Application Note

Application Note

Document No.: AN1102

APM32F103 Software I2C Reading and Writing EEPROM

Version: V1.0

1 Introduction

This application note provides how to use the ordinary IO port of the APM32F103 series to simulate I2C timing and achieve bidirectional communication with AT24C02.

APM32F103 has two built-in I2C bus interfaces, which can work in multi-master mode or slave mode and support both standard and fast modes. The I2C interface supports 7-bit or 10-bit addressing, and supports slave address addressing in 7-bit slave mode. It has built-in hardware CRC generator and calibrator. DMA operation can be used and SMBus bus version 2.0/PMBus bus is supported.

Contents

2 I2C Introduction

I2C is a short-distance communication protocol. In physical implementation, I2C bus is composed of two signal lines (SDA and SCL) and one ground wire. These two signal lines can be used for bidirectional transmission.

•Two signal lines, SCL clock line and SDA data line. SCL provides timing for SDA, and SDA transmits/receives data in series.

•Both SCL and SDA signal lines are bidirectional.

•The two systems share a ground wire when they use I2C bus for communication.

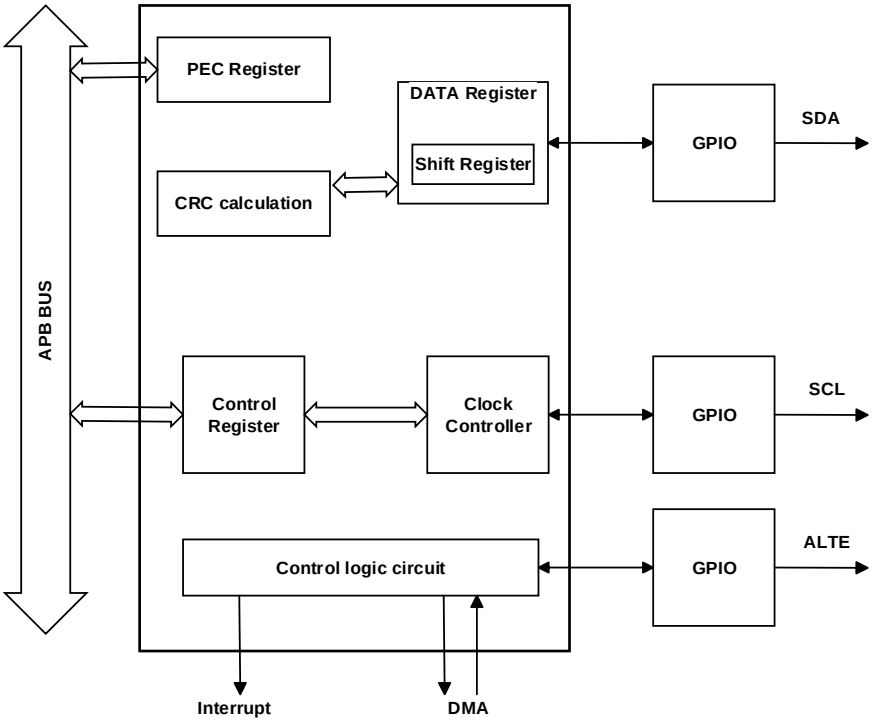


Figure 1 I2C Function Structure Diagram

2.1 Overview of I2C protocol

Data is transmitted in the form of frame, and each frame is composed of 1 byte (8 bits).

In the rising edge phase of SCL, SDA needs to keep stable and SDA changes when SCL is low.

In addition to data frame, I2C bus also has start signal, stop signal and acknowledgment signal.

- Start bit: During the stable high level period of SCL, a falling edge of SDA starts

transmission.

- Stop bit: During the stable high level period of SCL, a rising edge of SDA stops transmission.
- Acknowledge bit: Used to indicate successful transmission of one byte.

After the bus transmitter (regardless of master or slave) transmits 8-bit data, SDA will release (from output to input). During the ninth clock pulse period, the receiver will pull down SDA to acknowledge receiving of data.

I2C communication reading and writing process

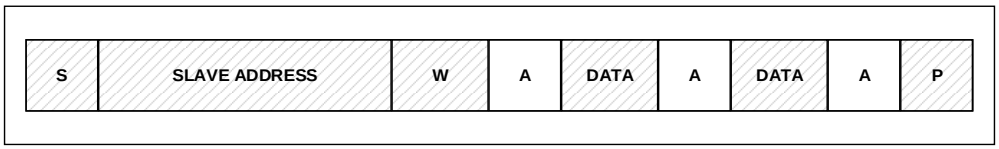


Figure 2 Master Writes Data to Slave

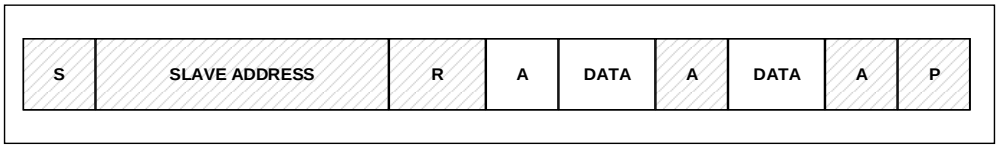


Figure 3 Master Reads Data from Slave

Note: (I2C sends an 8-bit command, and the last bit represents the read/write bit)

(1) Shaded area: This data is transmitted from master to slave.

(2) S: Start signal.

(3) SLAVE ADDRESS: Slave address.

(4) Unshaded area: This data is transmitted from slave to master.

(5) R/W: Selection bit of transmission direction.

(6) 1: Read.

(7) 0: Write.

(8) P: Stop signal.

