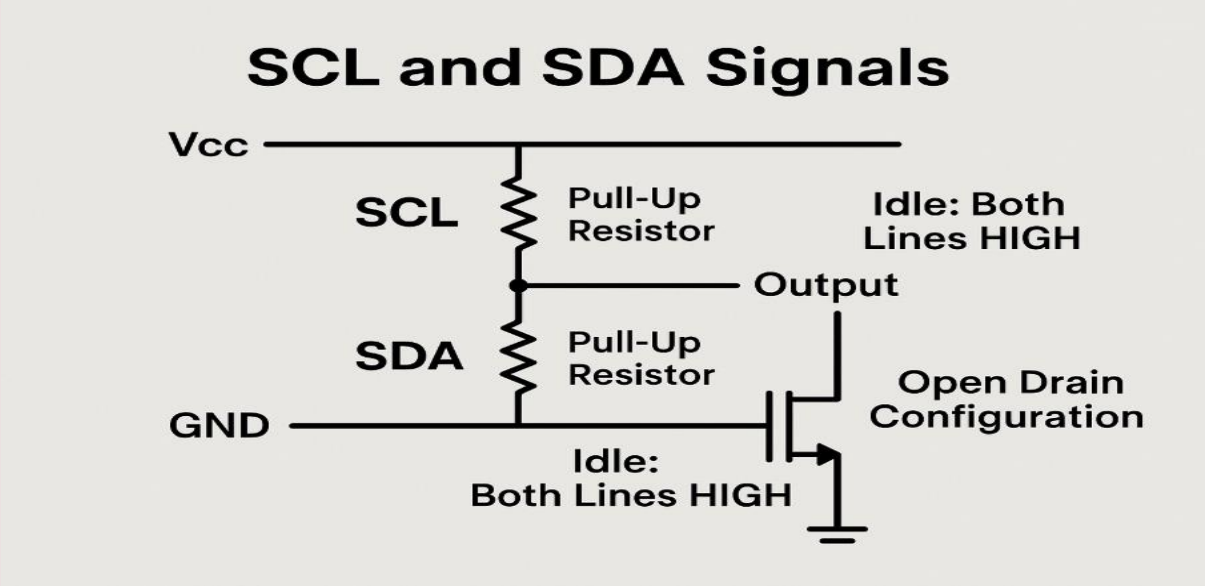
Feature	I2C	SPI
Specification	Defined by NXP	No dedicated global spec
Multi-Master Support	✅ Yes (hardware-managed arbitration)	❌ No, requires software handling
Automatic Acknowledgment (ACK)	✅ Yes	❌ No (software implementation needed)
Pins Required	2 (SCL & SDA)	4+ (SCLK, MOSI, MISO, SS)
Addressing	✅ Yes (slave address-based communication)	❌ No (uses chip select pin)
Data Transfer Mode	Half-duplex	Full-duplex
Speed	Up to 4 MHz (Ultra-Fast Mode)	Faster (Peripheral Clock / 2)
Clock Control	Slave can stretch the clock	Master controls the clock
Best Use Cases	Sensor communication, EEPROMs	High-speed data transfer, displays

- Why Choose I2C?
  - Fewer Pins: Connect multiple devices using just SCL & SDA.
  - Automatic ACK: Reduces software overhead.
  - Multi-Master Support: More flexible than SPI in complex systems.
  - Clock Stretching: Slaves can hold the clock if busy.

- Why Choose SPI?
  - Faster Data Transfer (Up to 50x faster than I2C in some cases).
  - Full Duplex Communication: Can send and receive data simultaneously.
  - Ideal for High-Speed Applications: Used in memory chips, screens, and high-speed peripherals.

Understanding SCL and SDA Signals in I2C

- SCL (Serial Clock) & SDA (Serial Data) are Bidirectional Lines
  - Both SCL & SDA are connected to Vcc via pull-up resistors.
  - When the bus is idle, both lines remain HIGH.
- Open Drain Configuration
  - The I2C pins must be configured as Open Drain.
  - P-MOS is absent, and only N-MOS is used in the circuit.
  - A pull-up resistor is required to achieve both LOW and HIGH states.
- Pull-Up Resistor Selection
  - Correct resistor values are crucial for proper operation.
  - Can use internal or external pull-up resistors.
- Troubleshooting Tip:
  - If I2C is not working, check the voltage across SDA & SCL.
  - It should be 3.3V when idle.

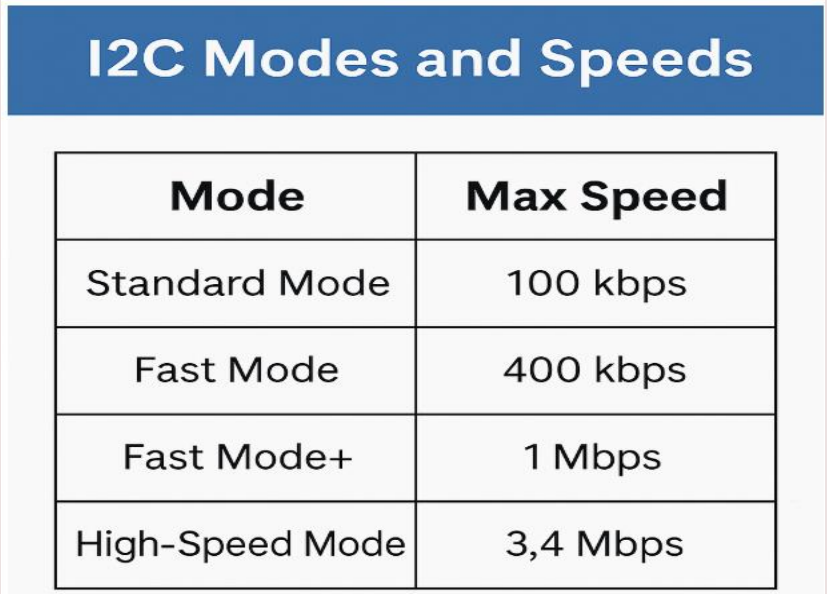


Understanding I2C Modes and Their Speeds

- Different I2C Modes & Data Rates

I2C supports four different modes, each with a different maximum data rate:

Mode	Max Speed	Supported by ST?
Standard Mode	100 Kbps	✅ Yes
Fast Mode	400 Kbps	✅ Yes
Fast Mode+	1 Mbps	⚠️ Some variants
High-Speed Mode	3.4 Mbps	❌ Not supported by ST



Understanding the Basics of I2C Protocol

- I2C Data Transfer Process
  - Master always initiates communication on the SDA line.
  - Data transfer starts with a START condition followed by an Address Phase (8 bits).
  - 7-bit Slave Address + 1-bit Read/Write (R/W) bit.
    - 0 (Write) → Master sends data to Slave.
    - 1 (Read) → Master receives data from Slave.
  - Each byte (8 bits) must be followed by an ACK (Acknowledge) bit.
- I2C Write Operation

1. Master sends START condition
2. Master sends 7-bit slave address + R/W bit = 0 (Write mode)
3. Slave responds with ACK
4. Master sends 1 byte of data
5. Slave acknowledges the received data
6. Master can send more bytes (each followed by an ACK)
7. Master generates STOP condition (Releases the bus)

- I2C Read Operation

1. Master sends START condition
2. Master sends 7-bit slave address + R/W bit = 1 (Read mode)
3. Slave responds with ACK
4. Slave sends data byte
5. Master acknowledges received data
6. Master can keep reading or send STOP condition

- START & STOP Conditions
  - START Condition: Master takes control of the bus.
  - STOP Condition: Master releases the bus, allowing another master to initiate communication.

