Geehy SEMICONDUCTOR

# 应用笔记

## Application Note

**文档编号：AN1102**

## APM32F103 软件 I2C 读写 EEPROM

**版本：V1.0**

© 珠海极海半导体有限公司

# 1 引言

本应用笔记提供如何使用 APM32F103 系列的普通 IO 口进行模拟 I2C 时序，并与 AT24C02 实现双向通信。

APM32F103 内置 2 个 I2C 总线接口，均可工作与多主模式或从模式，支持标准和快速模式。I2C 接口支持 7 位或 10 位寻址，7 位从模式时支持从地址寻址。内置了硬件 CRC 发生器、校验器。它们可以使用 DMA 操作并支持 SMBus 总线 2.0 版/PMBus 总线。

# 目录

**1 引言 ........ 1**

**2 I2C 简介 ........ 4**

2.1 I2C 协议概述 ........ 4

2.2 数据有效性 ........ 6

2.3 I2C 时序 ........ 6

**3 AT24C02 介绍 ........ 9**

**4 硬件设计 ........ 10**

**5 软件设计 ........ 11**

5.1 GPIO 模拟 I2C 配置 ........ 11

5.2 AT24C02 相关函数 ........ 19

5.3 Main 函数 ........ 24

**6 实验现象 ........ 27**

**7 版本历史 ........ 28**

# 2 I2C 简介

I2C 是一种短距离通信协议，物理实现上，I2C 总线由两根信号线（SDA 与 SCL）和一根地线组成，两根信号线为双向传输的。

• 两根信号线，SCL 时钟线、SDA 数据线。由 SCL 为 SDA 提供时序，SDA 串行发送/接收数据。

• SCL、SDA 这两根信号线均为双向。

• 两个系统使用 I2C 总线通信时共用同一根地线。

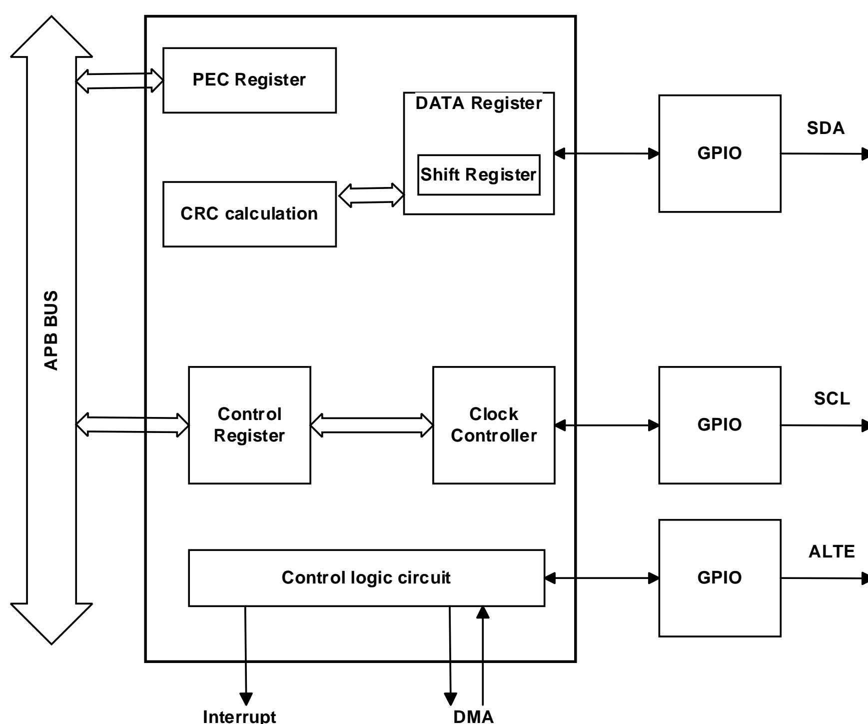


图 1 I2C 功能结构图

## 2.1 I2C 协议概述

数据以帧的形式传输，每一帧中由 1 个字节(8 位)组成。

在 SCL 的上升沿阶段，SDA 需要保持稳定，SDA 在 SCL 为低期间作出改变。

除了数据帧，I2C 总线还有起始信号，停止信号，应答信号。

- 起始位：在 SCL 为稳定的高电平期间，SDA 的一个下降沿启动传输。

- 停止位：在 SCL 为稳定的高电平期间，SDA 的一个上升沿停止传输。
- 应答位：用于表示一个字节传输成功。总线发送器(无论主机还是从机),

在发送 8 个位的数据后，SDA 将释放(由输出变为输入)，在第九个时钟脉冲期间，接收器将 SDA 拉低，来应答接收到了数据。