After the start signal is generated, all slaves will wait for the slave address signal transmitted by the master. In I2C bus, the address of each device is unique. When the address signal matches the device address, the slave will be selected, and the unselected slave will ignore the future data signal.

When the master direction is writing data

After broadcasting the address and receiving the acknowledgment signal, the master will transmit data to the slave, the data length is one byte, and every time the master transmits one byte of data, it needs to wait for the acknowledgment signal transmitted by the slave. After all the bytes have been transmitted, the master will transmit a stop signal (STOP) to the slave, indicating that the transmission is completed.

When the master direction is reading data

After broadcasting the address and receiving the acknowledgment signal, the slave will transmit the data to the master. The size of the data packet is 8 bits. Every time the slave transmits one byte of data, it needs to wait for the acknowledgment signal from the master. When the master wants to stop receiving data, it needs to return a non-acknowledgment signal to the slave, then the slave will stop transmitting the data automatically.

2.2 Data validity

In the process of data transmission, the data on SDA line must be stable when the clock signal SCL is at high level. Only when the SCL is at the low level, can the level state of SDA be changed, and a clock pulse is needed for the bit transmission of each data.

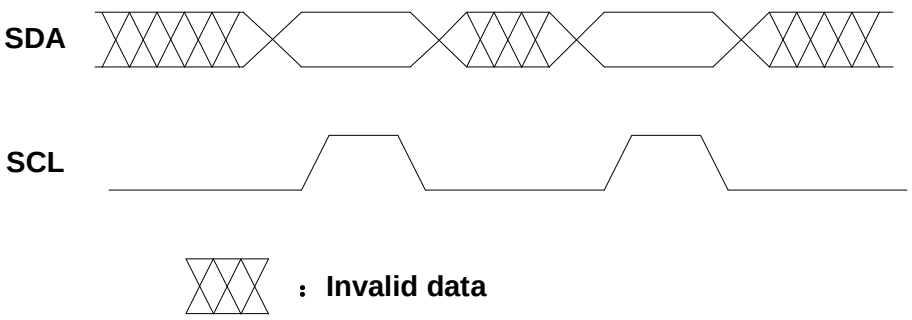


Figure 4 SDA Timing Diagram

2.3 I2C timing

I2C has three signals during data transmission, which are start signal, stop signal, and acknowledgment signal.

2.3.1 I2C start signal (START)

The SATRT signal is defined as: when the clock line SCL remains high, the data line SDA level is pulled down, representing the start of a data transmission. The start signal is a timing signal for level jump, rather than a level signal. This signal is issued by the master device, and the I2C bus must be in an idle state before this signal is established.

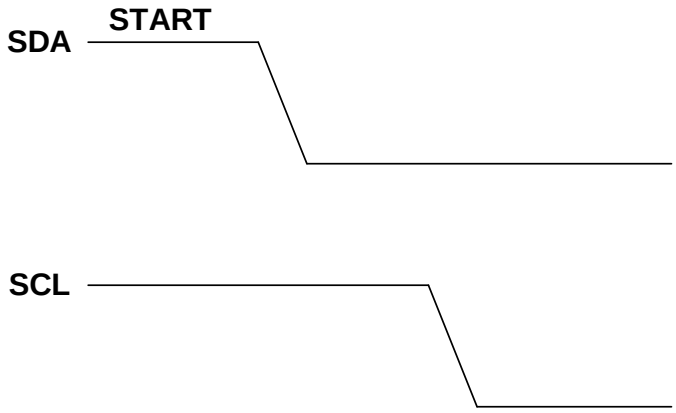


Figure 5 START Signal Timing Diagram

2.3.2 I2C stop signal (STOP)

The STOP signal is defined as: when the clock line SCL remains high, the data line SDA level is released, representing the end of a data transmission. The stop signal is a timing signal for level jump, rather than a level signal. This signal is issued by the master device, and the I2C returns to an idle state after this signal is established.

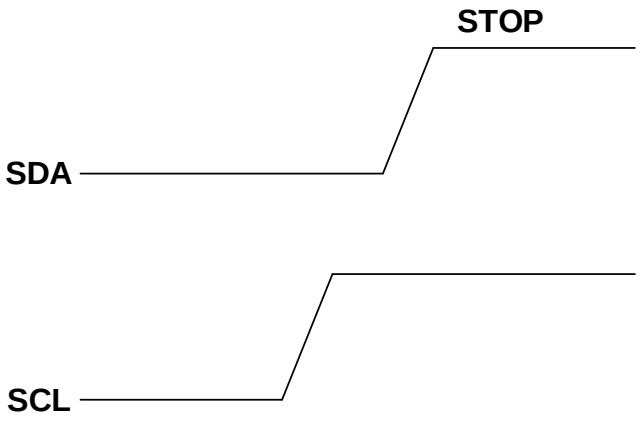


Figure 6 STOP Signal Timing Diagram

2.3.3 I2C acknowledgment signal (ACK)

All data on the I2C bus is transmitted in 8-bit byte. During I2C communication, every time the transmitter transmits a data, the receiver will feed back an acknowledgment signal.

When the acknowledgment signal is low level, it is specified as a valid acknowledgment (ACK); when the acknowledgment signal is high level, it is specified as a non-valid acknowledgment (NACK). Requirement for feedback of valid acknowledgment signal: Before receiving the 9th pulse, the receiver will pull down the SDA line to ensure that the SCK line outputs a stable low level when the SCL line is at high level.

If the receiver is a master device, when it receives the last byte from the slave device, it will transmit a NACK signal to notify the slave device to stop transmitting data and release the SDA line, so that the master device can transmit a stop signal.

