

I2C

25 June 2025 10:24

1. What is I2C?

Developed in 1982 by Philips(now NXP).

Primarily used for short distance data communications.

I2C is a **synchronous, half-duplex, multi-master** serial communication protocol used to communicate with:

- EEPROMs
- RTC (DS3231)
- Sensors (e.g., MPU6050)
- Displays (e.g., SSD1306 OLED)

-> Synchronous meaning, Both devices share a clock, so they know **when** to send and read data.

Opposite of asynchronous (like UART), which uses start/stop bits instead of a clock.

In I²C, the clock line (SCL) is controlled by the master, and both devices use it to synchronize data bits.

-> Half Duplex meaning, Communication happens in only one direction at a time, i.e Only one device talks at a time.

-> Multi Master meaning, More than one device can be a master on the same I²C bus.

Any master can initiate communication if the bus is free.

I²C handles bus arbitration if two masters try to talk at once.

Bus arbitration meaning, If two (or more) masters try to use the I²C bus at the same time, the protocol has a way to decide who gets control — without messing up communication.

So I²C lets all masters try, but only the one sending the lowest address (in binary) wins — because its bits produce more 0s early on.

-> Serial meaning, Data is sent one bit at a time, over one wire.

Each data bit on SDA is valid **only during a high pulse of SCL**.

2. Working of I2C :

Start Condition :

-> In the idle state, SDA and SCL are both high

-> The start condition occurs when a node :

First pulls SDA low, **while SCL is high**

-> This "claims the bus", Node is now the master, Prevents any other node from taking control of the bus.

-> Prevents any other nodes from taking control of the bus.

-> Master that has seized the bus also starts the clock, which is shared by both master and slave.

-> The Trise, CCR values determine the clk speed and rise time.

-> $T_{high} = CCR * T_{PCLK1}$ (CCR = 210, $T_{PCLK1} = 1/42\text{MHz}$)

-> $F_{SCL} = 1 / (2 * CCR * T_{PCLK1})$

Slave Address :

-> Each I2C node on a bus must have a unique fixed address.

-> Normally 7 bits long, MSB first. 10 bit addresses are also supported, but these are uncommon.

-> Address maybe hard coded. Address maybe (partially) configurable via external address lines or jumpers.

Timing Relationship between SDA and SCL :

-> SDA does not change between clock rising edge and clock falling edge.

-> During data transmission, SDA only transitions while SCL is low

-> An SDA transition when SCL is high, indicates a start or stop condition.

Read / Write bit :

-> Read / Write bit follows the slave address.

-> Set by master to indicate desired operation.

0 = Master wants to write data to slave.

1 = Master wants to read data from slave.

-> Often interpreted and /or decoded as part of the address byte.

Acknowledge bit :

-> Sent by the receiver of a byte of data

0 = acknowledgement (ACK)

1 = negative acknowledgement (NACK)

- > Recall that I2C in idle state is high.
- Lack of Response = NACK
- > Used after slave address and each data byte
- > ACK after data byte(s) confirms receipt of data.
- > ACK after slave address confirms that
 - A slave with that address confirms receipt of data.
 - The slave is ready to read/write data (depending on R/W bit)

Data Byte :

- > Data byte contains the information being transferred between master and slave
 - Memory or register contents, addresses etc.
- > Always 8 bit long. MSB will be the first bit.
- > Always followed by an ACK bit.
 - Set to zero by the receiver if data has been received properly.

Stop condition :

- > Stop condition indicates the end of data bytes.
 - First, SCL returns (and remains) high.
 - Then, SDA returns (and remains) high,
- > Recall that for data bytes, SDA only transitions when SCL (clk) is low.
 - SDA transition when SCL high = stop condition.
- > Bus becomes idle. No clk signal, Any nide can now use the start condition to claim the bus and begin a new communication.

Open drain:

- > Each line (SDA and SCL) is connected to voltage(Vcc or Vdd) via a "pull up" resistor.
 - One resistor per line, not per device.
- > Each I2C device contains logic that can open and close a drain.
- > When drain is "closed" , the line is pulled low(connected to ground).
- > When drain is "open" , the line is pulled high(connected to voltage).
- > I2C lines are high in the idle state, and hence also called an "open drain" system.

