

Chapter 6 Inter Integrated Circuit Communication

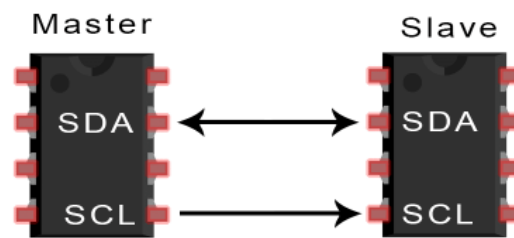
6.1 Introduction:

The Inter-Integrated Circuit (I²C) Protocol is a protocol intended to allow multiple "peripheral" digital integrated circuits ("chips") to communicate with one or more "controller" chips.

6.1.1 Why Use I2C?

- I2C combines the best features of SPI and UARTs.
- With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves.
- This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.
- Most I²C devices can communicate at **100kHz (Standard mode)** or **400kHz (Fast mode)** and up to **1 Mbit/s (Fast-mode Plus)**, or up to **3.4 Mbit/s (High-speed mode)**.

Like UART communication, I2C only uses **two wires** to transmit data between devices:



SDA (Serial Data) – The line for the master and slave to send and receive data.

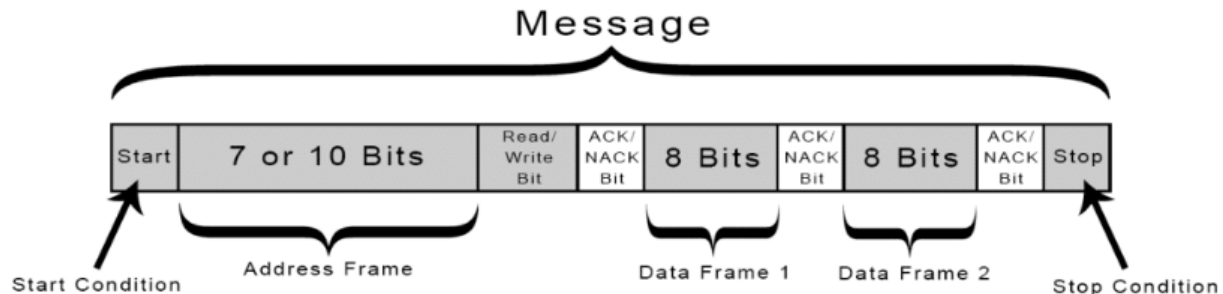
SCL (Serial Clock) – The line that carries the clock signal.

I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line). Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The **clock signal** is always controlled by the **master**.

Wires Used	2
Maximum Speed	Standard mode= 100 kbps
	Fast mode= 400 kbps
	High speed mode= 3.4 Mbps
	Ultra fast mode= 5 Mbps
Synchronous or Asynchronous?	Synchronous
Serial or Parallel?	Serial
Max # of Masters	Unlimited
Max # of Slaves	1008

6.1.2 How I2C Works?

With I2C, data is transferred in *messages*. Messages are broken up into *frames* of data. Each message has an address frame that contains the binary address of the **slave**, and one or more **data frames** that contain the data being transmitted. The message also includes **start and stop conditions**, read/write bits, and ACK/NACK bits between each data frame:



Start Condition: The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.

Stop Condition: The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

Address Frame: A 7- or 10-bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

Read/Write Bit: A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

ACK/NACK Bit: Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

Addressing:

I2C **doesn't have slave select lines** like SPI, so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by *addressing*. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address. If the address matches, it sends a low voltage ACK bit back to the master. If the address doesn't match, the slave does nothing and the SDA line remains high.

Read/write bit

The address frame includes a single bit at the end that informs the slave whether the master wants to write data to it or receive data from it. If the master wants to send data to the slave, the read/write bit is a low voltage level. If the master is requesting data from the slave, the bit is a high voltage level.