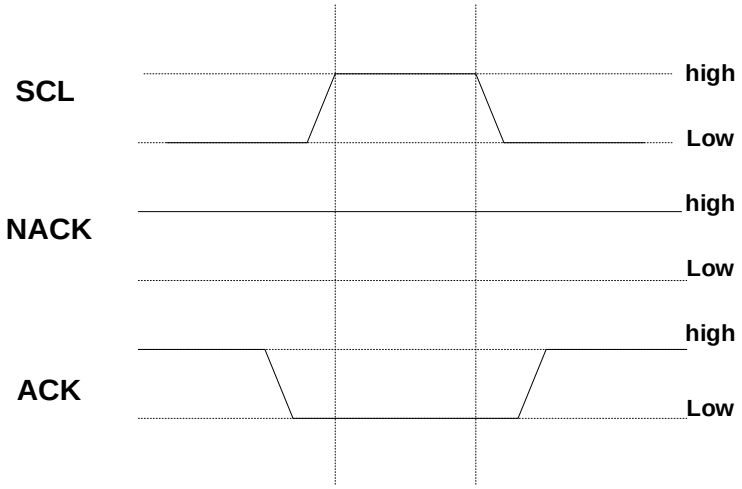


Figure 7 ACK and NACK Signal Timing Diagram

3 Introduction to AT24C02

AT24C02 is a serial EEPROM (Electrically Erasable Programmable Read-Only Memory) chip produced by Atmel (now Microchip Technology) Company. It is a member of the AT24C series chips, with a storage capacity of 2K bits (256 bytes).

The following are the performance characteristics of AT24C02.

(1) The I2C (Inter Integrated Circuit) bus interface is adopted. It is a commonly used serial communication protocol and can transmit data among multiple devices. It supports I2C communication rates in both standard mode (100 KHz) and fast mode (400 KHz).

(2) It has erasable and programmable functions, and can read and write data through power supply. It adopts 8-bit address addressing and can connect multiple AT24C02 chips, so that the system can be expanded to a larger storage capacity.

(3) Data storage is non-volatile and the data can be maintained even in the event of power outage. It also has internal write protection function, and can protect stored data through hardware or software to prevent accidental writing or erasing.

(4) It is widely used in various electronic devices, such as storing configuration information, calibration data, serial numbers, and log records. Due to the moderate capacity, simple interface, and low power, it has been widely used in embedded systems and small electronic devices.

In general, the AT24C02 chip features small storage capacity, simple interface, good reliability, and low power, so it is suitable for the data storage requirements of many embedded systems and small electronic devices.

The following is the communication timing diagram of AT24C02.

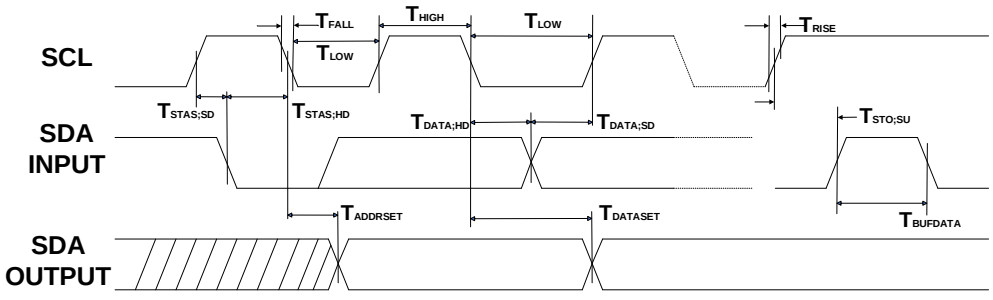


Figure 8 AT24C02 Communication Timing Diagram

4 Hardware design

This design uses the PB6 and PB7 pins of APM32F103ZE for GPIO simulating I2C, and the SCL and SDA of AT24C02 are connected to PB6 and PB7 of APM32F103ZE, respectively. The connection relationship is shown in the figure.

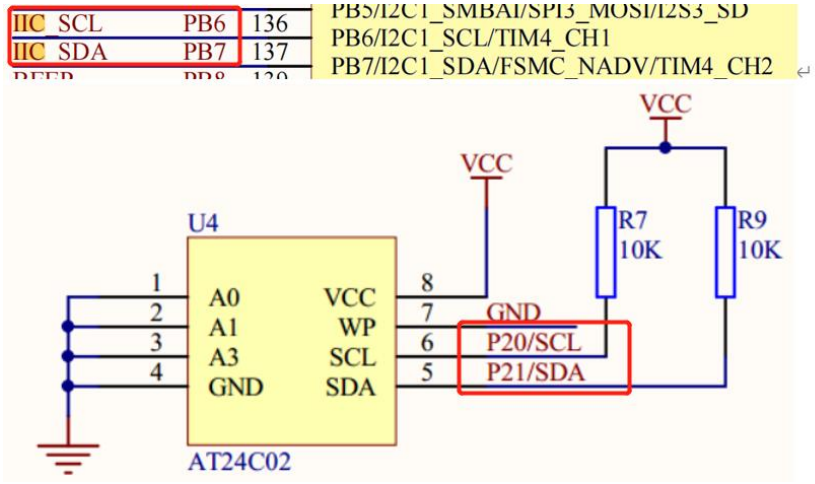


Figure 9 Diagram of Connection between APM32F103ZE and AT24C02

5 Software design

This chapter mainly introduces the code implemented by GPIO simulating I2C, and the APM32F103ZE using software I2C to communicate with AT24C02. The workflow diagram is shown below.