**I2C 通信读写过程**

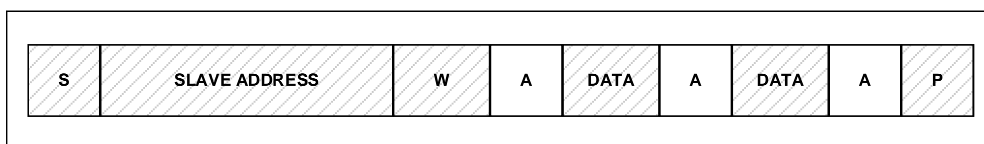


**图 2 主机写数据至从机**

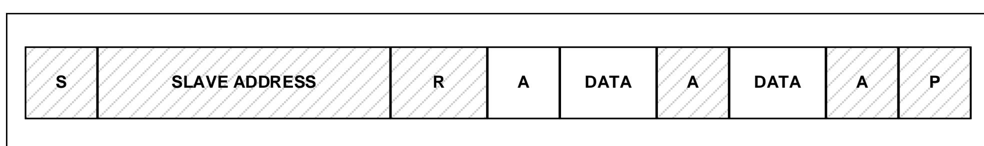


**图 3 主机由从机读取数据**

备注：（I2C 发送 8 位命令，最后一位代表读写位）

（1） 阴影部分：此数据由主机传输到从机。

（2） S：起始信号。

（3） SLAVE ADDRESS：从机地址。

（4） 非阴影部分：此数据由从机传输到主机。

（5） R/W ：传输方向选择位。

（6） 1 ：读取。

（7） 0 ：写入。

（8） P：停止信号。

起始信号产生后，所有从机都将等待主机发送的从机地址信号，I2C 总线中，每个设备的地址都是唯一的，当地址信号与设备地址匹配后，从机将被选中，没被选中的从机将忽略以后的数据信号。

**主机方向为写数据时**

广播完地址后，接收到应答信号，主机向从机发传输数据，数据长度为一个字节，主机每次发完一个字节数据后，都需等待从机发送的应答信号，当传输的所有字节完成后，主机向从机发送一个停止信号（STOP），表示为传输完成。

**主机方向为读数据时**

广播完地址后，接收到应答信号，从机开始向主机传输数据，数据包的大小为 8 位，从机每发送完一个字节数据，都要等待主机的应答信号，当主机想停止接收数据时，需要向从机返回一个非应答信号，则从机自动停止数据传输。

## 2.2 数据有效性

数据发送过程中，时钟信号 SCL 高电平期间，SDA 线上数据必须稳定，只有当 SCL 在低电平期间 SDA 的电平状态才能发生改变，每个数据比特传输都需要一个时钟脉冲。

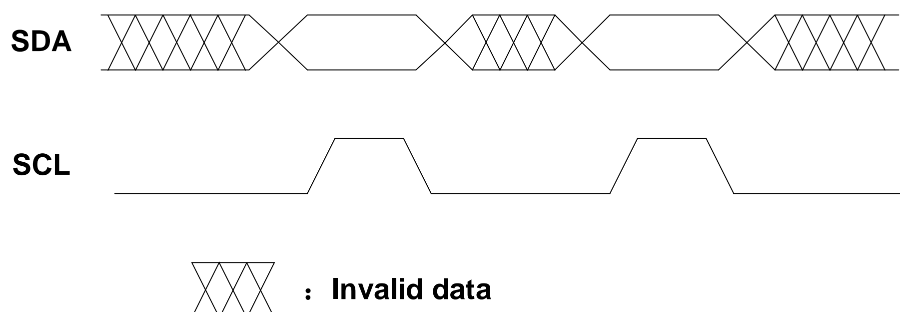


**图 4 SDA 时序图**

## 2.3 I2C 时序

I2C 在传输数据过程中共 3 条信号，分别是开始信号，结束信号和应答信号。

### 2.3.1 I2C 开始信号（START）

SATRT 信号定义为：时钟线 SCL 保持高电平期间，数据线 SDA 电平拉低，标志着一次数据传输的开始。启动信号是一种电平跳转的时序信号，而不是一种电平信号。该信号由主设备发出，在建立该信号之前，I2C 总线必须处于空闲状态。

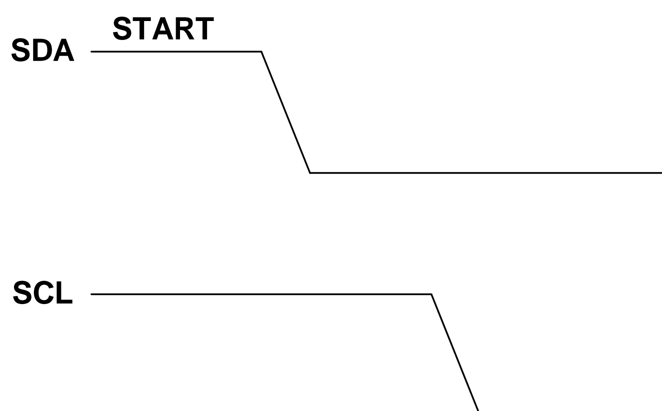

图 5 START 信号时序图

### 2.3.2 I2C 停止信号（STOP）

STOP 信号定义为：时钟线 SCL 保持高电平期间，数据线 SDA 释放，返回高电平，标志着一次数据传输的结束。停止信号也是一种电平跳转的时序信号，而不是一种电平信号。该信号由主设备发出，建立该信号之后，I2C 返回空闲状态。