Understanding START and STOP Conditions in I2C

- What Are START and STOP Conditions?
  - Every I2C transaction begins with a START condition and ends with a STOP condition.
  - **START Condition (S):**
    - SDA transitions from HIGH to LOW while SCL is HIGH.
    - Indicates that the bus is now busy.
  - **STOP Condition (P):**
    - SDA transitions from LOW to HIGH while SCL is HIGH.
    - Releases the bus, making it free for other masters.
- Address Phase & Acknowledgment
  - **After the START condition, the master sends an 8-bit address.**
    - 7-bit Slave Address + 1-bit R/W (Read/Write) Bit.
    - R/W = 0 → Write Operation (Master → Slave)
    - R/W = 1 → Read Operation (Slave → Master)
  - **Slave sends an ACK (Acknowledgment) after receiving a valid address.**
- Repeated START Condition
  - Master can generate another START condition without a STOP condition.
  - Used when switching from Write to Read mode without releasing the bus.
  - Keeps the bus busy to prevent other masters from taking control.
- Bus Status in I2C
  - Bus is BUSY after a START condition.
  - Bus is FREE after a STOP condition.
  - Other masters can use the bus after a STOP condition.
- Master-Slave Mode in Microcontrollers
  - Most microcontrollers support both Master and Slave modes.
  - If the MCU generates a START condition, it automatically becomes the Master.
  - After generating a STOP condition, it returns to Slave mode.
  - Sensors and peripherals are usually fixed as Slaves.

Understanding the Address Phase in I2C

- Address Phase Breakdown
  - After the START condition, the Address Phase (8 bits) follows.
  - First 7 bits → Slave Address
  - 8th bit → Read/Write (R/W) Bit
    - 0 (Write) → Master sends data to Slave
    - 1 (Read) → Master receives data from Slave
- ACK (Acknowledge) & NACK (Not Acknowledge)
  - 9th Clock Cycle is reserved for ACK/NACK.
  - ACK (Acknowledgment) → SDA line pulled LOW by the receiver.
  - NACK (Not Acknowledge) → SDA line remains HIGH at the 9th clock cycle.
  - Master always generates the clock for ACK/NACK signals.
- What Happens After NACK?

- If a NACK is received, the Master can either:
  ✅ Send a STOP Condition → End the transaction.
  ✅ Send a Repeated START → Retry or start a new transfer.

- **Master Writing Data to Slave (Write Operation)**
  1. Master generates a START condition.
  2. Master sends 7-bit Slave Address + R/W = 0 (Write mode).
  3. Slave sends ACK (acknowledging the address).
  4. Master sends 1st byte of data → Slave ACKs.
  5. Master sends 2nd byte of data → Slave ACKs.
  6. Master sends 3rd byte of data → Slave ACKs.
  7. Master generates STOP condition (releases the bus).

  ✅ Each byte sent by Master is followed by an ACK from Slave.

- Master Reading Data from Slave (Read Operation)
  1. Master generates a START condition.
  2. Master sends 7-bit Slave Address + R/W = 1 (Read mode).
  3. Slave sends 1st byte of data → Master ACKs.
  4. Slave sends 2nd byte of data → Master ACKs.
  5. Slave sends 3rd byte of data → Master NACKs (signals end of data reception).
  6. Master generates STOP condition (releases the bus).

  ✅ Master ACKs every received byte except the last one (NACK).

- Key Differences Between Read & Write

Operation	Master Sends	Master Receives	Final Signal
Write	Data Bytes	ACK from Slave	STOP Condition
Read	Address Only	Data Bytes from Slave	NACK & STOP

Understanding Repeated START (Sr) in I2C

- What is a Repeated START?
  - Repeated START (Sr) means generating a new START condition without a STOP condition.
  - Why use it? To prevent releasing the bus between a write and a read operation.
- Example: Reading from an EEPROM at Address 0x45
- Without Repeated START (STOP in between)

1. Master generates START condition
2. Master sends Slave Address (Write Mode, R/W = 0)
3. Slave (EEPROM) ACKs
4. Master writes the memory address (0x45) to EEPROM
5. Master generates STOP condition (Releases Bus ❌)
6. Master generates a new START condition
7. Master sends Slave Address (Read Mode, R/W = 1)
8. EEPROM sends data from 0x45
9. Master generates STOP condition

NOTE: Issue - Master releases the bus after writing the address (0x45).

Risk: Another master can take control before the read operation completes.

- With Repeated START (No STOP in between)

1. Master generates START condition
2. Master sends Slave Address (Write Mode, R/W = 0)
3. Slave ACKs
4. Master writes the memory address (0x45)
5. Master generates REPEATED START (Sr) instead of STOP
6. Master sends Slave Address (Read Mode, R/W = 1)
7. EEPROM sends data from 0x45
8. Master generates STOP condition (Only after receiving data)

- Advantage:
  - Master keeps control of the bus until the entire operation is complete.
  - No chance for another master to interrupt the transaction.
  - Useful in multi-master environments.

- When to Use Repeated START?
  - ✔ When performing a Write followed by a Read operation.
  - ✔ In multi-master I2C setups to prevent bus conflicts.
  - ❌ Not necessary in a single-master system, but still good practice.

Starting I2C Driver Development in ST Microcontroller

- Step 1: Identify Available I2C Peripherals
  - Refer to the Reference Manual of your microcontroller.
  - ST Microcontroller has 3 I2C peripherals (I2C1, I2C2, I2C3).
  - All I2C peripherals are connected to the APB1 bus.