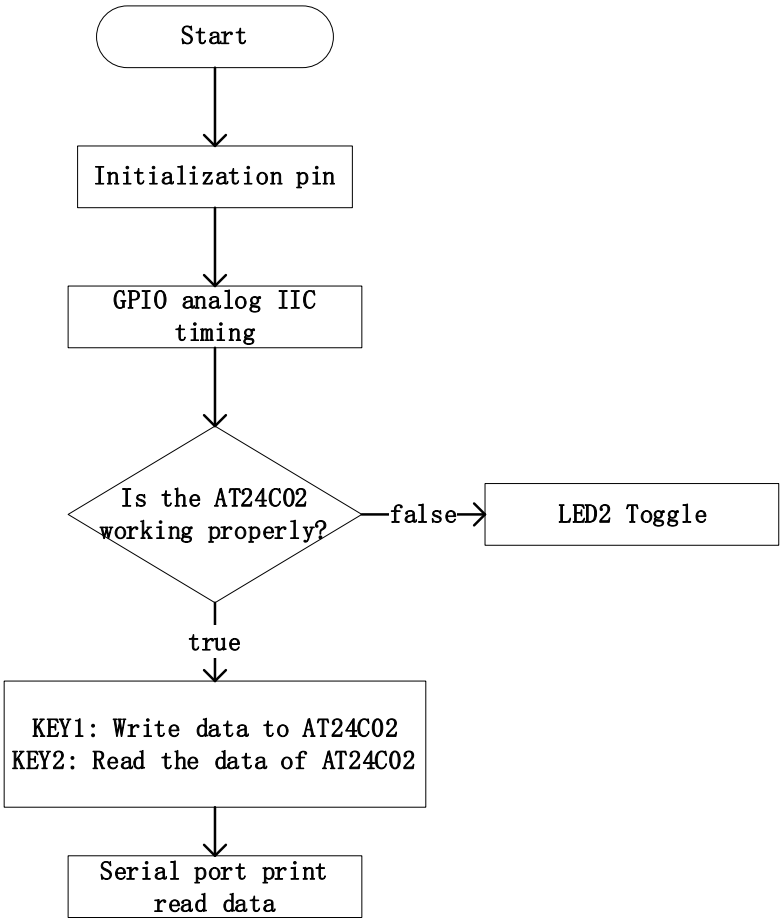


Figure 10 System Workflow Diagram

5.1 GPIO Simulating I2C Configuration

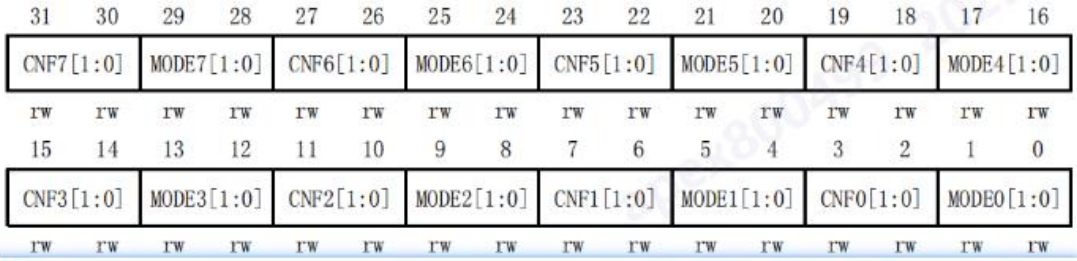
5.1.1 SDA_IN_OUT

//SDA is set to input mode

```
void SDA_IN(void)
{
    GPIOB->CFGLOW &= 0X0FFFFFFF;
    GPIOB->CFGLOW |= (uint32_t)8<<28;
```

```
}

//SDA is set to output mode

void SDA_OUT(void)

{

      GPIOB->CFGLOW &= 0X0FFFFFFF;

     GPIOB->CFGLOW |= (uint32_t)3<<28;

}
```

Please refer to the following APM32F103 User Manual for the description of CFGLOW register, SDA_IN() function configures the high 4 bits of CFGLOW register in GPIOB to “1000”, and sets the corresponding PB7 pin to input mode; SDA_OUT()function configures the high 4 bits of CFGLOW register in GPIOB to “0011”, and sets the corresponding PB7 pin to push-pull output mode.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

11.6.1 Low 8-bit port configuration register (GPIOx_CFGLOW) (x=A..E)

Offset address: 0x00

Reset value: 0x4444 4444

Field	Name	R/W	Description
29:28 25:24 21:20 17:16 13:12 9:8 5:4 1:0	MODEy[1:0]	R/W	Port x Pin y Mode Configure (y=0…7) 00: Input mode (state after reset) 01: Output mode, the maximum output speed is 10MNz 10: Output mode, the maximum output speed is 2MNz 11: Output mode, the maximum output speed is 50MNz See the data manual for the definition of maximum output speed.
31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	CFGy[1:0]	R/W	Port x Pin y Function Configure (y=0…7) In input (MODE[1:0]=00) mode: 00: Analog input mode 01: Floating input mode (state after reset) 10: Pull-up/Pull-down input mode 11: Reserved In output mode (MODE[1:0]>00): 00: General push-pull output mode 01: General open-drain output mode 10: Push-pull output mode of multiplexing function 11: Open-drain output mode of multiplexing function

Figure 11 CFGLOW Register Description

5.1.2 I2C_Init

```
//PB6 (SCL), PB7 (SDA) initialization
void I2C_Init(void)
{
    GPIO_Config_T gpioConfig;
    RCM_EnableAPB2PeriphClock(RCM_APB2_PERIPH_GPIOB);
    gpioConfig.pin = I2C_PIN_SCL|I2C_PIN_SDA;
    gpioConfig.mode = GPIO_MODE_OUT_PP; //push-pull output
    gpioConfig.speed = GPIO_SPEED_50MHz;
    GPIO_Config(GPIOB, &gpioConfig);
    GPIO_SetBit(GPIOB,I2C_PIN_SCL|I2C_PIN_SDA);
}
```

I2C_Init() function configures the GPIOB clock and configures PB6 (SCL) and PB7 (SDA) to push-pull output mode. As the I2C timing is high level at the beginning, PB6 and PB7 are set to high level by default.