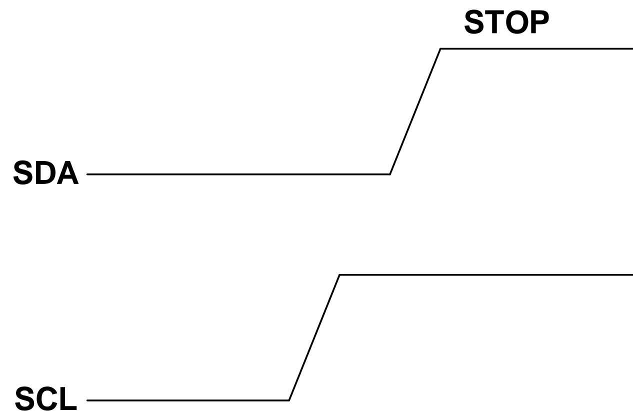


图 6 STOP 信号时序图

### 2.3.3 I2C 应答信号（ACK）

I2C 总线上的所有数据都是以 8 位字节传送的，I2C 通信时，发送器每发送一次数据，接收器都会反馈一个应答信号。

应答信号为低电平时，规定为有效应答（ACK）；应答信号为高电平时，规定为非有效应答（NACK）。反馈有效应答信号的要求：接收器在接收第 9 个脉冲前，将 SDA 线拉低，确保当 SCL 线为高电平时，SDA 线输出稳定的低电平。

如果接收器是主设备，则当其收到从设备发出的最后一个字节时，会发送一个 NACK 信号，以通知从设备停止数据传送，并释放 SDA 线，以便主设备发送一个停止信号。

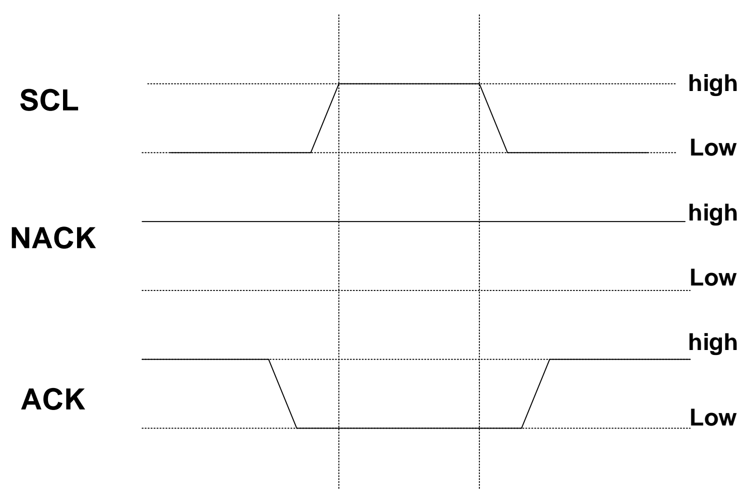


图 7 ACK 与 NACK 信号时序图

# 3 AT24C02 介绍

AT24C02 是一款串行 EEPROM（Electrically Erasable Programmable Read-Only Memory）芯片，由 Atmel（现在是 Microchip Technology）公司生产。它是 AT24C 系列芯片中的一员，具有 2K 位（256 字节）的存储容量。

如下是 AT24C02 的性能特点。

（1） 采用 I2C（Inter-Integrated Circuit）总线接口，这是一种常用的串行通信协议，可以在多个设备之间进行数据传输。它支持标准模式（100 KHz）和快速模式（400 KHz）的 I2C 通信速率。

（2） 具有可擦写和可编程的功能，可以通过电源供电进行数据的读取和写入。它采用 8 位地址寻址，可以连接多个 AT24C02 芯片，使得系统能够扩展更大的存储容量。

（3） 数据存储是非易失性的，即使在断电情况下，数据仍然可以保持。它还具有内部写保护功能，可以通过硬件或软件方式来保护存储的数据，防止误写或擦除。

（4） 广泛应用于各种电子设备中，例如存储配置信息、校准数据、序列号、日志记录等。由于其容量适中、接口简单、低功耗等特点，它在嵌入式系统和小型电子设备中得到了广泛应用。