I2C Features in STM32F429Zi

25 June 2025 17:26

I2Cs in stm32f:

-> There 3 I2Cs in stm32f429, (this info will be useful while coding).

-> All the 3 I2C buses are connected to the APB1 bus, which runs on 45MHz clock.

SMBus = I²C + extra rules for better safety, reliability, and compatibility in systems like laptops and servers.

-> The Alternate Function code for this is AF4.

I2C1	I2C2	I2C3
PORT B	PORT F	PORT H
PB5 = I2C1_SMBA	PF0 = I2C2_SDA	PB7 = I2C3_SCL
PB6 = I2C1_SCL	PF1 = I2C2_SCL	PH8 = I2C3_SDA
PB7 = I2C1_SDA	PF2 = I2C2_SMBA	PH9 = I2C3_SMBA

It supports the standard mode (Sm, up to 100 kHz) and Fm mode (Fm, up to 400 kHz).

Features :

-> Parallel bus/ I2C protocol converter : STM32 can convert regular data from it's internal system into I2C compatible format and vice versa.

-> Multimaster capability : The same STM32 I²C block can **act as a Master (sender/controller) or a Slave (receiver/responder)**.

-> I2C Master Features : Clock Generation and Start/Stop generation.

-> I2C Slave Features :

- Programmable I2C Address detection
- Dual Addressing Capability to acknowledge 2 slave addresses
- Stop bit detection

-> Generation and detection of 7-bit/10-bit addressing and General Call

-> Supports different communication speeds:

- Standard Speed (up to 100 kHz)
- Fast Speed (up to 400 kHz)

-> Analog Noise filter: Blocks small voltage spikes on lines.

-> Programmable digital noise filter for STM32F42xxx.

-> Status flags:

- Transmitter/Receiver mode flag
- End-of-Byte transmission flag
- I2C busy flag

-> Error Flags :

- **Arbitration lost** (2 masters try to talk, one loses)
- **No ACK** received from slave : Acknowledgment failure after address/ data transmission
- Start/Stop sent at wrong time : Detection of misplaced start or stop condition
- **Overrun/Underrun** (data overflow if clock stretching disabled)

-> 2 Interrupt Vectors :

- 1 Interrupt for successful address/ data communication
- 1 Interrupt for error condition.

-> Optional Clock Stretching : Devices can **hold the clock line low** to pause communication if they're not ready.

-> 1-byte buffer with DMA capability : Data is stored in a **1-byte buffer**, and can be sent/received using **DMA** for speed and efficiency.

-> Configurable PEC (packet error checking) generation or verification:

- PEC value can be transmitted as last byte in Tx mode
- PEC error checking for last received byte.

-> SMBus 2.0 Compatibility:

- 25 ms clock low timeout delay : Timeout handling (e.g., line stuck low for >25 ms = error)
- 10 ms master cumulative clock low extend time
- 25 ms slave cumulative clock low extend time

- Hardware PEC generation/verification with ACK control
- Address Resolution Protocol (ARP) supported : Special addressing via **ARP**

More I2C Description

26 June 2025 10:04

- The interface can operate in one of the four following modes:
 - Slave transmitter • Slave receiver • Master transmitter • Master receiver
- By default, it operates in slave mode. The interface automatically switches from slave to master, after it generates a START condition and from master to slave, if an arbitration loss or a Stop generation occurs, allowing multimaster capability.
- Communication Flow :
 - In Master mode, the I2C interface initiates a data transfer and generates the clock signal.
 - A serial data transfer always begins with a start condition and ends with a stop condition.
 - Both start and stop conditions are generated in master mode by software.
- In Slave mode, the interface is capable of recognizing its own addresses (7 or 10-bit), and the General Call address.
 - The General Call address detection may be enabled or disabled by software.
- A 9th clock pulse follows the 8 clock cycles of a byte transfer, during which the receiver must send an acknowledge bit to the transmitter.
- The I2C interface addresses (dual addressing 7-bit/ 10-bit and/or general call address) can be selected by software.