- Step 2: Understand the I2C Block Diagram

  The I2C peripheral architecture consists of:

Component	Description
SDA & SCL Pins	Main communication lines for I2C
Noise Filters	Removes glitches/spikes on SDA & SCL
Data Register (DR)	Stores data for transmission/reception
Shift Register	Transfers data between internal and external devices
Clock Control Block (CCR)	Controls SCL frequency using the Clock Control Register (CCR)

Component	Description
Own Address Register (OAR)	Stores the Slave Address when in Slave Mode
Control Registers (CR1 & CR2)	Configures various parameters for the I2C peripheral
Status Registers (SR1 & SR2)	Provides status flags for I2C operations

- Step 3: Configure Key Registers for I2C Driver
  - Clock Control Register (CCR) → Controls SCL frequency.
  - Control Registers (CR1 & CR2) → Enables I2C & configures settings.
  - Status Registers (SR1 & SR2) → Monitors transmission & reception status.
  - Own Address Register (OAR1 & OAR2) → Used when the microcontroller is a slave.

Implementing the I2C_Init() API in STM32 I2C Driver

- Steps for I2C Initialization

Before enabling the I2C peripheral, we need to configure several key parameters:

Step	Description
1. Enable I2C Peripheral Clock	The I2C clock must be enabled before configuration.
2. Configure I2C Mode	Standard Mode (100 KHz), Fast Mode (400 KHz), or Ultra Fast Mode.
3. Set Serial Clock Speed (SCL)	Defines how fast data is transmitted over the I2C bus.
4. Configure Device Address (if in Slave Mode)	Each slave must have a unique 7-bit address.
5. Enable Acknowledge (ACK) Feature	ACK is disabled by default, so it must be enabled manually.
6. Set Rise Time (TRISE)	Ensures proper timing for signal transitions.

Step	Description
Configuration	
7. Enable the I2C Peripheral	Enable I2C only after completing all configurations.

- Key Considerations for I2C Clock Configuration
  - The higher the frequency, the shorter the cable length should be.
  - Long cables = Lower frequency (e.g., 50 KHz) for stable communication.
  - Faster speeds like 400 KHz or 1 MHz require shorter cables and well-matched impedance.

- I2C Register References for Initialization

1. Control Register 2 (CR2) → Set Peripheral Clock Frequency.
2. Clock Control Register (CCR) → Configure I2C Speed.
3. Own Address Register (OAR1) → Store Slave Address (if applicable).
4. Control Register 1 (CR1) → Enable/Disable Peripheral & Acknowledgment.
5. TRISE Register (TRISE) → Set rise time for SDA/SCL signals.

Understanding I2C Serial Clock Configuration in STM32

- Registers Involved in I2C Clock Configuration

To set the Serial Clock (SCL) frequency, we must configure two registers:

1. CR2 (Control Register 2) → Set Peripheral Clock Frequency (FREQ field)
2. CCR (Clock Control Register) → Set SCL frequency based on mode (SM/FM)

- Step 1: Configure Peripheral Clock Frequency (FREQ field in CR2)
  - APB Bus Clock must be stored in FREQ field of CR2
    - APB1 Bus Clock = 16 MHz
    - Set FREQ = 16

```
I2Cx->CR2 |= (16 << I2C_CR2_FREQ);
```

- Step 2: Calculate and Set CCR Value for Standard Mode (100 KHz)

❖ Formula for Standard Mode (SM, 100KHz)

$$CCR = \frac{T_{high}}{T_{PCLK1}}$$

$$CCR = \frac{5\mu s}{62.5ns} = 80 \text{ (0x50 in HEX)}$$

✓ Set CCR = 80 (0x50) in CCR Register

I2Cx->CCR = 80;

- Step 3: Configure CCR for Fast Mode (200 KHz & 400 KHz)

❖ Fast Mode with Duty = 0 (Tlow = 2 * Thigh)

$$CCR = \frac{T_{high} + T_{low}}{3 \times T_{PCLK1}}$$

Example for **200 KHz** (5 µs period)

$$CCR = \frac{5\mu s}{3 \times 62.5ns} = 26$$

I2Cx->CCR = 26; // For 200 KHz

Example for **400 KHz** (2.5 µs period)

$$CCR = \frac{2.5\mu s}{3 \times 62.5ns} = 13$$

I2Cx->CCR = 13; // For 400 KHz

Understanding Clock Stretching in I2C

- What is Clock Stretching?
  - Clock stretching occurs when a device holds the SCL line LOW to pause the I2C communication.
  - The master determines the clock speed, but a slave can stretch the clock if it needs more time to process data.
  - I2C communication resumes when the slave releases SCL (clock goes HIGH again).

- Why is Clock Stretching Important?
- Slaves use clock stretching when:
  - ✔ They need more time to process received data.
  - ✔ They are busy handling another operation.
  - ✔ They are slower than the master's clock speed.
- Prevents Data Loss:
  - Without clock stretching, master might misinterpret delayed ACK as NACK, leading to transmission failure.

- Example: Normal vs Clock-Stretched Scenario

✅ No Clock Stretching (Normal Operation)

1. Master sends data byte (8 clock cycles).
2. Slave receives data successfully.
3. Slave sends ACK in the 9th clock cycle.

✅ With Clock Stretching (Slave Needs More Time)

1. Master sends data byte (8 clock cycles).
2. Slave is busy, so it holds SCL LOW (clock stretch).
3. Master pauses communication automatically.
4. When ready, slave releases SCL and sends ACK.

- How to Enable Clock Stretching?
  - I2C hardware handles stretching automatically—no manual intervention needed.
  - You only need to enable the feature in the I2C configuration registers.

  NOTE: Clock stretching is crucial when dealing with slow-response peripherals like EEPROMs, ADCs, and sensors.

Implementing I2C_Init() API: Configuring Clock & Prescalars

- Steps to Configure I2C Clock in ST

  Before initializing I2C, we need to:

  1. Configure CR1 Register → Enable/Disable Acknowledge (ACK).
  2. Set Peripheral Clock Frequency (FREQ field in CR2).
  3. Calculate PCLK1 (APB1 Clock) Value Dynamically.
  4. Use CCR Register to Set SCL Frequency.
  5. Use TRISE Register for Signal Rise Time Configuration.

- Step 1: Configure CR1 (Acknowledge Control)

  Bit 10 (ACK) → Controls automatic acknowledgment.

  ```
  uint32_t tempreg = 0;

  tempreg |= (pI2CHandle->I2C_Config.I2C_AckControl << I2C_CR1_ACK);

  pI2CHandle->pI2Cx->CR1 = tempreg;
  ```

- Step 2: Calculate PCLK1 for FREQ Configuration in CR2

❖ Steps to Calculate APB1 Clock (PCLK1):

1. Get system clock source (HSI, HSE, or PLL).
2. Divide system clock using AHB prescaler.
3. Divide AHB clock using APB1 prescaler.
4. Return final PCLK1 value.