总体而言，AT24C02 芯片具有较小的存储容量、简单的接口、良好的可靠性和低功耗等特点，适用于许多嵌入式系统和小型电子设备中的数据存储需求。

如下为 AT24C02 的通信时序图。

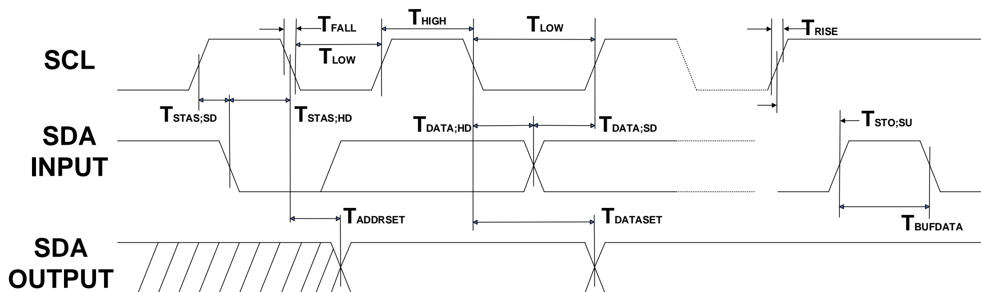


**图 8 AT24C02 通信时序图**

# 4 硬件设计

本次设计使用 APM32F103ZE 的 PB6、PB7 引脚进行 GPIO 模拟 I2C，将 AT24C02 的 SCL 和 SDA 分别连接在 APM32F103ZE 的 PB6、PB7 上，连接关系如图所示。

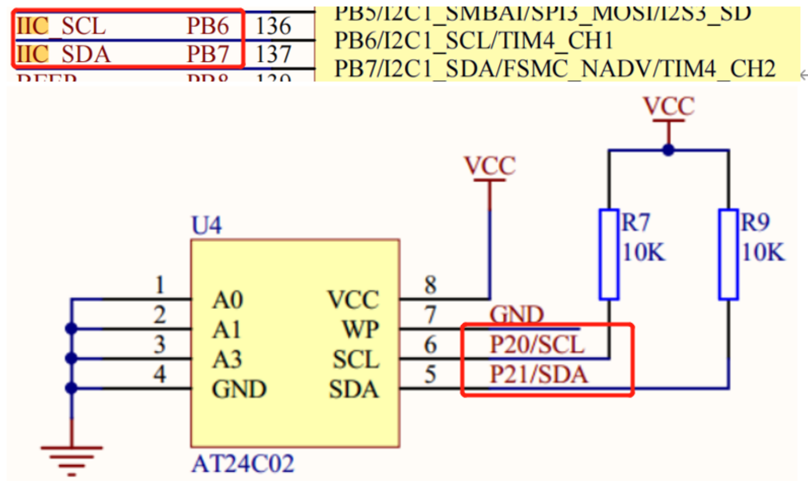


**图 9 APM32F103ZE 与 AT24C02 的连接图**

# 5 软件设计

本章主要介绍 GPIO 模拟 I2C 所实现的代码，以及 APM32F103ZE 使用软件 I2C 与 AT24C02 通信。其工作流程图如下所示。

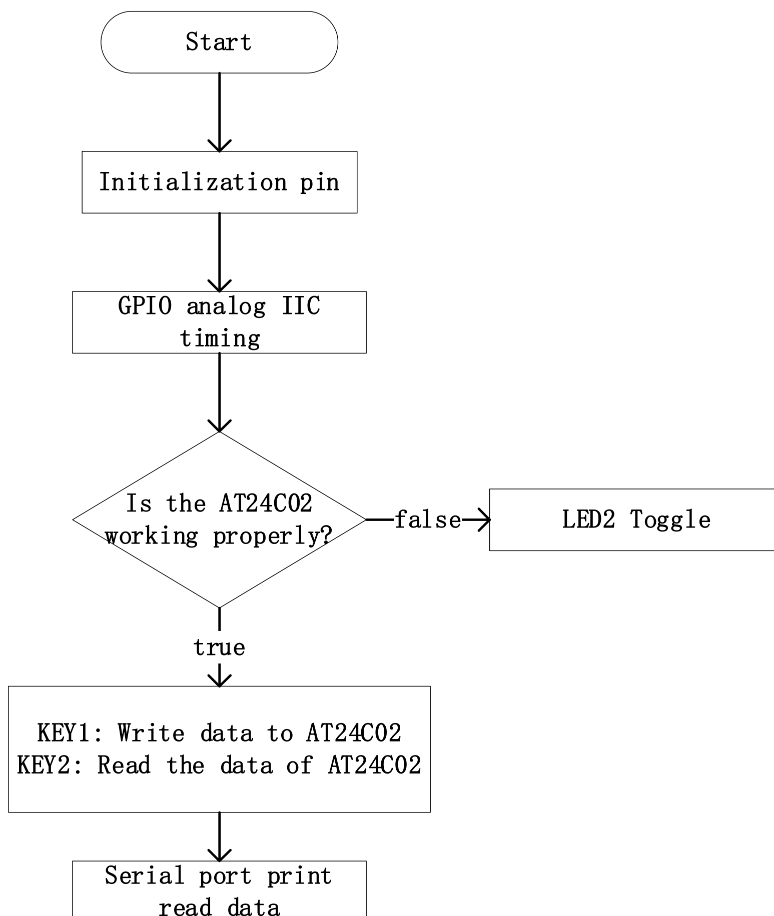


图 10 系统工作流程图

## 5.1 GPIO 模拟 I2C 配置

### 5.1.1 SDA_IN_OUT

//SDA 设置为输入模式

```
void SDA_IN(void)

{

      GPIOB->CFGLOW &= 0X0FFFFFFF;

      GPIOB->CFGLOW |= (uint32_t)8<<28;
```

```
}
//SDA 设置为输出模式
void SDA_OUT(void)
{
    GPIOB->CFGLOW &= 0X0FFFFFFF;
    GPIOB->CFGLOW |= (uint32_t)3<<28;
}
```