5.1.3 I2C_Start

```
//I2C start timing
void I2C_Start(void)
{
    SDA_OUT(); //SDA output
    I2C_SDA_SET;
    I2C_SCL_SET;
    Delay_us(4);
    I2C_SDA_RESET;          //START: When CLK is at high level, DATA changes from high level to low level
```

```
    Delay_us(4);
    I2C_SCL_RESET; // Clamp the I2C bus and prepare to transmit or receive data
}
```

According to the I2C Start Timing (START) diagram, when the SCL line is at a stable high level, pulling down the SDA line represents the start of the I2C timing, so first perform I2C_SDA_RESET, and after delay for a period of time, perform I2C_SCL_RESET to simulate START timing of I2C.

5.1.4 I2C_Stop

```
//I2C stop timing
void I2C_Stop(void)
{
    SDA_OUT();                  //SDA line output
    I2C_SCL_RESET;
    I2C_ SDA_RESET;          //STOP: When CLK is at a high level, DATA changes from low level to high level
    Delay_us(4);
    I2C_SCL_SET;
    I2C_SDA_SET; // transmit I2C bus stop signal
    Delay_us(4);
}
```

According to the I2C Stop Timing (STOP) Diagram, when the SCL line is at a stable high level, pulling up the SDA line represents the stop of the I2C timing, so first perform I2C_SCL_SET, and then perform I2C_SDA_SET to simulate STOP timing of I2C.

5.1.5 I2C_Wait_Ack

```
//Wait for ACK acknowledgment
uint8_t I2C_Wait_Ack(void)
{
    uint8_t ucErrTime = 0;
```

```
    SDA_IN();                   //SDA is set to input
    I2C_SDA_SET;
    Delay_us(1);
    I2C_SCL_SET;
    Delay_us(1);
    while(I2C_READ_SDA)
    {
        ucErrTime++;
        if(ucErrTime > 250)
        {
            I2C_Stop();
            return 1;
        }
    }
    I2C_SCL_RESET;              //Clock outputs 0
    return 0;
}
```

In the I2C_Wait_Ack() function, set SDA to input mode to read the SDA level. Pull up SCL and SDA. If the SDA line recognizes a low level, it indicates that it has received ACK and returns 0; if ACK is not received after 250 times, it will return 1 and the I2C transmission will be interrupted.

In I2C_Wait_Ack() function, after it has successfully received ACK, it will execute the code of I2C_SCL_RESET, and SCL will be pulled down for a period of time, and this period of time is called clock pull-down period. The function of pulling down SCL is:

（1） Data stability: Pulling down the SCK line can ensure stability of data on the data line (SDA) and avoid data changes during clock switching.

（2） Adaption to the speed of slave device: The slave device may require more time to process the received data due to the slow processing speed. By pulling down the SCK line, the master device can wait for the slave device to complete data processing, so that the speed of data transmission can adapt to the processing speed of the slave device.

（3） Clock synchronization: The clock pull-down period can ensure clock synchronization

between the master and slave devices, to ensure correct timing of data transmission.

In general, the purpose of pulling down the SCK line for a period of time is to stabilize data transmission, adapt to the speed of the slave device, and ensure clock synchronization. This can ensure accurate transmission and correct processing of data in I2C communication.

5.1.6 I2C_Ack

```
//GPIO simulates generating ACK timing
void I2C_Ack(void)
{
    I2C_SCL_RESET;
    SDA_OUT();
    I2C_SDA_RESET;
    Delay_us(2);
    I2C_SCL_SET;
    Delay_us(2);
    I2C_SCL_RESET;
}
```

According to the I2CAcknowledgment Timing (ACK) Diagram, in case of pulse change in the SCL, the SDA will remain at a low level, to ensure that when the SCL is at a stable high level, the SDA remains at a low level. Therefore, first perform I2C_SDA_RESET, then the SCL will generate a pulse, and the SCL will jump from “0” to “1”, and then jump from “1” to “0” to simulate the ACK signal of I2C timing.

5.1.7 I2C_NAck

```
//GPIO simulates generating NACK timing
void I2C_NAck(void)
{
    I2C_SCL_RESET;
    SDA_OUT();
    I2C_SDA_SET;
    Delay_us(2);
```

```
    I2C_SCL_SET;
    Delay_us(2);
    I2C_SCL_RESET;
}
```

According to the I2C Non-acknowledgment Timing (NACK) Diagram, in case of pulse change in the SCL, the SDA will remain at a high level, to ensure that when the SCL is at a stable high level, the SDA remains at a high level. Therefore, first perform I2C_SDA_RESET, then the SCL will generate a pulse, and the SCL will jump from “0” to “1”, and then jump from “1” to “0” to simulate the NACK signal of I2C timing.

