# IIC总线及其应用

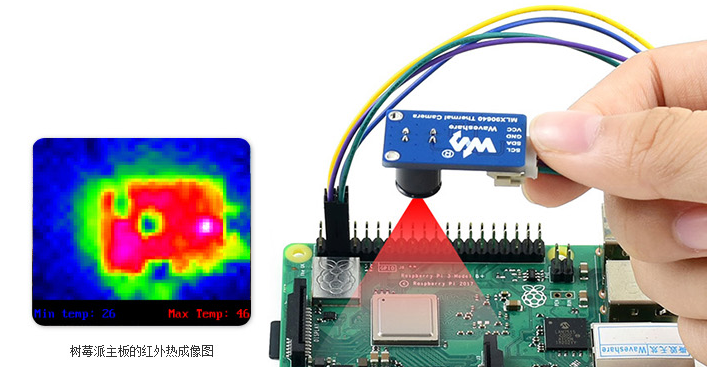
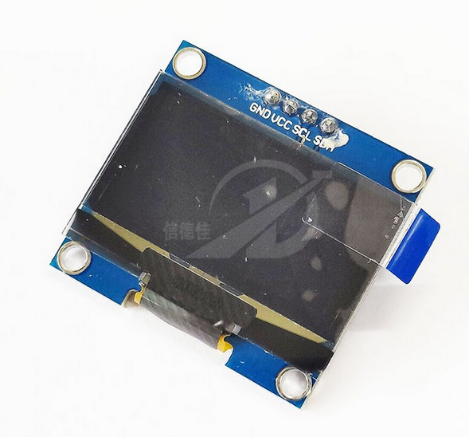
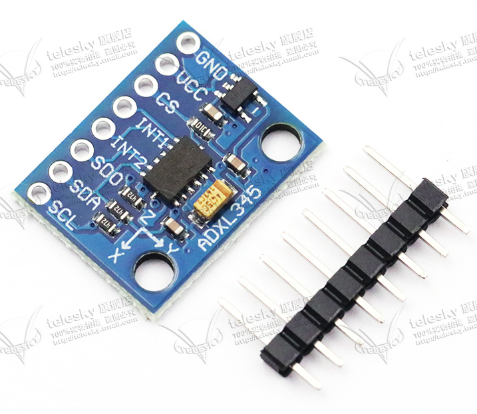
## I2C总线的基本概念

**I2C的基本概念**

I2C是Inter-Integrated Circuit的简称，读作：I-squared-C。

I2C总线是由Philips公司开发的一种简单、双向 二线制 同步 串行 总线。它只需要两根线即可在连接于总线上的器件之间传送信息。

为了让主板、嵌入式系统或手机用以连接低速周边外部设备而发展。

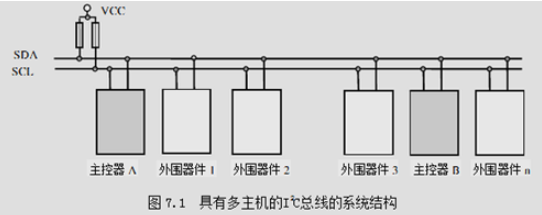


I2C总线广泛应用在EEPROM、实时时钟、LCD、及其他芯片的接口。

I2C允许相当大的工作电压范围，典型的电压基准：+3.3V或+5V。

**I2C总线的结构：**

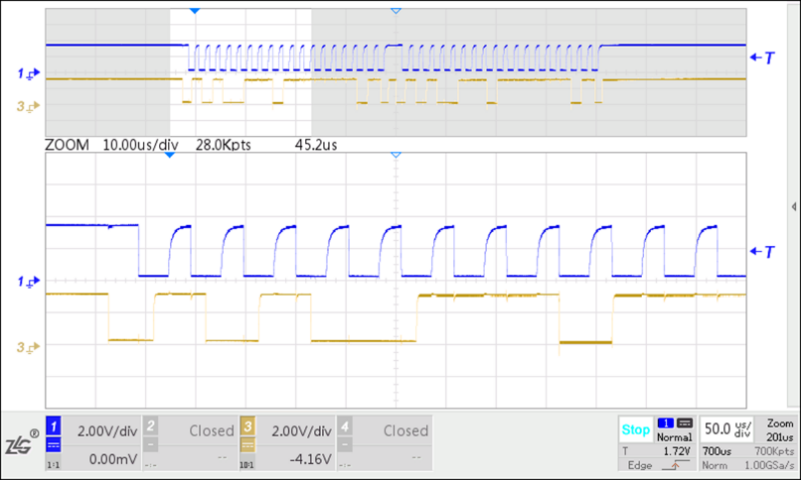
I2C总线通过两根线SDA（串行数据线）和SCL（串行时钟线）将主控制器和外围设备连接起来。