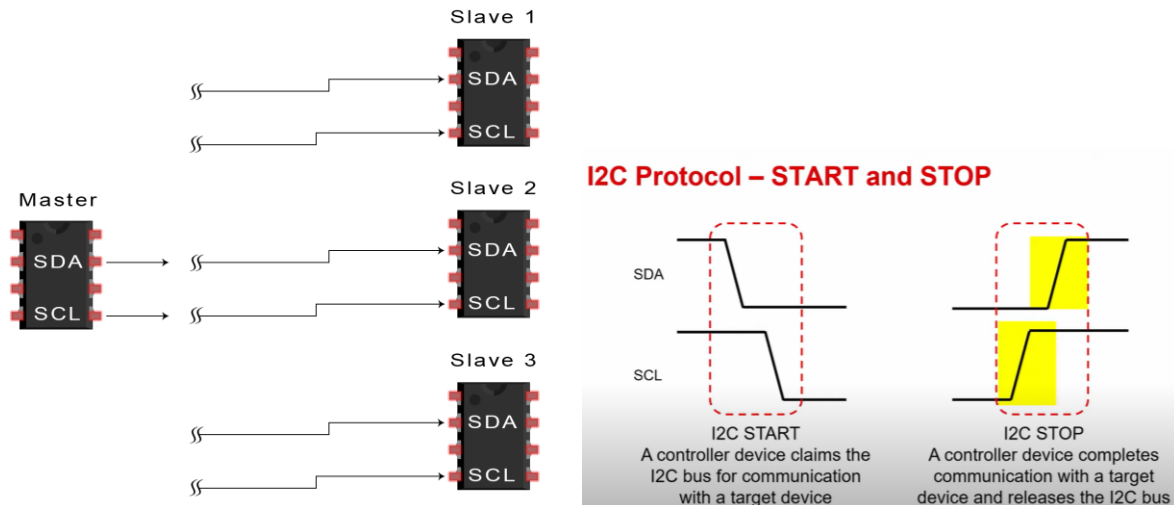
The Data frame

After the master detects the ACK bit from the slave, the first data frame is ready to be sent. The data frame is always 8 bits long, and sent with the most significant bit first. Each data frame is immediately followed by an ACK/NACK bit to verify that the frame has been received successfully.

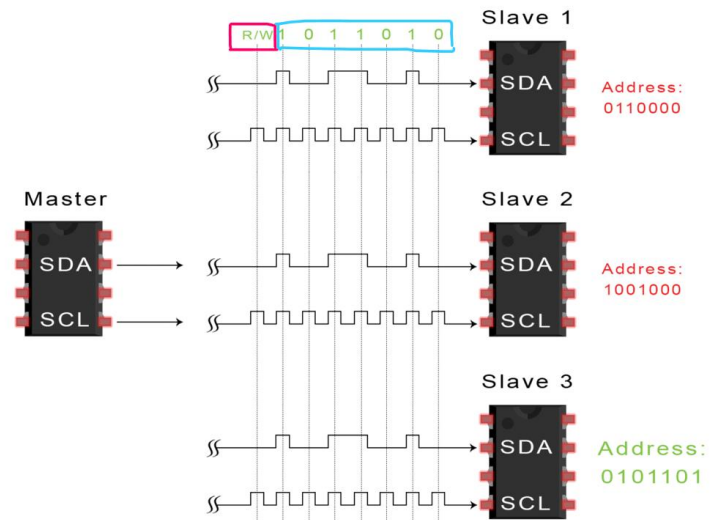
The ACK bit must be received by either the master or the slave (depending on who is sending the data) before the next data frame can be sent. After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

6.1.2 Steps of I2C Data Transmission:

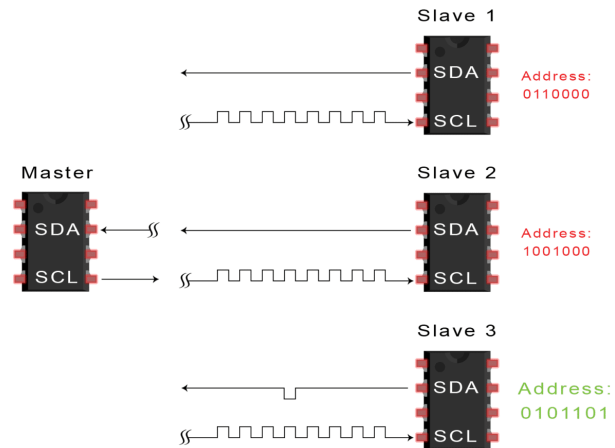
1. The master sends the **start condition** to every connected slave by switching the **SDA line from a high voltage level to a low voltage level** before switching the **SCL line from high to low**.