- Step 3: Implement RCC_GetPCLK1Value() Function

```
uint32_t RCC_GetPCLK1Value(void)
{
   uint32_t pclk1, SystemClk;
   uint8_t clksrc, temp;

   // Get System Clock Source
   clksrc = ((RCC->CFGR >> 2) & 0x3); // Bits 2-3 for system clock switch status

   if (clksrc == 0)      SystemClk = 16000000;  // HSI = 16MHz
   else if (clksrc == 1) SystemClk = 8000000;   // HSE = 8MHz
   else   SystemClk = RCC_GetPLLOutputClock();  // PLL Clock (if used)

   // Get AHB Prescaler
   temp = ((RCC->CFGR >> 4) & 0xF); // Bits 4-7 for AHB prescaler
   uint8_t AHB_PreScaler[8] = {2, 4, 8, 16, 64, 128, 256, 512};

   uint32_t ahb_prescaler = (temp < 8) ? 1 : AHB_PreScaler[temp - 8];

   // Get APB1 Prescaler
   temp = ((RCC->CFGR >> 10) & 0x7); // Bits 10-12 for APB1 prescaler
   uint8_t APB1_PreScaler[4] = {2, 4, 8, 16};

   uint32_t apb1_prescaler = (temp < 4) ? 1 : APB1_PreScaler[temp - 4];

   // Calculate final PCLK1
   pclk1 = (SystemClk / ahb_prescaler) / apb1_prescaler;
   return pclk1;
}
```

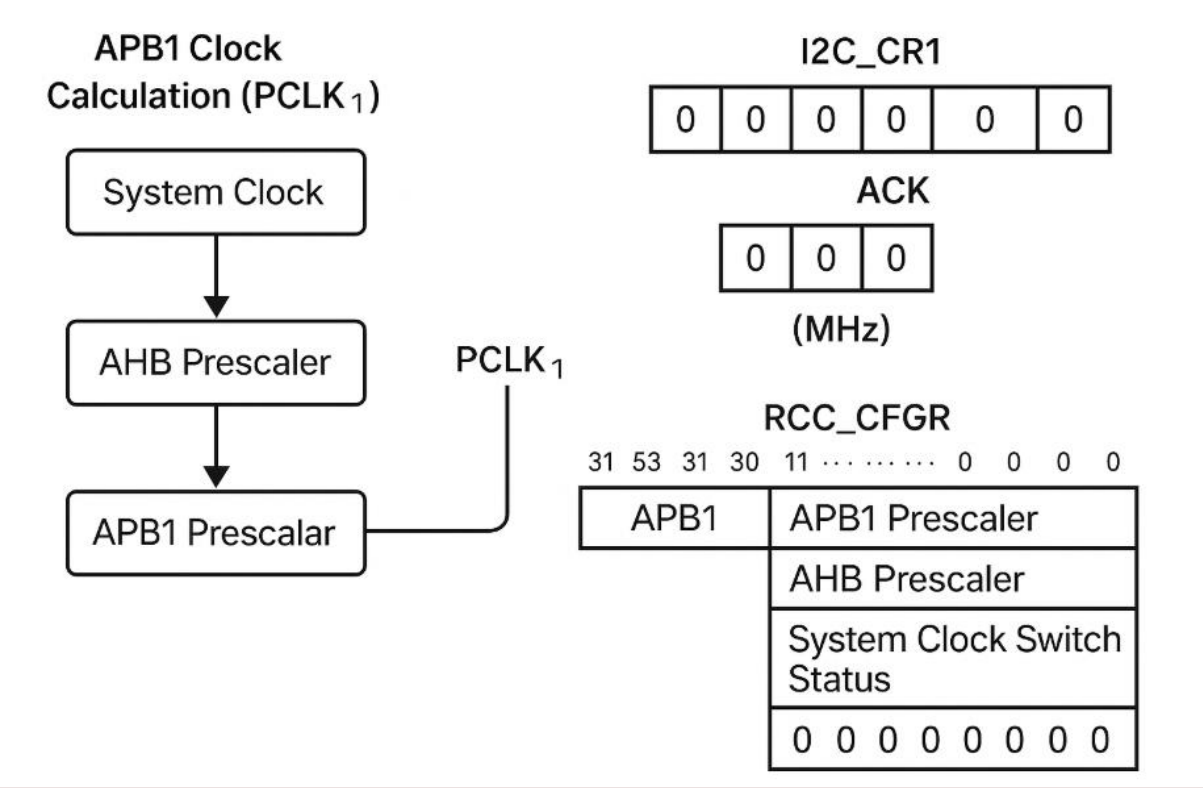
- Step 4: Program FREQ Field in CR2 Register

```
uint32_t pclk1 = RCC_GetPCLK1Value();

pI2CHandle->pI2Cx->CR2 |= (pclk1 / 1000000U); // Set FREQ field in MHz
```

✅ APB1 = 16 MHz → Set FREQ = 16

✅ APB1 = 8 MHz → Set FREQ = 8



Implementing CCR (Clock Control Register) Configuration in I2C_Init()

➢ Steps to Configure CCR (Clock Control Register) in ST MCU I2C

The CCR register is crucial for setting the serial clock (SCL) speed in Standard Mode (SM) or Fast Mode (FM).

- ✅ CCR is 12 bits (0-11) → Used for Clock Generation
- ✅ Bit 15: Mode Selection (0 = Standard Mode, 1 = Fast Mode)
- ✅ Bit 14: Duty Cycle Selection (For Fast Mode Only)

🔹 Step 2: Check the Speed Mode (Standard or Fast Mode)

✅ **Standard Mode (<=100 KHz)**

- Set **Bit 15 = 0** (Standard Mode).
- **Use the formula:**

$$CCR = \frac{F_{PCLK1}}{2 \times F_{SCL}}$$

✅ **Fast Mode (>100 KHz)**

- Set **Bit 15 = 1** (Fast Mode).
- Configure **Duty Cycle (Bit 14)**.
- **Use the appropriate formula based on Duty Cycle:**