I2C的两根线SDA（串行数据线）和SCL（串行时钟线）都是双向I/O线，接口电路为开漏输出，需通过上拉电阻接电源VCC，当总线空闲时，两根线都是高电平。

I2C总线是一种多主控总线，即可以在总线上放置多个主设备节点，在停止位（P）发出后，即通讯结束后，主设备节点可以成为从设备节点。

**I2C总线的特点**：

* I2C通信双方地位不对等，通信由主设备发起，并主导传输过程，从设备按I2C协议接收主设备发送的数据，并及时给出响应。
  + 主设备节点：产生时钟并发起通信的设备节点
  + 从设备节点：接收时钟并响应主设备节点寻址的设备节点
* 主设备、从设备由通信双方决定（I2C协议本身无规定），一个节点既能当主设备，也能当从设备（需要软件进行配置）。
* 主设备负责调度总线，决定某一时刻和哪个从设备通信。同一时刻，I2C总线上只能有一对主设备、从设备通信。
* 每个I2C从设备在I2C总线通讯中有一个I2C从设备地址，该地址唯一，是从设备的固有属性，通信中主设备通过从设备地址来找到从设备。
* I2C使用一个7bit的设备地址，一组总线最多和112个节点通信。
* 连接到相同总线的 IC 数量受到总线的最大电容 400pF 限制 。
  + 总线上电容过大会导致波形失真、不完整
  + 
* 常见的I2C总线以传输速率的不同分为不同的模式：
  + 标准模式（100Kbit/s）
  + 低速模式（10Kbit/s）
  + 快速模式（400Kbit/s）
* 时钟频率可以被下降到零，即暂停通信。

## I2C总线的通讯

**I2C总线的通讯特征：**

双向、二线制、串行、同步、非差分、低速率

**串行通信**：所有的数据以位为单位在SDA线上串行传输

**同步通信**：双方工作在同一个时钟下，一般是通信的A方通过一根CLK信号线，将A设备的时钟传输到B设备，B设备在A设备传输的时钟下工作。同步通信的特征是：通信线中有CLK。

**非差分**：I2C通信速率不高，且通信距离近，使用电平信号通信。

**低速率**：I2C一般是同一个板子上的两个IC芯片间通信，数据量不大，速率低。速率：几百KHz，速率可能不同，不能超过IC的最高速率。

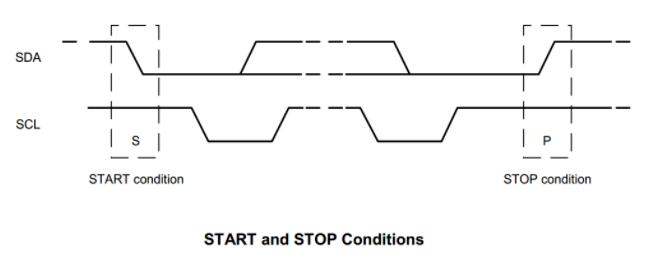
**I2C总线的状态**

**空闲态**：没有设备发生通信。SDA和SCL都是高电平

**忙态**：其中一个从设备和主设备通信，I2C总线被占用，其他从设备处于等待状态。

**I2C总线通信协议**

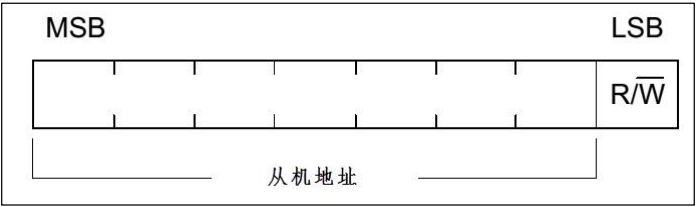
每个通信周期都由一个起始位开始通信，由一个结束位结束通信，中间部分是传递的数据。



I2C总线通讯由起始位开始通讯，由结束位停止通讯，并释放I2C总线。起始位和结束位都由主设备发出。

* 起始位（S）：在SCL为高电平时，SDA由高电平变为低电平
* 结束位（P）：在SCL为高电平时，SDA由低电平变为高电平

每个通信周期，主设备会先发8位的从设备地址（从设备地址由高7位的实际从设备地址和低1位的读/写标志位组成），主设备以广播的形式发送从设备地址，I2C总线上的所有从设备收到地址后，判断从设备地址是否匹配，不匹配的从设备继续等待，匹配的设备发出一个应答信号。



同一时刻，主设备、从设备只能有一个设备发送数据。

**数据格式与应答**

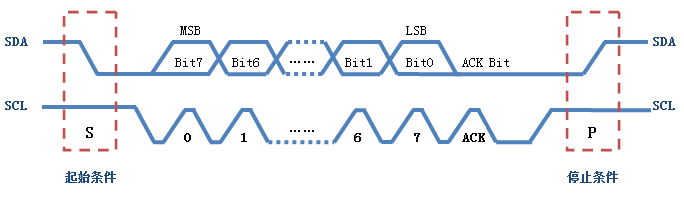
I2C数据以字节（即8bits）为单位传输，每次传输可以发送的字节数量不受限制。

每个字节传输完后都会有一个ACK应答信号。应答信号的**时钟**是由主设备产生的。

先传输最高位(MSB)