参照如下 APM32F103 用户手册关于 CFGLOW 寄存器的说明，SDA_IN()函数将 GPIOB 中的 CFGLOW 寄存器的高 4 位配置成了“1000”，对应设置 PB7 引脚为输入模式；SDA_OUT()函数将 GPIOB 中的 CFGLOW 寄存器高 4 位配置成了“0011”，对应 PB7 引脚为推挽输出模式。

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

**图 11 CFGLOW 寄存器说明**

### 5.1.2 I2C_Init

```
//PB6（SCL）、PB7（SDA）初始化
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 ;          //推挽输出
    gpioConfig.speed = GPIO_SPEED_50MHz;
    GPIO_Config(GPIOB, &gpioConfig);
    GPIO_SetBit(GPIOB,I2C_PIN_SCL|I2C_PIN_SDA);
}
```

I2C_Init()函数配置了 GPIOB 的时钟，将 PB6（SCL）、PB7（SDA）配置成了推挽输出模式，由于 I2C 时序开始时为高电平，故将 PB6、PB7 默认置为高电平。

### 5.1.3 I2C_Start

```
//I2C 开始时序
void I2C_Start(void)
{
    SDA_OUT();                  //SDA 输出
    I2C_SDA_SET;
    I2C_SCL_SET;
    Delay_us(4);
    I2C_SDA_RESET;          //START:当 CLK 为高电平时，DATA 从高电平变为低电平
    Delay_us(4);
    I2C_SCL_RESET;            //钳住 I2C 总线，准备发送或接收数据
}
```

根据 I2C 开始时序（START）图，当 SCL 线为稳定的高电平时，将 SDA 线拉低代表 I2C 时序的

开始，故先将 I2C_SDA_RESET，延时一段时间后，再将 I2C_SCL_RESET，即可模拟 I2C 的 START 时序。

### 5.1.4 I2C_Stop

```
//I2C 停止时序
void I2C_Stop(void)
{
    SDA_OUT();                  //SDA 线输出
    I2C_SCL_RESET;
    I2C_SDA_RESET;              //STOP:当 CLK 为高电平时，DATA 从低电平变为高电平
    Delay_us(4);
    I2C_SCL_SET;
    I2C_SDA_SET;                //发送 I2C 总线结束信号
    Delay_us(4);
}
```

根据 I2C 停止时序（STOP）图，当 SCL 线为稳定的高电平时，将 SDA 线拉高代表 I2C 时序的停止，故先将 I2C_SCL_SET，再将 I2C_SDA_SET，即可模拟 I2C 的 STOP 时序。

### 5.1.5 I2C_Wait_Ack

```
// 等待 ACK 应答
uint8_t I2C_Wait_Ack(void)
{
    uint8_t ucErrTime = 0;
    SDA_IN();                   //SDA 设置为输入
    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;            //时钟输出 0
    return 0;
}
```

I2C_Wait_Ack()函数中，将 SDA 设置为输入模式，以读取 SDA 电平。将 SCL 和 SDA 拉高，若 SDA 线识别到低电平，则代表收到了 ACK，返回 0；若 250 次之后未收到 ACK 则返回 1，同时中断此次 I2C 传输。

在 I2C_Wait_Ack()函数中，成功接收到 ACK 后，会执行 I2C_SCL_RESET 这一段代码，其会将 SCL 拉低一段时间，这个时间被称为“时钟拉低期”。拉低 SCL 的作用为：

（1） 稳定数据：拉低 SCK 线可以确保数据线（SDA）上的数据稳定，避免在时钟转换期间发生数据的变化。

（2） 适应从设备速度：从设备可能由于处理数据的速度较慢而需要更多时间来处理接收到的数据。通过拉低 SCK 线，主设备可以等待从设备完成数据处理，使得数据传输的速度适应从设备的处理速度。

（3） 时钟同步：时钟拉低期可以确保主设备和从设备之间的时钟同步，使得数据传输的时序正确。

总的来说，将 SCK 线拉低一段时间是为了稳定数据传输、适应从设备的速度并确保时钟同步。这样可以保证在 I2C 通信中数据的准确传输和正确处理。

### 5.1.6 I2C_Ack

```
//GPIO 模拟产生 ACK 时序
void I2C_Ack(void)
{
    I2C_SCL_RESET;
```

```
    SDA_OUT();
    I2C_SDA_RESET;
    Delay_us(2);
    I2C_SCL_SET;
    Delay_us(2);
    I2C_SCL_RESET;
}
```