Before writing the code we have to decide in which mode we will be using the I2C, /master mode or Slave mode, and accordingly we can use master or slave, Transmitter and Receiver.

Coding and setting requirements

26 June 2025 11:09

STEPS TO CONFIGURE I2C

1. Enable the I2C clk and GPIO clk.
2. Configure the I2C Pins for Alternate Functions.
 - a) Select Alternate Function in MODER Register.
 - b) Select Open Drain Output
 - c) Select High Speed for the pins.
 - d) Select Pull-up for both the pins.
 - e) Configure the Alternate Function in AFR Register (Refer rm0090, pg 257, 2nd diagram).
3. Reset the I2C.
4. Program the peripheral input clk in I2C_CR2 Register in order to generate correct timings
5. Configure the clock control registers.
6. Configure the rise time register.
7. Program the I2C_CR1 register to enable the peripheral.

CCR calculation :

CCR formula = $(Tr(SCL) + Tw(SCLH)) / T_{pclk1}$

There is no I2C characteristics table in the stm32f429 characteristics, so relying on the other F4 series

But the T_{rise} value for standard mode (from the internet) is 1000ns, $Tw(SCLH)$ as 4000ns

$T_{pclk1} = 1 / \text{clk freq.}$ (so I am giving either 36MHz or 16 MHz or 42 MHz.

By Calculation :

For 16MHZ : CCR= 80

For 36MHZ : CCR= 180

For 45MHZ : CCR= 225

For 42MHZ : CCR = 210

T_rise register :

$T_{rise} = (Tr(SCL) / T_{pclk1}) + 1$

For 16MHZ : $T_{rise} = 17$

For 36MHZ : $T_{rise} = 37$

For 45MHZ : $T_{rise} = 46$

For 42MHZ : $T_{rise} = 43$

STEPS TO START COMMUNICATION:

->Send the start condition

->Wait for SB bit to set. This indicates the condition is set.

STEPS TO WRITE DATA :

->Wait for the TXE bit to set, indicates the DR is empty.

->Send the data to the DR register.

->Wait for the BTF nit to set, this indicates the end of last data transmission

STEPS TO SEND THE SLAVE ADDRESS :

->Send the Slave Address to the DR Register.

->Wait for the ADDR bit to set, This indicates the end of address transmission.

->Clear the ADDR by reading the SR1 and SR2.

STEPS TO WRITE MULTIPLE DATA :

->Wait for the TXE bit to set, this indicates that the CR is empty

-> Keep Sending the data to DR register after performing the check if the Txe bit is et.

-> Once the data transfer is complete, Wait for the VTF to set, this indicates the last data transmission

Slave Address

->You can use 0x42, 0x50, 0x68, etc., as long as it's **between 0x08 and 0x77**, and not conflicting with anything else.

SLAVE RECEIVER :

->The ADDR bit is set when the received address and the address on the I2C's OAR1 matches.

-> So we wait until it is set.

-> Once it sets, this bit can be cleared by simply reading the SR1 and SR2 registers, and it is important to clear the ADDR bit.

-> The RXNE bit sets when the receive buffer gets the data, means the a data byte is received from master.

I2C_CR2.FREQ is a 6 - bit field in the I2C_CR2 register.

→ **I2C1, I2C2, I2C3** are on the **APB1 bus**

→ Look at your **clock tree** to find the frequency of APB1.