5.1.8 I2C_Send_Byte

```
//GPIO simulate I2C: Transmit a byte
void I2C_Send_Byte(uint8_t txd)
{
    uint8_t t;
    SDA_OUT();
I2C_SCL_RESET; // Pull down the clock to start data transmission
    for (t = 0; t < 8; t++)
    {
        I2C_SDA_WRITE((txd&0x80)>>7);
        txd <<= 1;
        Delay_us(2);
        I2C_SCL_SET;
        Delay_us(2);
        I2C_SCL_RESET;
        Delay_us(2);
    }
}
```

As the SDA line can jump only when the SCL is at a low level, first perform I2C_SCL_RESET, then call I2C_SDA_WRITE() function to write each bit to the address of the corresponding

controlled device, and after the one bit of data is written, the SCL will be pulled up to ensure the stability of the data transmission. Then perform I2C_SCL_RESET to prepare for the next data transmission.

5.1.9 I2C_Read_Byte

```
//GPIO simulate I2C: Read a byte
uint8_t I2C_Read_Byte(unsigned char ack)
{
    unsigned char i,receive=0;
    SDA_IN();                    //SDA is set to input
    for (i = 0; i < 8; i++ )
    {
        I2C_SCL_RESET;
        Delay_us(2);
        I2C_SCL_SET;
        receive <<= 1;
        if(I2C_READ_SDA)
        {
            receive++;
        }
        Delay_us(1);
    }
    if (!ack)
    {
        I2C_NAck();                //Transmit NACK
    }
    else
    {
        I2C_Ack();                 //Transmit ACK
```

```
    }
    return receive;
}
```

In I2C_Read_Byte() function, SDA is set to input mode to read the IO port. Subsequently, a rising edge is given to SCL to trigger data sampling, and a "receive" variable is defined in the function. If I2C_READ_SDA is 0, it means that the read data is "0", and then receive=0; On the contrary, receive=1. The 'receive' variable is used to store the read data. The function of "ack" variable is to indicate whether the controlled device needs to transmit an ACK signal. If "ack=1", it means that the data of the control device has not been transmitted completely and the controlled device needs to return ACK; if "ack=0", it means that the data of the control device has been transmitted, ACK is not required, and the controlled device needs to return NACK.

5.1.10 Macro definition

The relevant macro definitions used in the above code are as follows.

```
//IO operating function
#define I2C_PIN_SCL GPIO_PIN_6
#define I2C_PIN_SDA GPIO_PIN_7
#define I2C_SCL_SET           GPIO_SetBit(GPIOB,I2C_PIN_SCL)
#define I2C_SCL_RESET       GPIO_ResetBit(GPIOB,I2C_PIN_SCL)
#define I2C_SDA_SET          GPIO_SetBit(GPIOB,I2C_PIN_SDA)
#define I2C_SDA_RESET       GPIO_ResetBit(GPIOB,I2C_PIN_SDA)
#define I2C_SDA_WRITE(n)    GPIO_WriteBitValue(GPIOB,I2C_PIN_SDA,n)
#define I2C_READ_SDA         GPIO_ReadInputBit(GPIOB,I2C_PIN_SDA)
```

Relevant instructions for the above macro definitions:

I2C_SCL_SET：　SCL line is pulled up

I2C_SCL_RESET：　SCL line is pulled down

I2C_SDA_SET：　SDA line is pulled up

I2C_SDA_RESET：　SDA line is pulled down

I2C_SDA_WRITE(n): SDA line level is set to 0 or 1

I2C_READ_SDA：　Read the current level of SDA line

This section is the I2C driver code, which implements such functions as I2C initialization (IO port), I2C start, I2C stop, ACK, I2C read and write. In other functions, it can communicate with the external I2C just by calling the related I2C function. This segment of code is not limited to AT24C02 and can be applied to other modules that use the I2C protocol for communication.

5.2 AT24C02 related function

5.2.1 AT24C02_Init

```
void AT24C02_Init(void)
{
    I2C_Init();
}
```

AT24C02_Init() function calls I2C_Init() function API, and GPIO pin is initialized.

5.2.2 AT24C02_ReadOneByte

```
uint8_t AT24C02_ReadOneByte(uint16_t ReadAddr)
{
    uint8_t temp=0;

    I2C_Start();

  I2C_Send_Byte(0XA0+((ReadAddr/256)<<1)); //transmitter address 0XA0, write data

    I2C_Wait_Ack();
    I2C_Send_Byte(ReadAddr%256); //transmit low address
    I2C_Wait_Ack();
    I2C_Start();
    I2C_Send_Byte(0XA1);           //Enter read mode
    I2C_Wait_Ack();
   temp=I2C_Read_Byte(0);
    I2C_Stop();                    //Generate a stop condition
```

```
    return temp;
}
```

AT24C02_ReadOneByte() reads a byte, and "ReadAddr" represents the address of the data to be read by the AT24C02 module. Before transmitting the read command, AT24C02 needs to first transmit a write command, set the address or register of the device to be read, and then transmit a read command to perform read operation on the device.

5.2.3 AT24C02_WriteOneByte

```
void AT24C02_WriteOneByte(uint16_t WriteAddr,uint8_t DataToWrite)
{
    I2C_Start();
    I2C_Send_Byte(0XA0+((WriteAddr/256)<<1)); //Transmitter address 0XA0, write data

    I2C_Wait_Ack();
    I2C_Send_Byte(WriteAddr%256);   //transmit low address
    I2C_Wait_Ack();
    I2C_Send_Byte(DataToWrite);     //Transmit a byte
    I2C_Wait_Ack();
  I2C_Stop(); //Generate a stop condition
    Delay_ms(10);
}
```

In the AT24C02_WriteOneByte () function, "WriteAddr" represents the address to write data to, and "DataToWrite" represents the data to be written. The write operation of the AT24C02 module is not as complex as the read operation. After the write command is transmitted, transmit the address and data to write to the device to complete the write operation of the module.