根据 I2C 应答时序（ACK）图，当 SCL 发生脉冲变化时，SDA 一直处于低电平，确保当 SCL 为稳定的高电平时，SDA 一直为低电平。故先将 I2C_SDA_RESET，而后 SCL 产生一次脉冲，SCL 由“0”跳变为“1”，再由“1”跳变为“0”，模拟 I2C 时序的 ACK 信号。

### 5.1.7 I2C_NAck

```
//GPIO 模拟产生 NACK 时序
void I2C_NAck(void)
{
    I2C_SCL_RESET;
    SDA_OUT();
    I2C_SDA_SET;
    Delay_us(2);
    I2C_SCL_SET;
    Delay_us(2);
    I2C_SCL_RESET;
}
```

根据 I2C 非应答时序（NACK）图，当 SCL 发生脉冲变化时，SDA 一直处于高电平，确保当 SCL 为稳定的高电平时，SDA 一直为高电平。故先将 I2C_SDA_RESET，而后 SCL 产生一次脉冲，SCL 由“0”跳变为“1”，再由“1”跳变为“0”，模拟 I2C 时序的 NACK 信号。

### 5.1.8 I2C_Send_Byte

```
//GPIO 模拟 I2C：发送一个字节
void I2C_Send_Byte(uint8_t txd)
```

```
{
    uint8_t t;
    SDA_OUT();
    I2C_SCL_RESET;                    //拉低时钟开始数据传输
    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);
    }
}
```

由于 I2C 必须在 SCL 为低电平时，SDA 线才能跳变，故现将 I2C_SCL_RESET，而后调用 I2C_SDA_WRITE()函数逐位写入对应被控设备的地址，写完一位数据后，将 SCL 拉高，确保此次传输数据的稳定性，而后 I2C_SCL_RESET，准备下一次的数据传输。

#### 5.1.9 I2C_Read_Byte

```
//GPIO 模拟 I2C：读取一个字节
uint8_t I2C_Read_Byte(unsigned char ack)
{
    unsigned char i,receive=0;
    SDA_IN();                         //SDA 设置为输入
    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();                    //发送 NACK
        }
        else
        {
            I2C_Ack();                     //发送 ACK
        }
        return receive;
}
```

I2C_Read_Byte()函数中，SDA 设置为输入模式，以便对 IO 口进行读取操作。随后给予 SCL 一个上升沿，以此触发数据的采样，函数中定义了一个"receive"变量，若 I2C_READ_SDA 为 0，则代表读取到的数据为"0"，则 recvice=0；相反，则 receive=1。"receive"变量用来存储读取的数据。"ack"变量的作用是提示是否需要被控设备发送 ACK 应答信号，若"ack=1"，则代表控制设备的数据还未传输完，需要返回被控设备返回 ACK 应答；若"ack=0"，则代表控制设备的数据已经传输完，无需 ACK 应答，需要被控设备返回 NACK 应答。

### 5.1.10 宏定义

上部分代码所使用的相关宏定义如下。

```
//IO 操作函数
#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)
```

对于上述宏定义的相关说明：

I2C_SCL_SET：　　SCL 线拉高

I2C_SCL_RESET：　SCL 线拉低

I2C_SDA_SET：　　SDA 线拉高

I2C_SDA_RESET：　SDA 线拉低

I2C_SDA_WRITE(n)：SDA 线电平置 0 或 1

I2C_READ_SDA：　读取 SDA 线当前电平

该部分为 I2C 驱动代码，实现包括 I2C 的初始化（IO 口）、I2C 开始、I2C 结束、ACK、I2C 读写等功能，在其他函数中，只需调用其相关的 I2C 函数即可和外部的 I2C 通信，此段代码并不局限于 AT24C02，可应用与其他使用 I2C 协议通信的模块。

## 5.2 AT24C02 相关函数

### 5.2.1 AT24C02_Init

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

AT24C02_Init()函数，调用了 I2C_Init()函数 API，GPIO 引脚初始化。

### 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)); //发送器件地址 0XA0,写数据

    I2C_Wait_Ack();
    I2C_Send_Byte(ReadAddr%256);    //发送低地址
    I2C_Wait_Ack();
    I2C_Start();
    I2C_Send_Byte(0XA1);            //进入读模式
    I2C_Wait_Ack();
    temp=I2C_Read_Byte(0);
    I2C_Stop();                     //产生一个停止条件
    return temp;
}
```

AT24C02_ReadOneByte()，读取一个字节，“ReadAddr”代表 AT24C02 模块要读取数据对应的地址。AT24C02 在发送读命令之前，需要先发送写命令，设置要读取设备的地址或寄存器，而后发送读命令，才能对设备进行读操作。

### 5.2.3 AT24C02_WriteOneByte