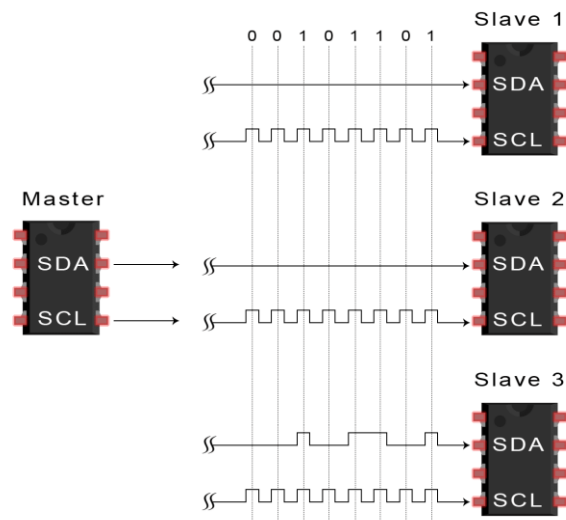
2. The master sends each slave the **7- or 10-bit address of the slave** it wants to communicate with, along with the **read/write bit**. (Usually the Read bit is “1” and Write Bit is “0”.)



3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an **ACK bit by pulling the SDA line low for one bit**. If the address from the master does not match the slave's own address, the slave leaves the SDA line high.



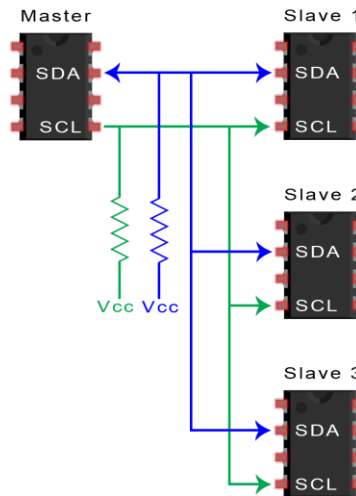
4. The master sends or receives the data frame:



5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:
6. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high:

6.2.3 Single Master with Multiple Slaves:

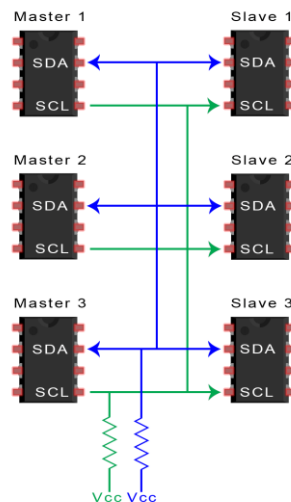
Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7-bit address, 128 (27) unique address are available. Using 10-bit addresses is uncommon, but provides 1,024 (210) unique addresses. To connect multiple slaves to a single master, wire them like this, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



6.2.4 Multiple Masters with Multiple Slaves:

Multiple masters can be connected to a single slave or multiple slaves. **The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line.** To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message.

If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it's safe to transmit the message. To connect multiple masters to multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:

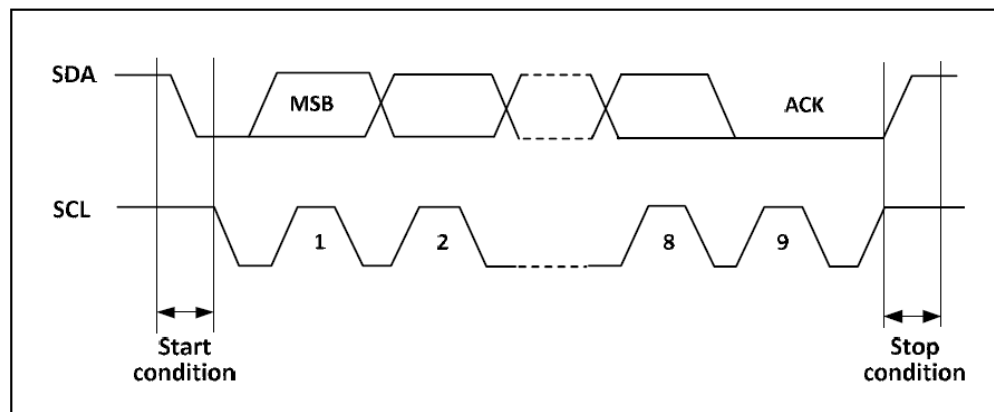


6.2 I2C with CH32v003:

6.2.1 Main Features

- Support **master** and **slave** modes
- Support **7-bit** or **10-bit** addresses
- Slave devices support **dual 7-bit addresses**
- Support two speed modes: **100KHz** and **400KHz**
- Multiple status modes, multiple error flags
- Support extended clock function
- 2 interrupt vectors
- DMA support (Direct Memory Access)
- Support PEC (Packet Error Checking or Packet Error Code)
- SMBus compatible

I2C is a half-duplex bus that can only operate in one of the following four modes at the same time: master device transmit mode, master device receive mode, slave device transmit mode and slave device receive mode. The I2C module works in slave mode by default and automatically switches to master mode when a start condition is generated and to slave mode when arbitration is lost or a stop signal is generated. the I2C module supports multi-master functionality. When working in master mode, the I2C module actively emits data and addresses. Both data and address are transmitted in 8-bit units, with the high bit before and the low bit after. After the start event is a one-byte (in 7-bit address mode) or two-byte (in 10-bit address mode) address, and for every 8-bit data or address sent by the host, the slave needs to reply with an answer ACK, which pulls the SDA bus low, as shown in Figure 13-1.



In order to work properly the I2C must be fed with the correct clock, which is a minimum of 2MHz in standard mode and 4MHz in fast mode.

6.3 I2C Example code:

In this example code we connect two CH32 modules and perform I2C communication when they are powered on at the same time.

```
7.  /***** (C) COPYRIGHT
    *****/
8.  * File Name      : main.c
9.  * Author         : WCH
10. * Version        : V1.0.0
11. * Date           : 2023/12/22
12. * Description    : Main program body.
13.
    *****/
14. * Copyright (c) 2021 Nanjing Qinheng Microelectronics Co., Ltd.
15. * Attention: This software (modified or not) and binary are used
    for
16. * microcontroller manufactured by Nanjing Qinheng
    Microelectronics.
17.
    *****/
18.
19. /*
20.  *@Note
21.  *7-bit addressing mode, master/slave mode, transceiver routine:
22.  *I2C1_SCL(PC2)\I2C1_SDA(PC1).
23.  *This routine demonstrates that Master sends and Slave receives.
24.  *Note: The two boards download the Master and Slave programs
    respectively,
25.  * and power on at the same time.
26.  * Hardware connection:
27.  * PC2 -- PC2
28.  * PC1 -- PC1
29.  *
30.  */
31.
32. #include "debug.h"
33.
34. /* I2C Mode Definition */
35. #define HOST_MODE 0
36. #define SLAVE_MODE 1
37.
38. /* I2C Communication Mode Selection */
39. #define I2C_MODE HOST_MODE
40. //#define I2C_MODE SLAVE_MODE
41.
42. /* Global define */
43. #define Size 6
44. #define RXAdderss 0x02
45. #define TxAdderss 0x02
46.
47. /* Global Variable */
48. u8 TxData[Size] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };
49. u8 RxData[5][Size];
50.
51. /*****
    *****
```

```

52.      * @fn      IIC_Init
53.      *
54.      * @brief    Initializes the IIC peripheral.
55.      *
56.      * @return    none
57.      */
58.      void IIC_Init(u32 bound, u16 address)
59.      {
60.          GPIO_InitTypeDef GPIO_InitStructure={0};
61.          I2C_InitTypeDef I2C_InitTSturcture={0};
62.
63.          RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOC |
RCC_APB2Periph_AFIO, ENABLE );
64.          RCC_APB1PeriphClockCmd( RCC_APB1Periph_I2C1, ENABLE );
65.
66.          GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
67.          GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
68.          GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
69.          GPIO_Init( GPIOC, &GPIO_InitStructure );
70.
71.          GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
72.          GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
73.          GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
74.          GPIO_Init( GPIOC, &GPIO_InitStructure );
75.
76.          I2C_InitTSturcture.I2C_ClockSpeed = bound;
77.          I2C_InitTSturcture.I2C_Mode = I2C_Mode_I2C;
78.          I2C_InitTSturcture.I2C_DutyCycle = I2C_DutyCycle_16_9;
79.          I2C_InitTSturcture.I2C_OwnAddress1 = address;
80.          I2C_InitTSturcture.I2C_Ack = I2C_Ack_Enable;
81.          I2C_InitTSturcture.I2C_AcknowledgedAddress =
I2C_AcknowledgedAddress_7bit;
82.          I2C_Init( I2C1, &I2C_InitTSturcture );
83.
84.          I2C_Cmd( I2C1, ENABLE );
85.
86.      }
87.
88.      /*****
89.      * @fn      main
90.      *
91.      * @brief    Main program.
92.      *
93.      * @return    none
94.      */
95.      int main(void)
96.      {
97.          u8 i = 0;
98.          u8 j = 0;
99.          u8 p = 0;
100.         SystemCoreClockUpdate();
101.         Delay_Init();
102.
103.         USART_Printf_Init(460800);
104.
105.         printf("SystemClk:%d\r\n",SystemCoreClock);
106.         printf( "ChipID:%08x\r\n", DBGMCU_GetCHIPID() );