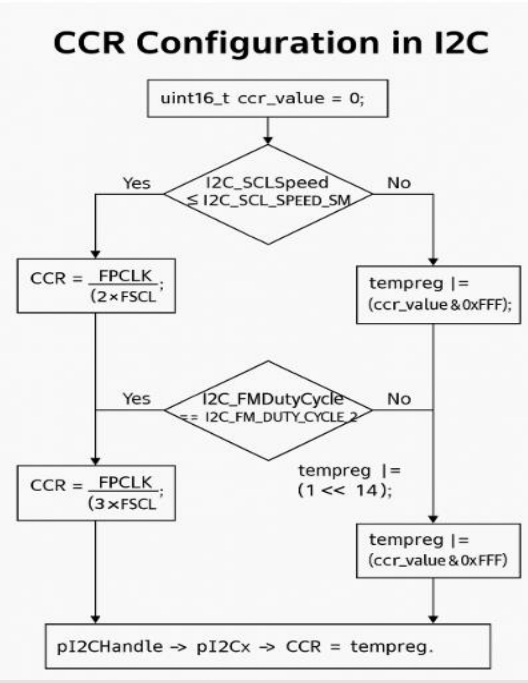
✔ **Duty = 0 (Tlow = 2 × Thigh):**

$$CCR = \frac{F_{PCLK1}}{3 \times F_{SCL}}$$

✔ **Duty = 1 (Tlow = 1.8 × Thigh, for max speed 400KHz):**

$$CCR = \frac{F_{PCLK1}}{25 \times F_{SCL}}$$

```
if (pI2CHandle->I2C_Config.I2C_SCLSpeed <= I2C_SCL_SPEED_SM)
{
   ccr_value = (RCC_GetPCLK1Value() / (2 * pI2CHandle->I2C_Config.I2C_SCLSpeed));
   tempreg |= (ccr_value & 0xFFF);  // Mask to keep only 12 bits
}
else
{
            tempreg |= (1 << 15); // Set Fast Mode
            if (pI2CHandle->I2C_Config.I2C_FMDutyCycle == I2C_FM_DUTY_CYCLE_2)
            {
               ccr_value = (RCC_GetPCLK1Value() / (3 * pI2CHandle->I2C_Config.I2C_SCLSpeed));
            }
            else
            {
               ccr_value = (RCC_GetPCLK1Value() / (25 * pI2CHandle->I2C_Config.I2C_SCLSpeed));
               tempreg |= (1 << 14);  // Set Duty Cycle 16/9
            }
            tempreg |= (ccr_value & 0xFFF);  // Mask to keep only 12 bits
}
```



Master Sending Data in I2C - Understanding the Transfer Sequence

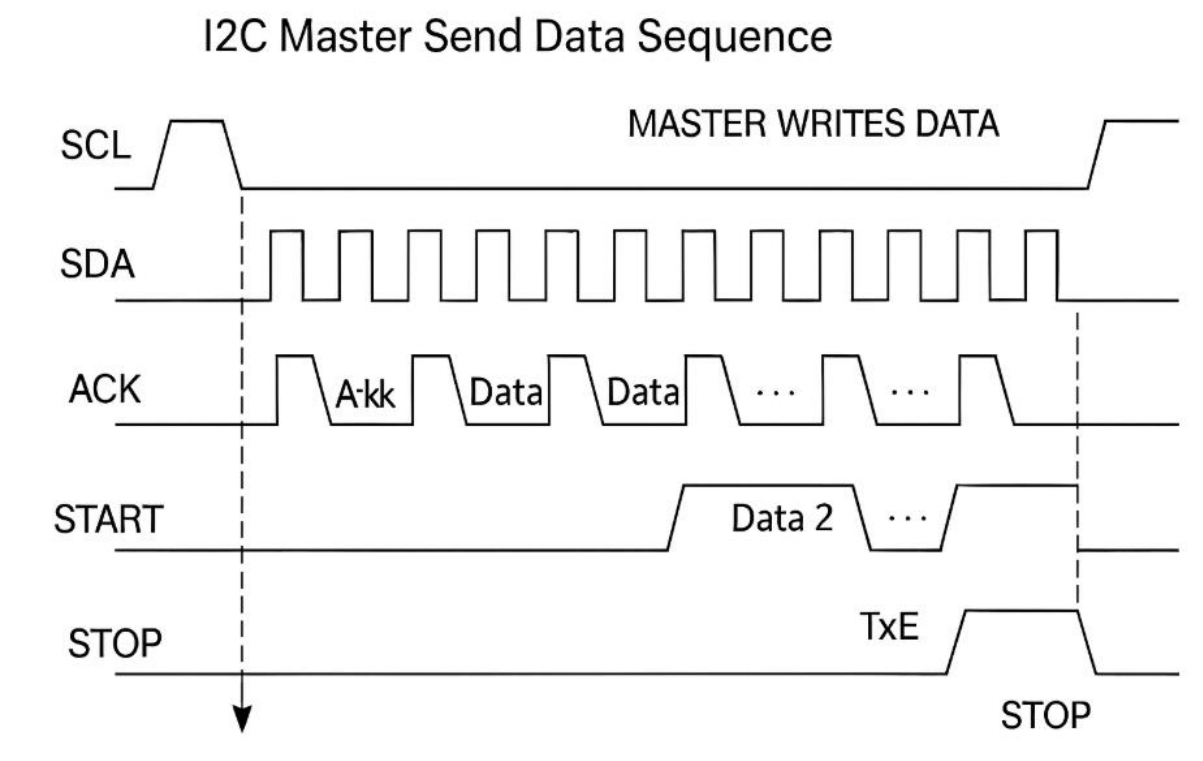
---

- Steps in I2C Master Data Transmission

Step	Action	Event Flag
1. Generate START Condition	Master initiates communication.	EV5 (SB=1)
2. Address Phase	Master sends 7-bit Slave Address + Write bit (0).	EV6 (ADDR=1)
3. Data Transmission Begins	Master writes first byte into DR (Data Register).	EV8_1 (TxE=1)
4. Data is Moved to Shift Register	Data moves from DR to Shift Register.	EV8 (TxE=1)
5. Slave Sends ACK	If slave acknowledges, next byte can be sent.	TxE remains 1
6. Data Transfer Continues	Master sends all bytes one by one.	ACK Received
7. Wait for Last Byte Transfer Completion	Ensure last byte is transmitted.	EV8_2 (TxE=1, BTF=1)
8. Generate STOP Condition	Master ends communication.	TxE & BTF cleared

---

- Clock Stretching During Transmission
  - The slave may stretch the clock if it needs more processing time.
  - The BTF flag (Byte Transfer Finished) is set when DR and Shift Register are empty.
  - Clock stretching prevents under-run conditions.