```
void AT24C02_WriteOneByte(uint16_t WriteAddr,uint8_t DataToWrite)
{
    I2C_Start();
    I2C_Send_Byte(0XA0+((WriteAddr/256)<<1));//发送器件地址 0XA0,写数据

    I2C_Wait_Ack();
    I2C_Send_Byte(WriteAddr%256);    //发送低地址
    I2C_Wait_Ack();
```

```
        I2C_Send_Byte(DataToWrite);         //发送字节
        I2C_Wait_Ack();
        I2C_Stop();//产生一个停止条件
        Delay_ms(10);
}
```

AT24C02_WriteOneByte ()函数中，“WriteAddr”代表要写入数据的地址，“DataToWrite”代表要写入的数据。对于 AT24C02 模块的写操作没有读操作那么复杂，在发送写命令后，再发送要写入设备的地址和数据，即可完成模块的写入操作。

### 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()，在“WriteAddr”写入长度为“Len”的“DataToWrite”。“WriteAddr”代表要写入数据的地址，“DataToWrite”为要写入的数据，“Len”代表要写入数据的长度。

### 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()，在“ReadAddr”读取长度为“Len”的数据，“ReadAddr”为被读取数据的地址，“Len”为被读取数据的长度。

### 5.2.6 AT24C02_Check

```
uint8_t AT24C02_Check(void)
{
      uint8_t temp;
      temp=AT24C02_ReadOneByte(255);//避免每次开机都写 AT24C02
      if(temp==0X55)
      {
            return 0;
      }
      else              //排除第一次初始化的情况
      {
            AT24C02_WriteOneByte(255,0X55);
            temp=AT24C02_ReadOneByte(255);
            if(temp==0X55)
            {
                  return 0;
            }
      }
      return 1;
}
```

AT24C02_Check()，检查 AT24C02 是否正常，在 AT24C02 模块的最后一个字节写入 0x55，若首次连接写不进去或第二次连接识别的数据不是“0x55”，代表该模块异常，返回 1；相反，返回 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()，读取“ReadAddr”地址的“NumToRead”个字节，存储到“pBuffer”数组中。“ReadAddr”代表被控设备的读取地址，“NumToRead”代表需要读取的字节数，“pBuffer”通常为空数组，用来存储读取后的数据。

### 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()，在“WriteAddr”地址写入字节数为“NumToWrite”的“pBuffer”数组。“WriteAddr”代表要写入的数据的地址，“NumToWrite”代表写入数据的长度，“pBuffer”代表需要写入的数组。

这部分代码实际就是通过 I2C 接口来操作 AT24C02 芯片，定义并实现相关通信的 API。

## 5.3 Main 函数

```
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];
    /*串口初始化配置*/
    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");
    /*检测 AT24C02 模块是否正常*/
    while(AT24C02_Check())
    {
```

```
        APM_MINI_LEDToggle(LED2);
        Delay_ms(200);
    }
    printf("AT24C02 has ready!!!\r\n");
    while(1)
    {
        /*检测 KEY1 是否按下*/
        if(!APM_MINI_PBGetState(BUTTON_KEY1))
        {
            Delay_ms(200);
            AT24C02_Write(0,(uint8_t*)data,size);//向地址为 0x00 的地址写入数据
            printf("AT24C02 Write!!!\r\n");
        }
        /*检测 KEY2 是否按下*/
        if(!APM_MINI_PBGetState(BUTTON_KEY2))
        {
            Delay_ms(200);
            AT24C02_Read(0,dataBuffer,size);//读取地址为 0x00 的数据并存储至 dataBuffer
            printf("AT24C02 Read!!!\r\n");