```



```

107.
108.     #if (I2C_MODE == HOST_MODE)
109.         printf("IIC Host mode\r\n");
110.         IIC_Init( 80000, TxAdderss);
111.
112.         for( j =0; j < 5; j++)
113.         {
114.             while( I2C_GetFlagStatus( I2C1, I2C_FLAG_BUSY ) != RESET );
115.
116.             I2C_GenerateSTART( I2C1, ENABLE );
117.
118.             while( !I2C_CheckEvent( I2C1, I2C_EVENT_MASTER_MODE_SELECT )
119. );
120.             I2C_Send7bitAddress( I2C1, 0x02, I2C_Direction_Transmitter );
121.             while( !I2C_CheckEvent( I2C1,
122. I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED ) );
123.             for( i=0; i< 6;i++ )
124.             {
125.                 if( I2C_GetFlagStatus( I2C1, I2C_FLAG_TXE ) != RESET )
126.                 {
127.                     Delay_Ms(100);
128.                     I2C_SendData( I2C1, TxData[i] );
129.                 }
130.             }
131.
132.             while( !I2C_CheckEvent( I2C1,
133. I2C_EVENT_MASTER_BYTE_TRANSMITTED ) );
134.             I2C_GenerateSTOP( I2C1, ENABLE );
135.             Delay_Ms(1000);
136.         }
137.     #elif (I2C_MODE == SLAVE_MODE)
138.         printf("IIC Slave mode\r\n");
139.         IIC_Init( 80000, RXAdderss);
140.
141.         for( p =0; p < 5; p++)
142.         {
143.
144.             i = 0;
145.             while( !I2C_CheckEvent( I2C1,
146. I2C_EVENT_SLAVE_RECEIVER_ADDRESS_MATCHED ) );
147.             while( i < 6 )
148.             {
149.                 if( I2C_GetFlagStatus( I2C1, I2C_FLAG_RXNE ) != RESET )
150.                 {
151.                     RxData[p][i] = I2C_ReceiveData( I2C1 );
152.                     i++;
153.                 }
154.             }
155.             I2C1->CTLR1 &= I2C1->CTLR1;
156.             printf( "RxData:\r\n" );
157.             for(p=0; p<5; p++)
158.             {
159.                 for( i = 0; i < 6; i++ )
160.                 {

```

```
161.         printf( "%02x ", RxData[p][i] );
162.     }
163.     printf( "\r\n " );
164. }
165.
166.
167. #endif
168.
169.     while(1);
170. }
```

6.4 Case Study for I2C Communication with Sensor:
Go to the File name as I2C Case Study in the same folder.