Implementing I2C Master Send Data API (I2C_MasterSendData)

- Steps for Implementing I2C_MasterSendData() API

Step	Action	Function/Flag Used
1. Generate START Condition	Master initiates communication.	I2C_GenerateStartCondition()
2. Wait for SB Flag	Confirm START condition is generated.	I2C_GetFlagStatus(pI2Cx, I2C_FLAG_SB)
3. Send Slave Address	Master sends 7-bit Slave Address + Write bit (0).	pI2Cx->DR = SlaveAddr;
4. Wait for ADDR Flag	Confirm slave address acknowledgment.	I2C_GetFlagStatus(pI2Cx, I2C_FLAG_ADDR)

Step	Action	Function/Flag Used
5. Clear ADDR Flag	Read SR1 & SR2 to clear ADDR.	temp = pI2Cx->SR1; temp = pI2Cx->SR2;
6. Send Data Bytes	Write data to DR register.	pI2Cx->DR = *TxBuffer;
7. Wait for TxE Flag	Ensure DR is empty before next byte.	I2C_GetFlagStatus(pI2Cx, I2C_FLAG_TXE)
8. Wait for BTF Flag (Last Byte)	Ensure last byte transmission is complete.	I2C_GetFlagStatus(pI2Cx, I2C_FLAG_BTF)
9. Generate STOP Condition	End communication after last byte.	I2C_GenerateStopCondition(pI2Cx);

Implementing I2C Address Phase & Clearing ADDR Flag

- Steps for Address Phase in I2C_MasterSendData() API

Step	Action	Function/Flag Used
1. Shift Slave Address	Move 7-bit Slave Address Left	SlaveAddr << 1
2. Clear Read/Write Bit	Set to 0 for Write	SlaveAddr & ~(1)
3. Write to DR Register	Send Address + RW Bit	pI2Cx->DR = SlaveAddr;
4. Wait for ADDR Flag	Confirm Address Acknowledgment	I2C_GetFlagStatus(pI2Cx, I2C_FLAG_ADDR)
5. Clear ADDR Flag	Read SR1 & SR2 to clear ADDR	I2C_ClearADDRFlag(pI2Cx);

Implementing Data Transmission in I2C_MasterSendData() API

- Steps for Sending Data

Step	Action	Function/Flag Used
1. Send Data Until Length Becomes 0	Use while(length > 0)	I2C_FLAG_TXE
2. Wait for TXE = 1	Ensure Transmit Register is Empty	I2C_GetFlagStatus(pI2Cx, I2C_FLAG_TXE)
3. Write Data to DR Register	Send Data Byte	pI2Cx->DR = *pTxBuffer
4. Decrement Length & Increment Buffer	Move to Next Byte	pTxBuffer++
5. Wait for TXE = 1 & BTF = 1	Ensure Last Byte Sent	I2C_GetFlagStatus(pI2Cx, I2C_FLAG_BTF)
6. Generate STOP Condition	Terminate Communication	I2C_GenerateStopCondition(pI2Cx)

I2C TRISE Configuration in I2C Initialization (I2C_Init)

The TRISE (Trise Time Register) is an important part of I2C peripheral configuration, as it ensures that the rise time of SCL follows I2C specifications.

- ➢ Understanding the TRISE Register
- ❖ What is TRISE?
  - The TRISE register defines the maximum rise time for SCL in Standard Mode (SM) or Fast Mode (FM).
  - This ensures that SCL rises fast enough to be detected as HIGH.
- ❖ TRISE Calculation Formula:

$$TRISE = \left(\frac{T_{\text{rise max}}}{T_{\text{PCLK1}}} \right) + 1$$

or equivalently:

$$TRISE = (F_{\text{PCLK1}} \times T_{\text{rise max}}) + 1$$

where:

- $T_{\text{rise max}}$ is obtained from the **I2C Specification**.
- F_{PCLK1} is the **Peripheral Clock Frequency**.
- T_{PCLK1} is the time period of the **APB1 clock**.

📌 **I2C Specification Values:**

Mode	Max Rise Time $T_{\text{rise max}}$
Standard Mode (100kHz)	1000 ns (1µs)
Fast Mode (400kHz)	300 ns

✅ **Example Calculation (Standard Mode, APB1 Clock = 16MHz)**

$$TRISE = \left(\frac{1000ns}{62.5ns}\right) + 1 = 16$$

✅ **Example Calculation (Fast Mode, APB1 Clock = 16MHz)**

$$TRISE = \left(\frac{300ns}{62.5ns}\right) + 1 = 5$$

```
void I2C1_Init(void)
{
   RCC->AHB1ENR |= (1 << 1);  // Enable GPIOB clock
   RCC->APB1ENR |= (1 << 21); // Enable I2C1 clock
   // Configure PB6 (SCL) and PB9 (SDA) as Alternate Function (AF4)
   GPIOB->MODER |= (2 << (6 * 2)) | (2 << (9 * 2));
   GPIOB->OTYPER |= (1 << 6) | (1 << 9);  // Open-drain
   GPIOB->PUPDR |= (1 << (6 * 2)) | (1 << (9 * 2)); // Pull-up
   GPIOB->AFR[0] |= (4 << (6 * 4));  // PB6 AF4 (I2C1_SCL)
   GPIOB->AFR[1] |= (4 << ((9 - 8) * 4));  // PB9 AF4 (I2C1_SDA)
   I2C1->CR1 |= (1 << 15);  // Software Reset
   I2C1->CR1 &= ~(1 << 15); // Clear Reset
   I2C1->CR2 |= (16 << 0);  // PCLK1 = 16MHz
   I2C1->CCR = 80;  // Standard Mode, 100KHz
   I2C1->TRISE = 17; // 1000ns / (1/16MHz) + 1
   I2C1->CR1 |= (1 << 0);  // Enable I2C
}
```

```
void I2C1_Write(uint8_t *data, uint8_t len)
{
   while (I2C1->SR2 & (1 << 1));  // Wait until not busy
   I2C1->CR1 |= (1 << 8);  // Generate Start Condition
   while (!(I2C1->SR1 & (1 << 0))); // Wait for SB flag
   I2C1->DR = SLAVE_ADDR << 1;  // Send Slave Address + Write (0)
   while (!(I2C1->SR1 & (1 << 1))); // Wait for ADDR
   (void) I2C1->SR2; // Clear ADDR flag
   for (uint8_t i = 0; i < len; i++)
{
      while (!(I2C1->SR1 & (1 << 7))); // Wait for TXE
      I2C1->DR = data[i];  // Send data byte
   }
   while (!(I2C1->SR1 & (1 << 2))); // Wait for BTF
   I2C1->CR1 |= (1 << 9);  // Generate Stop Condition
}
void I2C1_GPIOInits(void)
{
   RCC->AHB1ENR |= (1 << 1);  // Enable GPIOB clock
   GPIOB->MODER |= (2 << (6 * 2)) | (2 << (9 * 2));  // PB6, PB9 as Alternate Function
   GPIOB->OTYPER |= (1 << 6) | (1 << 9);  // Open-drain mode
   GPIOB->PUPDR |= (1 << (6 * 2)) | (1 << (9 * 2)); // Pull-up resistors
   GPIOB->AFR[0] |= (4 << (6 * 4));  // PB6 AF4 (I2C1_SCL)
   GPIOB->AFR[1] |= (4 << ((9 - 8) * 4));  // PB9 AF4 (I2C1_SDA)
}
```

```
void I2C1_Inits(void)
{
   RCC->APB1ENR |= (1 << 21); // Enable I2C1 clock
   I2C1_Handle.pI2Cx = I2C1;
   I2C1_Handle.I2C_Config.AckControl = I2C_ACK_ENABLE;
   I2C1_Handle.I2C_Config.DeviceAddress = 0x61; // Arbitrary address for master
   I2C1_Handle.I2C_Config.DutyCycle = I2C_FM_DUTY_2;
   I2C1_Handle.I2C_Config.SCLSpeed = I2C_SCL_SPEED_SM;  // 100 kHz
   I2C1_Init(&I2C1_Handle);
   I2C1_PeripheralControl(I2C1, ENABLE);
}
```