5.2.4 AT24C02_WriteLenByte

```
void AT24C02_WriteLenByte(uint16_t WriteAddr,uint32_t DataToWrite,uint8_t Len)
{
    uint8_t t;
```

```
    for(t=0;t<Len;t++)
    {
        AT24C02_WriteOneByte(WriteAddr+t,(DataToWrite>>(8*t))&0xff);
    }
}
```

AT24C02_WriteLenByte(), write “DataToWrite” with a length of "Len" to “WriteAddr”. "WriteAddr" represents the address to write data to, "DataToWrite" represents the data to be written, and "Len" represents the length of the data to be written.

5.2.5 AT24C02_ReadLenByte

```
uint32_t AT24C02_ReadLenByte(uint16_t ReadAddr,uint8_t Len)
{
    uint8_t t;
    uint32_t temp=0;
    for(t=0;t<Len;t++)
    {
        temp<<=8;
        temp+=AT24C02_ReadOneByte(ReadAddr+Len-t-1);
    }
    return temp;
}
```

AT24C02_ReadLenByte(), in data with a read length of “Len” in “ReadAddr”, “ReadAddr” is the address of the read data, and “Len” is the length of the read data.

5.2.6 AT24C02_Check

```
uint8_t AT24C02_Check(void)
{
    uint8_t temp;
    temp=AT24C02_ReadOneByte(255);//Avoid writing AT24C02 every time the machine is started
```

```
    if(temp==0X55)
    {
        return 0;
    }
    else            //Exclude situations of first initialization
    {
        AT24C02_WriteOneByte(255,0X55);
        temp=AT24C02_ReadOneByte(255);
        if(temp==0X55)
        {
            return 0;
        }
    }
    return 1;
}
```

AT24C02_Check(), check whether AT24C02 is normal, and write 0x55 in the last byte of AT24C02 module. If the first connection cannot be written or the data recognized by the second connection is not "0x55", it means that the module is abnormal and returns 1; on the contrary, it returns 0.

5.2.7 **AT24C02_Read**

```
void AT24C02_Read(uint16_t ReadAddr,uint8_t *pBuffer,uint16_t NumToRead)
{
    while(NumToRead)
    {
        *pBuffer++=AT24C02_ReadOneByte(ReadAddr++);
        NumToRead--;
    }
}
```

AT24C02_Read(), read "NumToRead" bytes of the "ReadAddr" address, and store it in "pBuffer" array. "ReadAddr" represents the read address of the controlled device, "NumToRead" represents the number of bytes to be read, and "pBuffer" is usually an empty array used to store the read data.

5.2.8 AT24C02_Write

```
void AT24C02_Write(uint16_t WriteAddr,uint8_t *pBuffer,uint16_t NumToWrite)
{
    while(NumToWrite--)
    {
        AT24C02_WriteOneByte(WriteAddr,*pBuffer);
        WriteAddr++;
        pBuffer++;
    }
}
```

AT24C02_Write(), write "pBuffer" array with the byte number of "NumToWrite" to the address of "WriteAddr". "WriteAddr" represents the address to write data to, "NumToWrite" represents the length of the data to be written, and "pBuffer" represents the array to be written.

This part of the code actually operates the AT24C02 chip through the I2C interface, defines and implements the API of relevant communication.

5.3 Main function

```
uint8_t data[10] = {0,1,2,3,4,5,6,7,8,9};
uint8_t size = sizeof(data);


int main(void)
{
    uint8_t dataBuffer[size];
    /*Serial port initialization configuration*/
    USART_Config_T usartConfig;
```

```
    usartConfig.baudRate = 115200;
    usartConfig.hardwareFlow = USART_HARDWARE_FLOW_NONE;
    usartConfig.mode = USART_MODE_TX_RX;
    usartConfig.parity = USART_PARITY_NONE;
    usartConfig.stopBits = USART_STOP_BIT_1;
    usartConfig.wordLength = USART_WORD_LEN_8B;

    APM_MINI_COMInit(COM1,&usartConfig);
    APM_MINI_LEDInit(LED2);
    APM_MINI_PBInit(BUTTON_KEY1,BUTTON_MODE_GPIO);
    APM_MINI_PBInit(BUTTON_KEY2,BUTTON_MODE_GPIO);

    Delay_Init();
    AT24C02_Init();
    printf("AT24C02 Test!!\r\n");
    /*Check if the AT24C02 module is normal*/
   while(AT24C02_Check())
   {
        APM_MINI_LEDToggle(LED2);
        Delay_ms(200);
   }
    printf("AT24C02 has ready!!!\r\n");
    while(1)
    {
    /*Check if KEY1 is pressed*/
        if(!APM_MINI_PBGetState(BUTTON_KEY1))
        {
            Delay_ms(200);
```

```
                AT24C02_Write(0,(uint8_t*)data,size); // Write data to the address 0x00
                printf("AT24C02 Write!!!\r\n");
            }
        /*Check if KEY2 is pressed*/
            if(!APM_MINI_PBGetState(BUTTON_KEY2))
            {
                Delay_ms(200);
                AT24C02_Read(0,dataBuffer,size); //Read the data with an address of 0x00 and
store to dataBuffer
                printf("AT24C02 Read!!!\r\n");

                for (int i = 0; i < size ; i++)
                {
                    printf("0x%x ,",dataBuffer[i]);//Print the read data through the serial port
                }
                printf("\r\n");
            }
        }
}
```

This segment of code first detects whether the AT24C02 module is working normally after the microcontroller is powered on. If the module is not connected or not working normally, LED2 flashes; if it works normally, the writing of AT24C02 can be controlled through the KEY1, and the reading of AT24C02 can be controlled through the KEY2.