如果从设备来不及处理主设备发送的数据，从设备会保持SCL线为低电平，强迫主设备等待从设备释放SCL线，直到从设备处理完后，释放SCL线，接着进行数据传输。

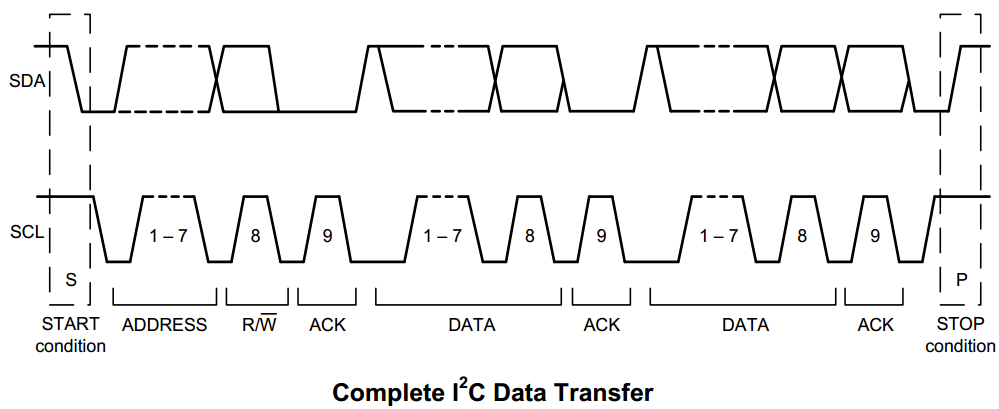
当从机不能响应从机地址时（例如它正在执行一些实时函数不能接收或发送），从机必须使数据线保持高电平，主机然后产生一个停止条件终止传输或者产生重复起始条件开始新的传输。



**写数据：**

1. 先发送一个起始位（S），
2. 主设备发送一个地址数据（由7bit的从设备地址，和最低位的写标志位组成的8bit字节数据，该读写标志位决定数据的传输方向）
3. 主设备释放SDA线，并等待从设备的应答信号（ACK）。
4. 每一个字节数据的传输都要跟一个应答信号位。
5. 数据传输以停止位（P）结束，并且释放I2C总线。





**读数据：**

1. 主设备发出一个起始位（S）
2. 主设备发送一个地址数据（由7bit的从设备地址，和最低位的写标志位组成的8bit字节数据，该读写标志位决定数据的传输方向）
3. 从设备回应(用来确定这个设备是否存在)，
4. 传输数据，传输数据之后，要有一个回应信号（确定数据是否接受完成)
5. 传输下一个数据。
6. 主设备发送一个停止信号



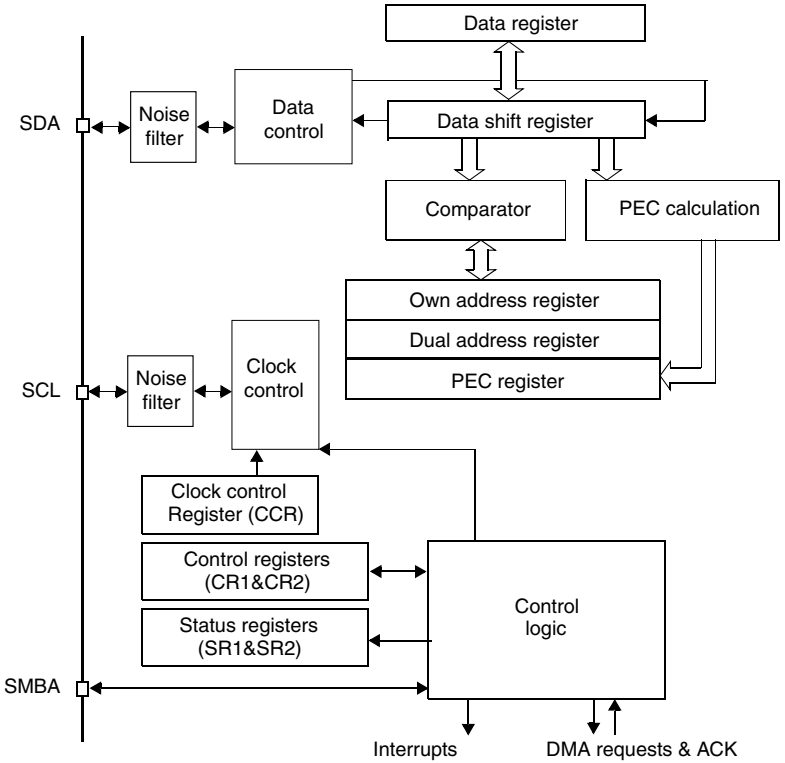
## STM32F401的I2C模块的功能

**STM32F401的I2C模块**

STM32F401提供了I2C外设模块，能够完成I2C总线的通信需求，具备以下特点：

* 支持主/从收发模式
* 支持三种传输速度
  + Standard mode (Sm, 100 Kbps)
  + Fast mode (Fm, 400 Kbps)
  + Fast-mode plus (1 Mbps)
* 支持发送接收中断
* 支持DMA
* 模拟噪声滤波器
* 数字噪声滤波器