Understanding I2C Master Reception Process

- Overview of Master Receiving Data from Slave
- Master Receives Data in Steps:

1. Master Generates START Condition → Becomes I2C Master.
2. Master Sends Address + Read Bit (1).
3. Slave ACKs the Address (If Available on Bus).
4. Master Reads Bytes One by One from Slave:
    - For Each Byte Received, Master Sends ACK (Except Last Byte).
5. For Last Byte, Master Sends NACK to Indicate End of Communication.
6. Master Generates STOP Condition to Complete the Transfer.

- Transfer Sequence for Master Receiver

1. START Condition → Master Initiates Communication.
2. Event EV5 → Start Bit (SB) is Set.
3. Address Phase → Master Sends (7-bit Address + Read Bit).

4. Slave ACKs the Address → Event EV6 (ADDR = 1).
5. Master Reads Bytes from Slave One by One:
    - For Each Byte, Master Sends ACK.
    - Event EV7 (RXNE = 1) → Data Register Full.
6. For Last Byte, Master Sends NACK.
7. Master Generates STOP Condition.

❖ Key Flags in Status Register (SR1):

Flag	Meaning
SB	Start Bit Set (EV5)
ADDR	Address Matched (EV6)
RXNE	Receive Buffer Not Empty (EV7)
STOP	Stop Condition Sent

➢ 3. Handling Different Data Length Cases

❖ Case 1: Receiving Only 1 Byte

- Start Condition → Address Phase.
- Set ACK = 0 (Disable Acknowledge).
- Set STOP = 1.
- Clear ADDR Flag.
- Wait for RXNE = 1 → Read Data.
- STOP Condition is Automatically Generated.

❖ Case 2: Receiving Multiple Bytes (e.g., 6 Bytes)

- Start Condition → Address Phase.
- Set ACK = 1.
- Clear ADDR Flag.
- Receive Bytes One by One:
    - For Last 2 Bytes:
        - When Length == 2, Disable ACK (ACK = 0).
        - Set STOP = 1.

- Receive Last Byte → NACK Sent.

Implementing I2C Master Receive Data API

➢ Steps to Implement I2C Master Reception

We will follow a step-by-step approach, similar to the send data API, but with additional logic for handling acknowledgment (ACK/NACK) properly.

✅ Step 1: Generate START Condition

- Call I2C_GenerateStartCondition() function.

✅ Step 2: Wait Until Start Condition is Completed

- Check the SB (Start Bit) flag in SR1 using I2C_GetFlagStatus().

✅ Step 3: Send Slave Address with Read Bit (1)

- Use I2C_ExecuteAddressPhase() function.
- Remember: SlaveAddr << 1 | 1 (Read bit set to 1).

✅ Step 4: Wait for Address Phase to Complete

- Check the ADDR (Address Matched) flag in SR1.

✅ Step 5: Handle Data Reception Based on Length 📍 Case 1: Receiving Only 1 Byte

- Disable ACK.
- Set STOP bit to automatically generate STOP condition.
- Clear ADDR Flag.
- Wait for RXNE (Receive Buffer Not Empty) Flag.
- Read Data from DR Register.

➢ Case 2: Receiving More Than 1 Byte

- Clear ADDR Flag.
- Read Bytes One by One:
  - If more than 2 bytes left → Send ACK.
  - If only 2 bytes left → Disable ACK and set STOP.
  - If last byte → Send NACK.

- Wait for RXNE Flag before reading data.

✅ Step 6: Generate STOP Condition

- Ensure STOP is set when receiving last byte.

✅ Step 7: Return from Function

- Master has successfully received all data.

```
void I2C_MasterReceiveData(I2C_Handle_t *pI2CHandle, uint8_t *pRxBuffer, uint32_t Len, uint8_t SlaveAddr)
{
   // 1. Generate START condition
   I2C_GenerateStartCondition(pI2CHandle->pI2Cx);

   // 2. Wait until SB (Start Bit) is set
   while (!I2C_GetFlagStatus(pI2CHandle->pI2Cx, I2C_FLAG_SB));

   // 3. Send Slave Address + Read Bit (1)
   I2C_ExecuteAddressPhaseRead(pI2CHandle->pI2Cx, SlaveAddr);

   // 4. Wait for ADDR flag (Address Matched)
   while (!I2C_GetFlagStatus(pI2CHandle->pI2Cx, I2C_FLAG_ADDR));

   if (Len == 1)
   {
      // 5a. Case: Receiving only 1 byte
      I2C_ManageAcking(pI2CHandle->pI2Cx, I2C_ACK_DISABLE);  // Disable ACK
      I2C_GenerateStopCondition(pI2CHandle->pI2Cx);  // Set STOP condition

      // Clear ADDR Flag
      I2C_ClearADDRFlag(pI2CHandle->pI2Cx);

      // Wait until RXNE (Receive Buffer Not Empty) flag is set
```

```
        while (!I2C_GetFlagStatus(pI2CHandle->pI2Cx, I2C_FLAG_RXNE));

        // Read Data from DR register
        *pRxBuffer = pI2CHandle->pI2Cx->DR;
    }
    else
    {
        // 5b. Case: Receiving more than 1 byte
        I2C_ClearADDRFlag(pI2CHandle->pI2Cx);

        // Receive all bytes
        for (uint32_t i = Len; i > 0; i--)
        {
            if (i == 2) // If only 2 bytes are left
            {
                // Disable ACK and set STOP
                I2C_ManageAcking(pI2CHandle->pI2Cx, I2C_ACK_DISABLE);
                I2C_GenerateStopCondition(pI2CHandle->pI2Cx);
            }

            // Wait until RXNE is set
            while (!I2C_GetFlagStatus(pI2CHandle->pI2Cx, I2C_FLAG_RXNE));

            // Read data
            *pRxBuffer = pI2CHandle->pI2Cx->DR;
            pRxBuffer++;
        }
    }
}
```

What We Learned

✅ I2C Events That Can Interrupt the Processor

- Start condition (SB)
- Address sent/received (ADDR)
- Stop condition (STOPF)
- Transmit buffer empty (TXE)
- Byte transfer finished (BTF)
- Receive buffer not empty (RXNE)
- Various error conditions (bus error, arbitration loss, ACK failure, etc.)