→ Check in STM32CubeMX or CubeIDE (Clock Configuration tab)

Main code

01 July 2025 10:51

```
#include "stm32f429xx.h"
#include "header.h"
#define addr 0x42
volatile char r;

void init_config()
{
    i2c1_gpio_config();
    i2c1_config();
    i2c2_gpio();
    i2c2_config();
    sys_config();
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN;

    GPIOG->MODER &= ~(3<<(14*2));

    GPIOG->MODER |= (1<<(14*2));
}
int main()
{
    init_config();
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN;

    GPIOG->MODER &= ~(3<<(13*2));

    GPIOG->MODER |= (1<<(13*2));

    I2C1_start();
    I2C1_write(addr,'A');
    r = slave_receive();

    while (1)
    {
        if (r == 'A')
        {
            GPIOG->ODR |= (1<<13); // Turn ON LED on PG13
        }
        else
        {
            GPIOG->ODR &= ~(1<<13); // Turn OFF LED
        }
    }
}
```


I2C1 (Master)

01 July 2025 10:52

```
#include "stm32f429xx.h"
#include "header.h"
#define addr 0x42
void i2c1_gpio_config()
{
    //Enable the clock for GPIO port B.
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;

    //Enable the clock for I2C peripheral.
    RCC->APB1ENR |= (1<<21);

    /* Select the mode for GPIO */

    //clear the bits for PB6(SCL) and PB7(SDA)
    GPIOB->MODER &= ~(3<<(6*2) | 3<<(7*2));

    //Setting the bits for pin 6 and 7
    GPIOB->MODER |= (2<<(6*2) | 2<<(7*2));

    //Set output type to open drain, PB6 and PB7
    GPIOB->OTYPER |= (1<<6) | (1<<7);

    //Set output speed to high
    GPIOB->OSPEEDR |= (2<<(6*2) | 2<<(7*2));

    //Set the port to pull up
    GPIOB->PUPDR &= ~(1<<(6*2) | 1<<(7*2));
    GPIOB->PUPDR |= (1<<(6*2) | 1<<(7*2));

    //Set GPIO Port B as alternate function register.
    GPIOB->AFR[0] |= (4<<(6*4) | 4<<(7*4));
}

void i2c1_config()
{
    I2C1->CR1=0;
    //reset the I2C
    //when SWRST bit is set, it resets the peripheral
    I2C1->CR1 |= (1<<15);
    I2C1->CR1 &= ~(1<<15);

    //setting up the peripheral clk
    /* Note : From stm32 cube id3 clk configuration table, APB1 runs on
    36MHz
    From keil, system configuration, the system clk is 16MHz, i am confused
    which to use, so i will be writing both.
    From the data sheet, APB1 runs on 45MHz, if we want to use 45MHz,
    we should manually write the clk configuration again.

    */
```

```

//I2C1->CR2 |= (16<<0);
I2C1->CR2 |= (36<<0);
//I2C1->CR2 |= (42<<0);
//I2C1->CR2 |= (45<<0);

//Clk control Register
//I2C1->CCR=80;//16MHz
I2C1->CCR=180;//36MHz
    //I2C1->CCR=210;//42MHz
//I2C1->CCR=225;//45MHz

// I2C1->TRISE = 17;//16MHz
I2C1->TRISE = 37;//36MHz
//I2C1->TRISE = 43;//42MHz
//I2C1->TRISE = 46;//45MHz

//Enable the peripheral using I2C_CR1 register.
I2C1->CR1 |=I2C_CR1_PE;

}
void I2C1_start()
{
    //Set the START bit.
    // Setting the START bit causes the interface to generate a Start condition and
    // to switch to master mode

    I2C1->CR1 |=I2C_CR1_START;//start condition is ntg

    //wait till SB bit is set, indicates the start condition is sent
    while(!(I2C1->SR1 & I2C_SR1_SB));

}
void I2C1_stop()
{
    //Set the STOP bit to stop the communication
    I2C1->CR1 |= I2C_CR1_STOP;
}
void I2C1_write(uint8_t add,char data)
{
    //send the address, giving zero in R/W bit
    I2C1->DR=(add<<1)|(0<<0);

    //Wait for the address bit to be set,
    //ADDR is set means the address matches with that of the slave
    while(!(I2C1->SR1 & I2C_SR1_ADDR));

    //Read the SR1 and SR2 registers to clear the ADDR bit
    uint8_t temp = I2C1->SR2 | I2C1->SR1;
    //(void)I2C1->SR2;

    //The hardware sets the AF bit, if no ACK is received.
    if (I2C1->SR1 & I2C_SR1_AF)
    {
        I2C1->SR1 &= ~I2C_SR1_AF; // Clear NACK flag
        I2C1_stop(); // Stop transmission
    }
}