**STM32F401的I2C外设模块结构框图**



SDA信号与数据移位寄存器相连，用于从串行总线上获取数据，或者把数据以串行的方式发送到总线上。

数据移位寄存与数据寄存器相连，用于接收或者发送数据。

数据移位寄存器跟一个比较器相连，用于判断接收的地址是否与本机地址相符。

SCL引脚连接时钟控制模块，用于产生通信用的时钟信号。

有两个控制寄存器CR1和CR2，用于设置控制逻辑。还有连个状态寄存器SR1和SR2，用于指示当前状态。

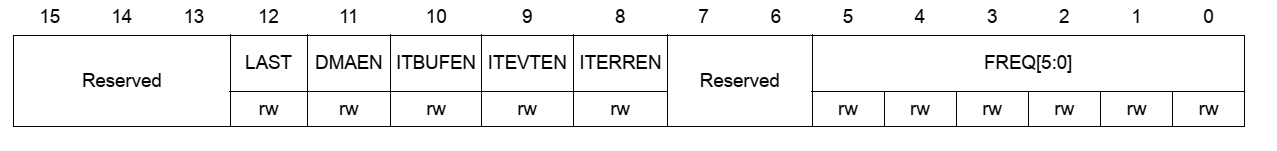
**I2C主机模式的使用：**

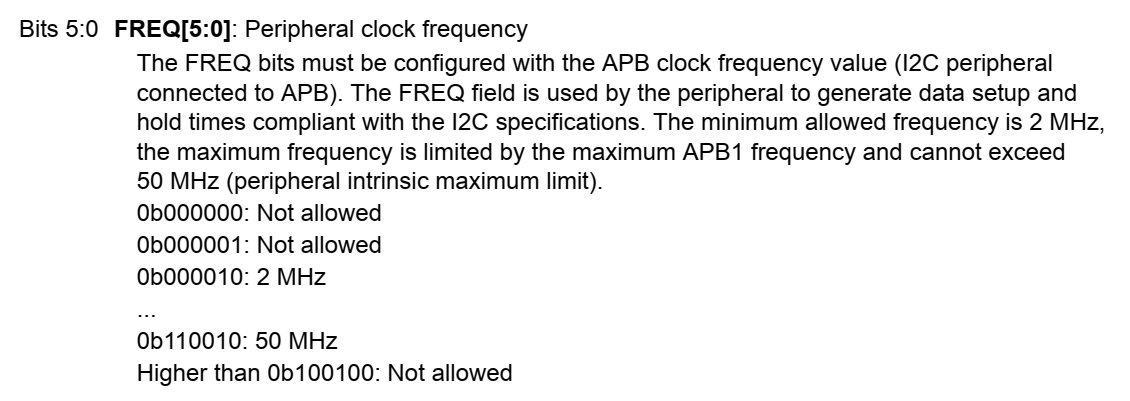
1. **配置时钟和波特率**

配置时钟，用于获得通信所需的波特率。需要设置I2C\_CR2寄存器、I2C\_CCR寄存器和I2C\_TRISE寄存器。

首先看需要设置I2C\_CR2。

**I2C\_CR2：**

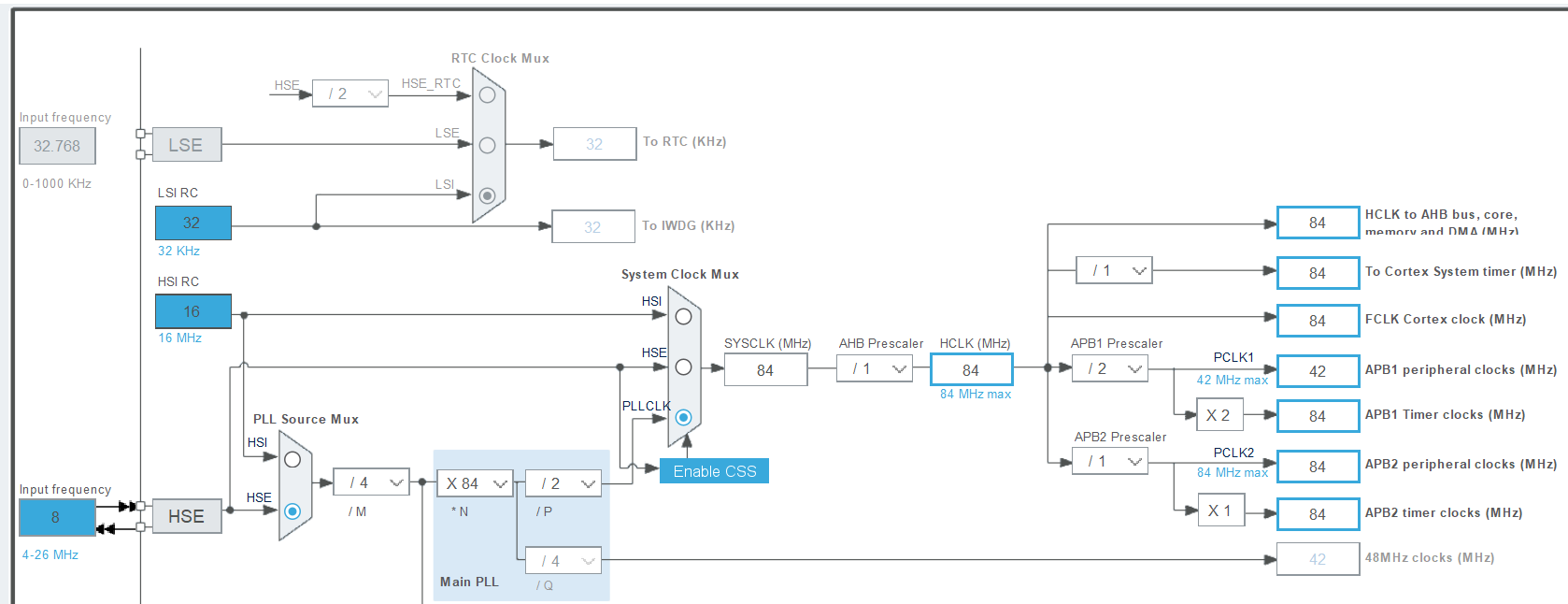




I2C\_CR2寄存器的FFREQ区域填入APB总线的时钟频率，单位为MHz。

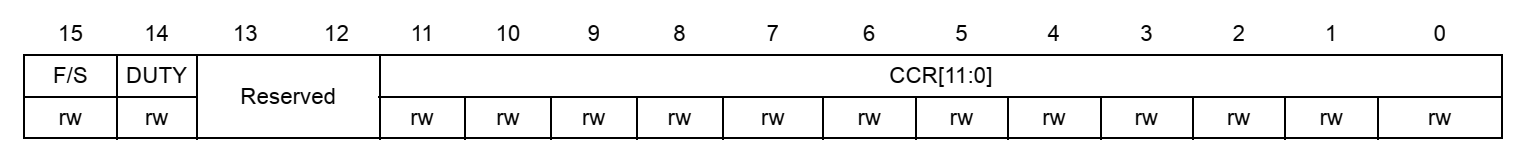
用于内部时钟周期TPLCK1=1/(FREQ MHz)