            for (int i = 0; i < size ; i++)
            {
                printf("0x%x ,",dataBuffer[i]);//串口打印读取的数据
            }
            printf("\r\n");
        }
    }
}
```

该段代码，单片机上电后首先检测 AT24C02 模块是否正常工作，若未连接模块或工作不正常，LED2 闪烁；若正常工作，可通过 KEY1 按键来控制 AT24C02 的写入，通过另外一个按键 KEY2 来控制 AT24C02 的读取。

# 6 实验现象

在代码编译成功后，我们通过下载代码到 APM32F103ZE 开发板上，将 AT24C02 模块的 SCL、SDA 分别与 APM32F103ZE 的 PB6、PB7 连接，将 PA9、PA10 作为串口线输出，通过按键 KEY1 写入数据，然后按键 KEY2 读取数据，现象如图所示。

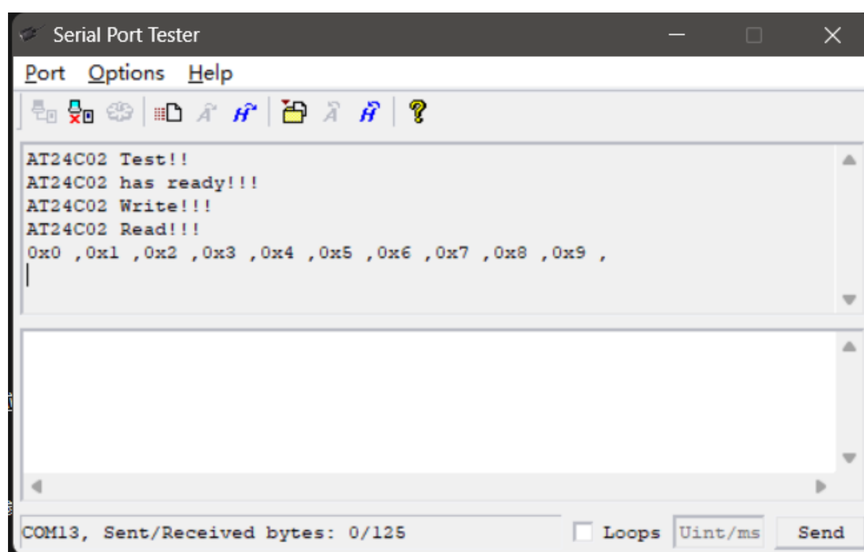


**图 12 串口打印输出数据**

串口打印读取的数据，同时，程序运行时，若 LED2 常亮，代表 AT24C02 模块正常工作。

# 7 版本历史

表格 1 文件版本历史

| 日期 | 版本 | 变更历史 |
|---|---|---|
| 2023.08.23 | 1.0 | 新建 |

# 声明

本手册由珠海极海半导体有限公司（以下简称“极海”）制订并发布，所列内容均受商标、著作权、软件著作权相关法律法规保护，极海保留随时更正、修改本手册的权利。使用极海产品前请仔细阅读本手册，一旦使用产品则表明您（以下称“用户”）已知悉并接受本手册的所有内容。用户必须按照相关法律法规和本手册的要求使用极海产品。

1、权利所有

本手册仅应当被用于与极海所提供的对应型号的芯片产品、软件产品搭配使用，未经极海许可，任何单位或个人均不得以任何理由或方式对本手册的全部或部分内容进行复制、抄录、修改、编辑或传播。

本手册中所列带有“®”或“TM”的“极海”或“Geehy”字样或图形均为极海的商标，其他在极海产品上显示的产品或服务名称均为其各自所有者的财产。

2、无知识产权许可

极海拥有本手册所涉及的全部权利、所有权及知识产权。

极海不应因销售、分发极海产品及本手册而被视为将任何知识产权的许可或权利明示或默示地授予用户。

如果本手册中涉及任何第三方的产品、服务或知识产权，不应被视为极海授权用户使用前述第三方产品、服务或知识产权，除非在极海销售订单或销售合同中另有约定。

3、版本更新

用户在下单购买极海产品时可获取相应产品的最新版的手册。

如果本手册中所述的内容与极海产品不一致的，应以极海销售订单或销售合同中的约定为准。

4、信息可靠性

本手册相关数据经极海实验室或合作的第三方测试机构批量测试获得，但本手册相关数据难免会出现校正笔误或因测试环境差异所导致的误差，因此用户应当理解，极海对本手册中可能出现的该等错误无需承担任何责任。本手册相关数据仅用于指导用户作为性能参数参照，不构成极海对任何产品性能方面的保证。

用户应根据自身需求选择合适的极海产品，并对极海产品的应用适用性进行有效验证和测试，以确认极海产品满足用户自身的需求、相应标准、安全或其它可靠性要求；若因用户未充分对极海产品进行有效验证和测试而致使用户损失的，极海不承担任何责任。

5、合规要求

用户在使用本手册及所搭配的极海产品时，应遵守当地所适用的所有法律法规。用户应了解产品可能受到产品供应商、极海、极海经销商及用户所在地等各国有关出口、再出口或其它法律的限制，用户（代表其本身、子公司及关联企业）应同意并保证遵守所有关于取得极海产品及／或技术与直接产品的出口和再出口适用法律与法规。

6、免责声明

本手册由极海“按原样”（as is）提供，在适用法律所允许的范围内，极海不提供任何形式的明示或暗示担保，包括但不限于对产品适销性和特定用途适用性的担保。

对于用户后续在针对极海产品进行设计、使用的过程中所引起的任何纠纷，极海概不承担责任。

7、责任限制

在任何情况下，除非适用法律要求或书面同意，否则极海和/或以“按原样”形式提供本手册的任何第三方均不承担损害赔偿责任，包括任何一般、特殊因使用或无法使用本手册相关信息而产生的直接、间接或附带损害（包括但不限于数据丢失或数据不准确，或用户或第三方遭受的损失）。

8、适用范围

本手册的信息用以取代本手册所有早期版本所提供的信息。

©2023 珠海极海半导体有限公司 – 保留所有权利