```

```

        GPIOG->ODR |= (1 << 14);
return; // Exit on error
    }

    //Transmit the data
    //I2C1->DR=data;
    //wait for the txe bit to set
    //If TXE bit is set, that means the DR is empty
    while(!(I2C1->SR1 & I2C_SR1_TXE));

    //Transmit the data
    I2C1->DR=data;

    //wait for the BTF bit to set
    while(!(I2C1->SR1 & I2C_SR1_BTF));
        I2C1_stop();
}

void I2C_writemulti(uint32_t *data,uint8_t size)
{
    //wait for the txe bit to set
    while(!(I2C1->SR1 & I2C_SR1_TXE));
    while(size)
    {
        //wait for the txe bit to set
        while(!(I2C1->SR1 & I2C_SR1_TXE));
        I2C1->DR = (volatile uint32_t) *data++;
        size--;
    }
    while(!(I2C1->SR1 & I2C_SR1_BTF));
}

```

I2C2 (Slave)

01 July 2025 10:56

```
#include "stm32f429xx.h"
#include "header.h"
#define addr 0x42
/*void i2c2_gpio()
{
    //Enable the clock for I2C peripheral.
    RCC->APB1ENR |= RCC_APB1ENR_I2C2EN;

    //Enable the clock for GPIO port F.
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOFEN;

    // Select the mode for GPIO

    //clear the bits for PF0(SDA) and PF1(SCL)
    GPIOF->MODER &= ~( 3<<(0*2) | 3<<(1*2));

    //Setting the bits for pin 0 and 1
    GPIOF->MODER |= (2 << (0*2) | 2 << (1*2));

    //Set output type to open drain, PF0 and PF1
    GPIOF->OTYPER |= (1<<0) | (1<<1);

    //Set output speed to high
    GPIOF->OSPEEDR |= (2 << (0*2) | 2 << (1*2));

    //Set the port to pull up
    GPIOF->PUPDR &= ~((3 << (0 * 2)) | (3 << (1 * 2)));
    GPIOF->PUPDR |= (1 << (0*2) | 1 << (1*2));

    //Set GPIO Port B as alternate function register.
    GPIOF->AFR[0] &= ~( (0xF << (0*4)) | (0xF << (1*4)));
    GPIOF->AFR[0] |= ( 4 << (0*4) | 4 << (1*4));
}
*/
void i2c2_gpio()
{
    //Enable the clock for GPIO port B
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;

    //Enable the clock for I2C peripheral.
    RCC->APB1ENR |= RCC_APB1ENR_I2C2EN;

    // Select the mode for GPIO

    //clear the bits for PB10(SCL) and PB11(SDA)
    GPIOB->MODER &= ~( 3<<(10*2) | 3<<(11*2));
    //Setting the bits as AF(10) for pin 10 and 11
    GPIOB->MODER |= (2 << (10*2) | 2 << (11*2));

    //Set output type to open drain, PB10 and PB11
    GPIOB->OTYPER |= (1<<10) | (1<<11);
```

```

//Set output speed to high
GPIOB->OSPEEDR |= (2 << (10*2) | 2 << (11*2));

//Set the port to pull up
GPIOB->PUPDR &= ~( (3 << (10 * 2)) | (3 << (11 * 2)));
GPIOB->PUPDR |= (1 << (10*2) | 1 << (11*2));

//Set GPIO Port B as alternate function register.
GPIOB->AFR[1] &= ~( (0xF << ((10-8)*4)) | (0xF << ((11-8)*4)));
GPIOB->AFR[1] |= ( 4 << ((10-8)*4) | 4 << ((11-8)*4));
}

void i2c2_config()
{
    I2C2->CR1=0;
    //By default i2c will be operating in slave mode

    //setting the clk
    //I2C2->CR2 = 16;
    I2C2->CR2 = 36;
    //    I2C2->CR2 = 42;
    //I2C2->CR2 = 45;

    //Set own slave address
    I2C2->OAR1 = (addr<<1);

    //Set ADDMODE bit as 0, as we are choosing 7 bit address
    I2C2->OAR1 &= ~(I2C_OAR1_ADDMODE);

    //I2C2->CCR=80;//16MHz
    I2C2->CCR=180;//36MHz
    //I2C2->CCR=210;//42MHz
    //I2C2->CCR=225;//45MHz

    //I2C2->TRISE = 17;//16MHz
    I2C2->TRISE = 37;//36MHz
    //I2C2->TRISE = 43;//42MHz
    //I2C2->TRISE = 46;//45MHz

    //Enable the ACK
    I2C2->CR1 |= I2C_CR1_ACK;

    //Enable the peripheral
    I2C2->CR1 |= I2C_CR1_PE;

}

char slave_receive()
{
    volatile uint16_t temp;
    while(!(I2C2->SR1 & I2C_SR1_ADDR));
    temp=I2C2->SR1 | I2C2->SR2;

    //wait for the RXNE is set
    while(!(I2C2->SR1 & I2C_SR1_RXNE));

    //read the received byte