查看时钟配置界面，APB1外设时钟频率为42MHz，因此FREQ区域需要填入42。



再看I2C\_CCR寄存器的功能

**I2C\_CCR：**



F/S ： 模式选择，0为标准模式（Fm）,1为快速模式（Fm）

DUTY：0：50%占空比，1：16低/9高

CCR：时钟控制

标准模式下：

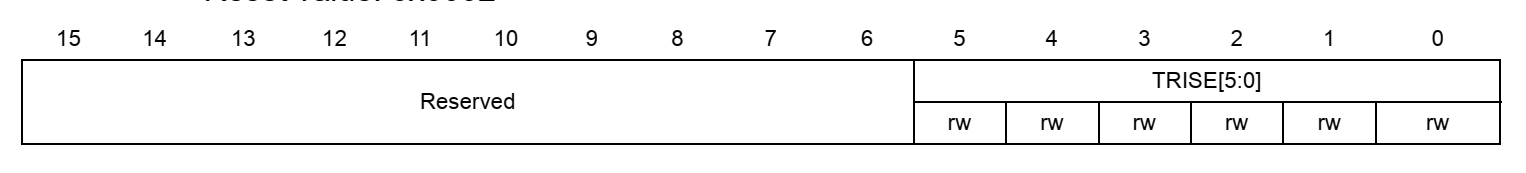
Thigh= CCR\* TPLCK1

Tlow= CCR\* TPLCK1

所以如果要把波特率配置为100kHz，CCR应该等于42\*5

I2C\_TRISE寄存器寄存器用于设置信号允许的最大上升沿的时间。由于信号为漏极开路上拉模式，因此总线上的电容可能是上升沿变慢。

**I2C\_TRISE**



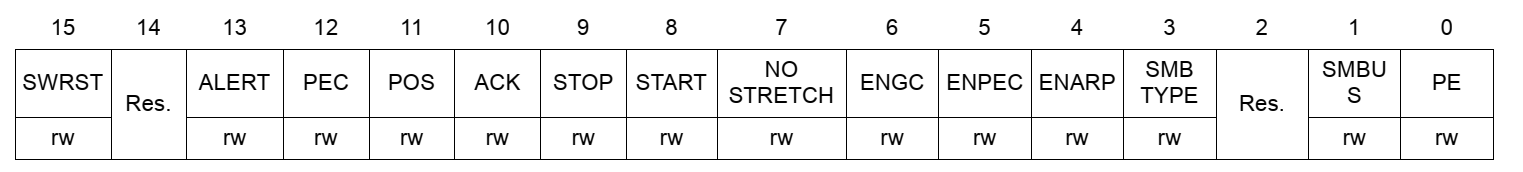
上升时间为：（TRISE-1）\* TPLCK1

最大上升时间为1000ns。

1. **使能I2C模块**

I2C模块的使能，在I2C\_CR1中设置：

I2C\_CR1



SWRST：写1使I2C模块进入重置状态

写0恢复I2C模块的运行

PE：0：外设停止运行，1：外设使能。

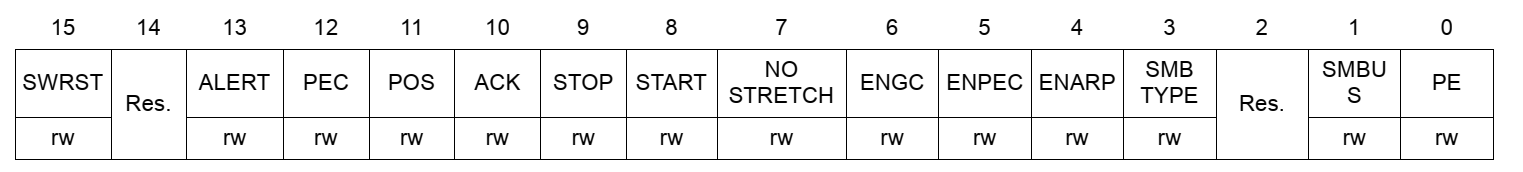
以上时钟和波特率的配置必须在外设使能前完成，即在PE=0时配置时钟和波特率。