✅ Control Bits in CR Register

- ITBUFEN → Enables TXE/RXNE interrupts
- ITEVTEN → Enables event interrupts (start, stop, BTF, etc.)
- ITERREN → Enables error interrupts

✅ I2C Interrupt Mapping

- I2C1_EVENT IRQ (IRQ31)
- I2C1_ERROR IRQ (IRQ32)
- I2C2_EVENT IRQ (IRQ33)
- I2C2_ERROR IRQ (IRQ34)
- I2C3_EVENT IRQ (IRQ79)
- I2C3_ERROR IRQ (IRQ80)

Summary of I2C Errors and BTF Handling

- **Types of I2C Errors**

Error Type	Cause	Effect	Solution
Bus Error (BERR)	SDA line changes unexpectedly when SCL is high	I2C communication fails	Restart communication
Arbitration Loss (ARLO)	Two masters send data at the same time, and one loses arbitration	Master stops communication	Retry communication later
ACK Failure (AF)	No ACK received from the slave	Master stops transmission	Check slave presence, retry communication
Overrun Error (OVR)	New byte received before the previous byte is read from DR	Data loss, slave sends extra data	Read data register quickly, enable clock stretching
Under-run Error	Data register is not written before shift register finishes sending previous byte	Wrong data may be sent	Write data on time, use BTF flag to stretch clock
PEC Error	CRC mismatch in communication	Corrupted data	Re-transmit, check CRC settings
Timeout Error (TIMEOUT)	Slave holds SCL low for too long	I2C communication stalls	Detect and reset communication

- **Role of BTF Flag (Byte Transfer Finished)**

✅ BTF in TX (Transmission)

- If TXE = 1 (Data Register empty) and software doesn't load new data before Shift Register (SR) becomes empty, BTF = 1.
- Clock stretching prevents under-run (SCL is held low).
- Prevents sending wrong data by ensuring new data is loaded on time.

✅ BTF in RX (Reception)

- If RXNE = 1 (Receive Buffer not empty) and software doesn't read data before new data arrives, BTF = 1.
- Clock stretching prevents overrun.
- Ensures received data is not lost.

I2C Debugging Guide: Essential Tips for Troubleshooting I2C Communication Issues

---

➢ Step 1: Check the Voltage on SCL & SDA

❖ Why?

Before anything else, make sure both SCL and SDA lines are at 3.3V when idle. If the voltage is incorrect, the I2C bus won't function properly.

➢ **How to check?**

- Use a multimeter to measure voltage between SCL/SDA and GND.
- Use a logic analyzer to observe the signal levels.

➢ **Possible Issues & Fixes:**

Issue	Possible Cause	Solution
SCL/SDA not at 3.3V	Pull-up resistors missing or not enabled	Enable internal/external pull-ups
One line stuck at LOW	Slave device holding the bus low	Check for a faulty/sluggish slave
No signal change on SDA	Wrong GPIO configuration	Check alternate function settings

---

➢ Step 2: Debugging ACK Failure (No Acknowledgment from Slave)

❖ Why?

ACK failure is one of the most common I2C issues, where the slave does not respond to the master's request.

❖ How to identify?

- Connect a logic analyzer and check if the ACK bit is missing after the address phase.
- If NACK appears, then the slave isn't responding.

- Possible Issues & Fixes:

Issue	Possible Cause	Solution
Wrong Slave Address	Master sending an incorrect address	Double-check the 7-bit slave address
ACK Disabled on Slave	Slave not configured to acknowledge	Enable ACK in the I2C control register
Slave Not Initialized	Slave not powered up or not responding	Verify power and initialization sequence
Bus Contention	Another master is disrupting communication	Check for multiple masters and arbitration

- Quick Debugging Steps:

1. Verify slave address: Ensure the correct 7-bit address is used in your firmware.
2. Check slave ACK settings: Make sure ACK is enabled in I2C_CR1 register.
3. Inspect pull-ups: Measure SDA/SCL with a multimeter and enable internal/external pull-ups if needed.
4. Test with a different slave device to confirm if the issue is with the master or the slave.

- Step 3: Master Not Generating Clock (SCL Stuck)
- **Why?**
  If the I2C master is not generating a clock signal, then no data transfer will occur.
- **How to identify?**
  - Use a logic analyzer to check if the SCL line toggles.
  - If the SCL line remains LOW, it means the bus is stuck.
- Possible Issues & Fixes:

Issue	Possible Cause	Solution
Peripheral Clock Disabled	I2C clock is not enabled in RCC	Enable clock in RCC_APB1ENR
Incorrect GPIO Config	SCL/SDA not set as Alternate Function	Check GPIO configurations
Slave Holding the Bus	Slave stuck due to an internal issue	Reset the slave

Issue	Possible Cause	Solution
Clock Stretching	Slave needs more time to process data	Allow clock stretching in firmware

- Quick Debugging Steps:

1. Check if I2C peripheral is enabled:
2. RCC->APB1ENR |= (1 << 21); // Enable I2C1 Clock
3. Verify GPIO configuration:
4. GPIOB->MODER |= (2 << 12) | (2 << 18); // Set PB6 & PB9 to AF mode
5. GPIOB->AFR[0] |= (4 << 24); // AF4 for PB6
6. GPIOB->AFR[1] |= (4 << 4); // AF4 for PB9
7. Check if SCL is toggling using a logic analyzer.
8. Force a bus reset if needed:
9. GPIOB->ODR |= (1 << 6); // Set SCL HIGH
10. delay(10);
11. GPIOB->ODR &= ~(1 << 6); // Set SCL LOW
12. delay(10);

- Step 4: Analyzing Timing Issues
- **Why?**
  I2C requires accurate timing, especially in Fast Mode (400 kHz).
- **How to check?**
  - Logic analyzer to check SCL frequency.
  - Verify timing registers (CCR/TRISE).
- Possible Issues & Fixes:

Issue	Possible Cause	Solution
Clock too fast/slow	Incorrect I2C timing settings	Adjust CCR and TRISE values
Slave Overflows RX Buffer	Master sending data too fast	Enable clock stretching

- Step 5: Debugging with a Logic Analyzer

❖ Why?

A logic analyzer is the best tool for debugging I2C issues.

❖ What to check?

- Start Condition: Does SDA go LOW while SCL is HIGH?
- Address Phase: Is the correct slave address sent?
- ACK/NACK: Does the slave acknowledge the request?
- Stop Condition: Does SDA go HIGH while SCL is HIGH?