```

```
char rec = I2C2->DR;

//wait for the STOP bit to set
while(!(I2C2->SR1 & I2C_SR1_STOPF));
tempI2C2->SR1;
I2C2->CR1 |=0;

return rec;
}
```

Sys_Config

01 July 2025 10:57

```
#include "stm32f429xx.h"
#include "header.h"
#define PLL_M 4
#define PLL_N 72
#define PLL_P 0 // PLL_p=2;

void sys_config()
{
    //Enable HSE and wait for the HSE to become ready
    RCC->CR |= RCC_CR_HSEON;
    while(!(RCC->CR & RCC_CR_HSERDY));

    //Enable power interface clk, and voltage regulator
    RCC->AHB1ENR |= RCC_APB1ENR_PWREN;
    PWR->CR |= PWR_CR_VOS;//this value always corresponds to reset value

    //Flash related setup
    //Configure the FLASH PREFETCH and the LATENCY
    //Get the value from cube MX / cube ide
    /* In the Pinout & Configuration
       Under RCC, In parameter settings
       In system Parameters, we can find the flash related bits
    */

    FLASH->ACR |= FLASH_ACR_ICEN | FLASH_ACR_PRFTEN | FLASH_ACR_DCEN |
    FLASH_ACR_LATENCY_5WS;

    //Configure Prescalers for peripheral and main clocks
    //refer from cube ide
    //All these bits are in RCC CFGR
    //AHB Prescaler
    RCC->CFGR |= RCC_CFGR_HPRE_DIV1;
    //APB1 Prescaler
    RCC->CFGR |= RCC_CFGR_PPRE1_DIV2;
    //APB2 Prescaler
    RCC->CFGR |= RCC_CFGR_PPRE2_DIV1;

    //configure the main pll
    RCC->PLLCFGR = (PLL_M << 0) | (PLL_N << 6) | (PLL_P << 16) | RCC_PLLCFGR_PLLSRC_HSE;
    //Select source as hse oscillator

    //Enable the PLL
    RCC->CR |=RCC_CR_PLLON;

    //wait for it to be ready
    while(!(RCC->CR & RCC_CR_PLLRDY));

    //set the clk source
    RCC->CFGR |=RCC_CFGR_SW_PLL;
    //waiting until the clk source is set to pll
```

```
while((RCC->CFGR & RCC_CFGR_SWS)!=RCC_CFGR_SWS_PLL);  
}
```


Header file

01 July 2025 10:59

```
#include "stm32f429xx.h"
#define header_h

void i2c1_gpio_config();
void i2c2_gpio();
void i2c1_config();
void i2c2_config();
void I2C1_start();
void I2C1_write(uint8_t addr,char data);
void I2C1_stop();
void I2C_writemulti(uint32_t *data,uint8_t size);
char slave_receive();
void sys_config();
```