1. **传输数据**

* **起始位和从机地址**

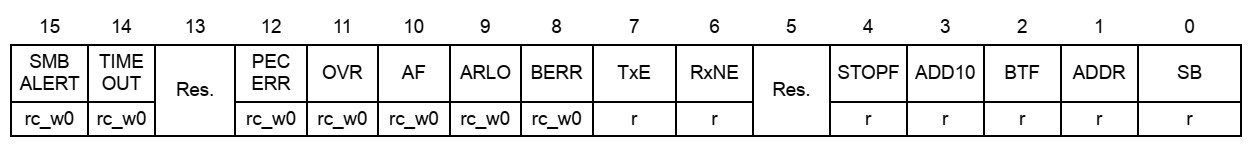
将START置位，主机会在传输线上会发送一个起始位。

I2C\_CR1



起始位发送完成后，SB位会置位

I2C\_SR1

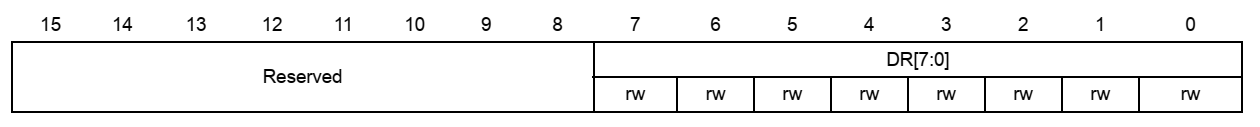


SB：0：未产生起始信号

1：已经产生起始信号

读I2C\_SR1，然后将地址写入到DR寄存器中，可以清除SB位。因此需要将接着将从机地址和读写控制写入大DR寄存器。

I2C\_DR

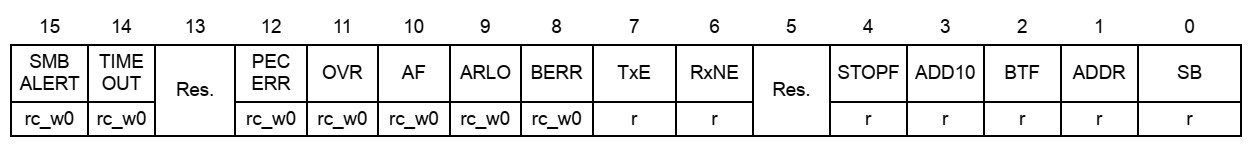


写地址时注意，1-7位为从机地址，0位为读写位。

地址读写位写入到DR寄存器后，开始发送地址。

收到从机的回应信号ACK后，I2C\_SR1的ADDR位置1。

I2C\_SR1

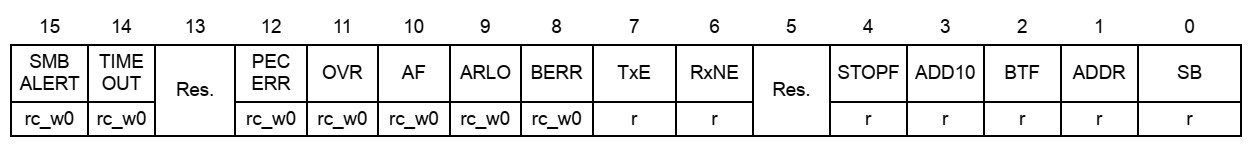


读取I2C\_SR1寄存器和I2C\_SR2寄存器会自动清除I2C\_SR1的ADDR位。

* **主机写数据**

如果发送的是写指令，I2C\_SR1的TxE位置1。

I2C\_SR1



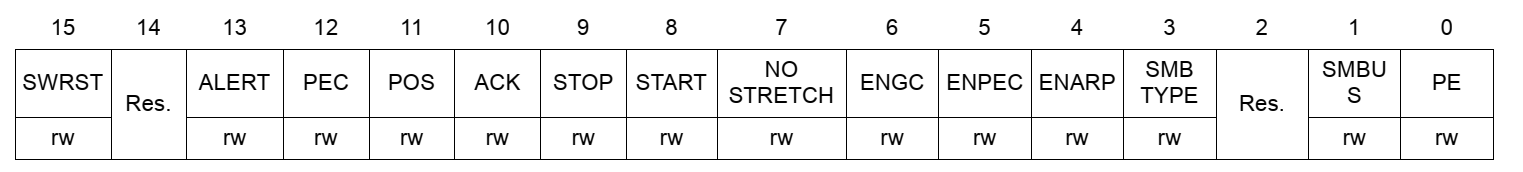
向I2C\_DR寄存器里写入要发送的数据，可以清除TxE位。同时数据开始在总线上传输