6 Experimental Phenomena

After the code is compiled successfully, we can download the code to the APM32F103ZE development board, connect the SCL and SDA of the AT24C02 module to PB6 and PB7 of APM32F103ZE, output PA9 and PA10 as serial ports, write data with the KEY1, and then read data with the KEY2, as shown in the figure.

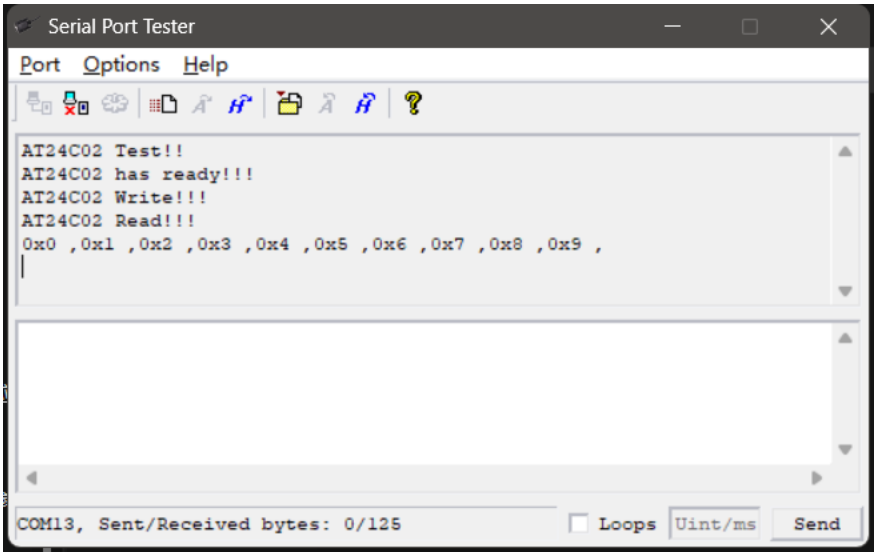


Figure 12 Print Output Data through Serial Port

The data is printed through the serial port; at the same time, if LED2 is normally on when the program is running, it indicates that the AT24C02 module is working normally.

7 Revision history

Table 1 Document Revision History

Date	Version	Revision History
August 23, 2023	1.0	New edition

Statement

This document is formulated and published by Geehy Semiconductor Co., Ltd. (hereinafter referred to as “Geehy”). The contents in this document are protected by laws and regulations of trademark, copyright and software copyright. Geehy reserves the right to make corrections and modifications to this document at any time. Please read this document carefully before using Geehy products. Once you use the Geehy product, it means that you (hereinafter referred to as the “users”) have known and accepted all the contents of this document. Users shall use the Geehy product in accordance with relevant laws and regulations and the requirements of this document.

1. Ownership

This document can only be used in connection with the corresponding chip products or software products provided by Geehy. Without the prior permission of Geehy, no unit or individual may copy, transcribe, modify, edit or disseminate all or part of the contents of this document for any reason or in any form.

The “极海” or “Geehy” words or graphics with “®” or “™” in this document are trademarks of Geehy. Other product or service names displayed on Geehy products are the property of their respective owners.

2. No Intellectual Property License

Geehy owns all rights, ownership and intellectual property rights involved in this document.

Geehy shall not be deemed to grant the license or right of any intellectual property to users explicitly or implicitly due to the sale or distribution of Geehy products or this document.

If any third party’s products, services or intellectual property are involved in this document, it shall not be deemed that Geehy authorizes users to use the aforesaid third party’s products, services or intellectual property, unless otherwise agreed in sales order or sales contract.

3. version update

Users can obtain the latest document of the corresponding models when ordering geehy products.

If the contents in this document are inconsistent with geehy products, the agreement in thesales order or the sales contract shall prevail.

4. information reliability

The relevant data in this document are obtained from batch test by geehy laboratory or cooperative third-party testing organization. However, clerical errors in correction or errors caused by differences in testing

environment may occur inevitably. Therefore, users should understand that geehy does not bear any responsibility for such errors that may occur in this document. The relevant data in this document are only used to guide users as performance parameter reference and do not constitute geehy's guarantee for any product performance.

Users shall select appropriate geehy products according to their own needs, and effectively verify and test the applicability of geehy products to confirm that geehy products meet their own needs, corresponding standards, safety or other reliability requirements. If loses are caused to users due to the user's failure to fully verify and test geehy products, geehy will not bear any responsibility.

5. legality

Users shall abide by all applicable local laws and regulations when using this document and the matching geehy products. Users shall understand that the products may be restricted by the export, re-export or other laws of the countiries of the products suppliers, geehy, geehy distributors and users.

Users (on behalf or itself, subsidiaries and affiliated enterprises) shall agree and promise to abide by all applicable laws and regulations on the export and re-export of geehy products and/or technologies and direct products.

6. disclaimer of warranty

This document is provided by geehy "as is" and there is no warranty of any kind, either expressed or implied, including, but not limited to, the warranties of merchantability and fitness for a particular purpose, to the extent permitted by applicable law.

Geehy will bear no responsibility for any disputes arising from the subsequent design or use by users.

7. limitation of liability

In no event unless required by applicable law or agreed to in writing will geehy or any other party who provide the document "as is", be liable for damages, including any general, special, direct, incidental or consequential damages arising out of the use or inability to use the document (including but not limited to loss of data or data being rendered inaccurate or losses sustained by users or third parties).

8. scope of application

The information in this document replaces the information provided in all previous versions of the document.