收到从机的回应后，I2C\_SR1的TxE位再次置1。向I2C\_DR寄存器继续里写入要发送的数据，可以清除TxE位。

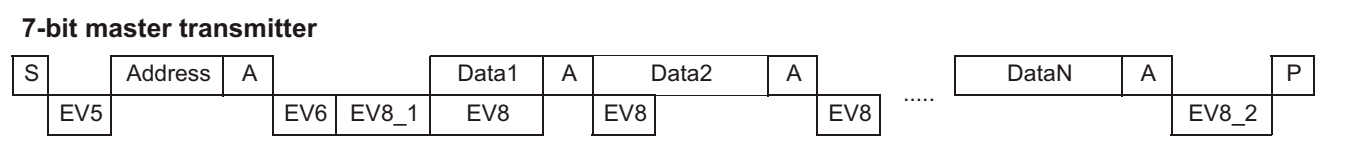
* **停止位**

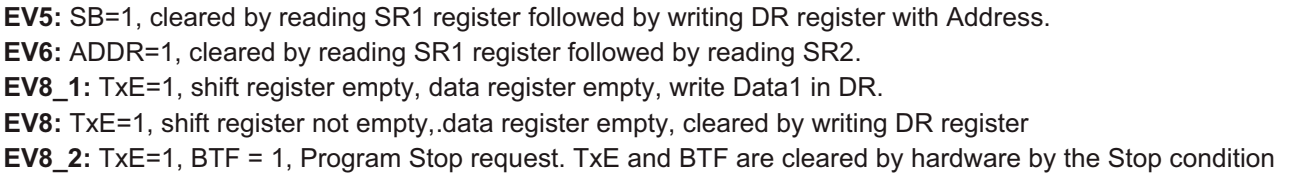
将最后一个数据写入I2C\_DR后，等待TxE位再次置1。然后将I2C\_CR1的STOP位置1，可以在发送结束位，结束此次通信。

I2C\_CR1



* **主机写数据的总线时序总结**

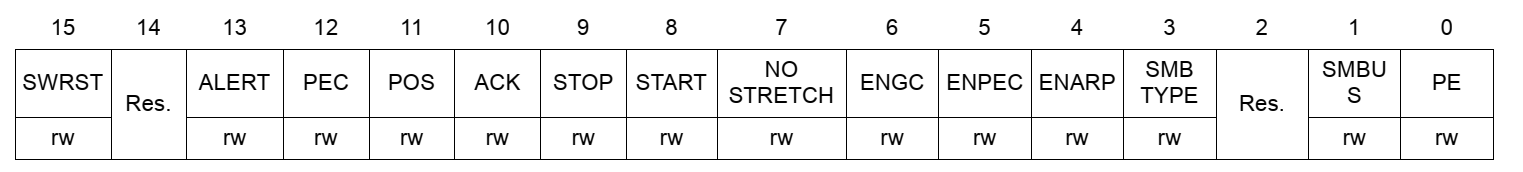




* **主机读数据**

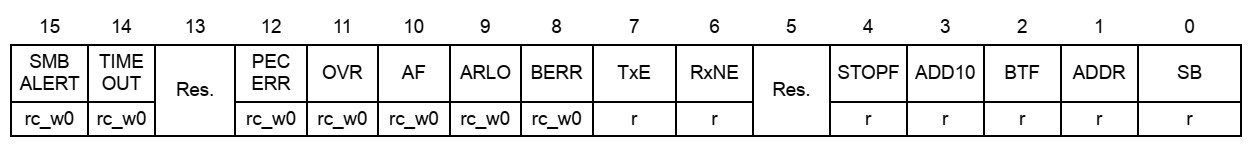
如果是读指令，特别是读多个数据，需要将I2C\_CR1的ACK位置1，使主机收到从机的数据后能够自动回应。

I2C\_CR1



发送完地址和读指令，并收到从机的回应后，I2C\_SR1的ADDR位置1。通过读I2C\_SR1寄存器和I2C\_SR2寄存器清除I2C\_SR1的ADDR位后，主机发送时钟，从从机接收数据。如果收到从机的数据，I2C\_SR1的RxNE位置1。

I2C\_SR1



RxNE：

0：数据寄存器为空

1：数据寄存器非空

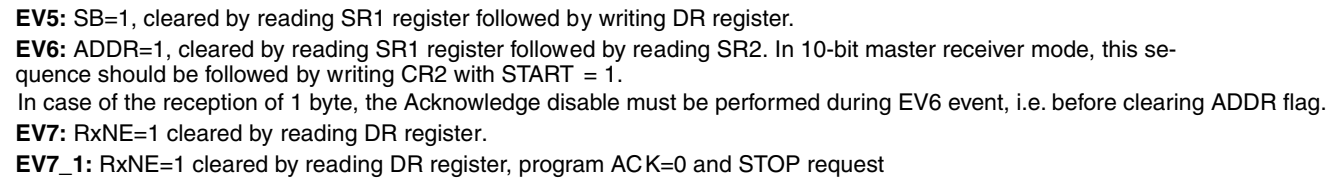
如果I2C\_CR1的ACK位为1，主机则会自动回应从机。

读DR寄存器可以获取数据并清除RxNE标志，并继续接受下一个数据。

如果要结束通信，要在读完倒数第二个数据后，将I2C\_CR1的ACK位置0，将I2C\_CR1的STOP位置1。

* **主机读数据的总线时序总结**



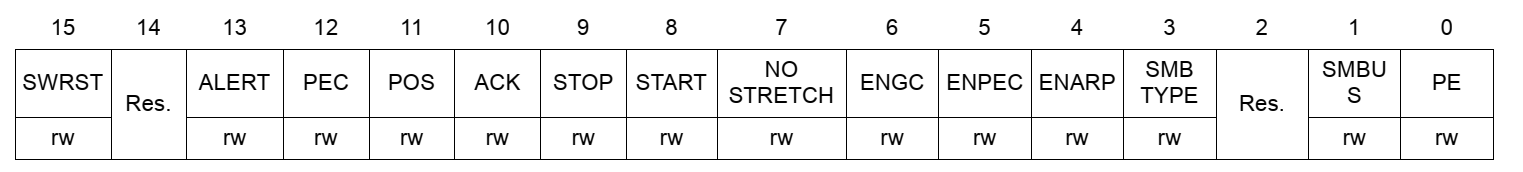


**I2C从机模式的使用：**

配置时钟、波特率与使能等操作跟主机模式相同

I2C默认为从机模式，如果不发送起始位，则总是以从机模式等待数据到来。需要将I2C\_CR1的ACK位置1，以便收到数据后能够自动回应。

I2C\_CR1



* **从机地址：**

从机检测到总线的起始位后，开始接收地址，如果接收到的地址与OAR1寄存器的地址一致的话，则发送ACK信号。

**I2C\_OAR1**



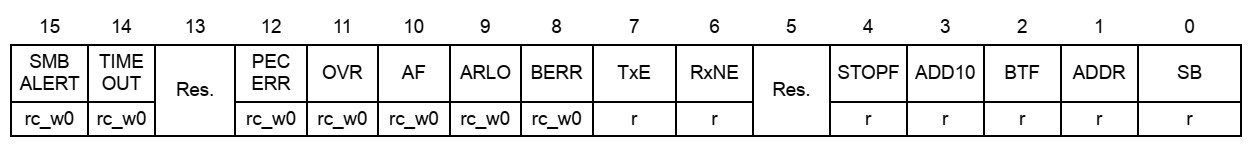
ADDMODE: 0： 7-bit 地址， 1：10-bit地址

ADD[7:1] 从机地址

ADD0：在7-bit地址模式下没有作用。

然后I2C\_SR1的ADDR位置1。

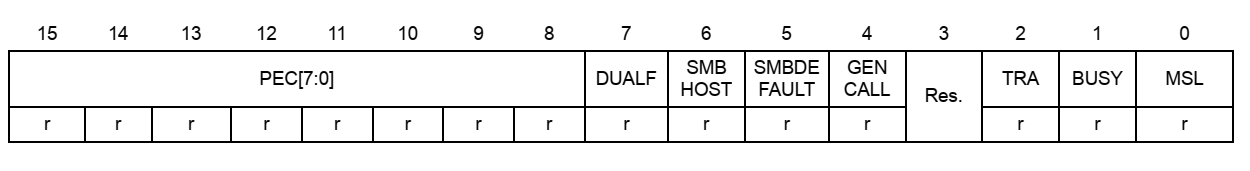
I2C\_SR1



连续读取I2C\_SR1和I2C\_SR2寄存器可以清除I2C\_SR1的ADDR位

收到地址位后，需要判断是读指令还是写指令。I2C\_SR2的TRA位指示是读指令还是写指令

I2C\_SR2



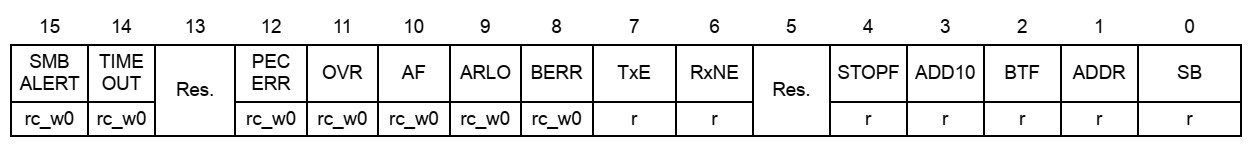
0：写指令，如要从机（本机）接收数据

1：读指令，需要从机（本机）发送数据

* **从机发送数据**

如果接收到的是**读指令**，从机（本机）需要发送数据，I2C\_SR1的TxE位会被置1.

I2C\_SR1



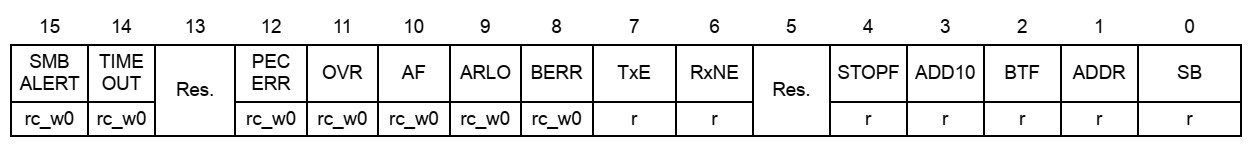
然后 从机（本机） 会把SCL拉低，暂停传送，直到将要发送的数据写入到DR寄存器中，I2C\_SR1的TxE位会被清除，从机会继续传送一个字节。

主机收到数据后，如果产生了ACK信号，那么从机（本机）收到ACK信号后，I2C\_SR1的TxE位会被置1，等待程序向DR寄存器中填入数据后继续发送一个字节。

* **停止位**

通信结束时，主机不发送回应信号并产生一个停止位，从机（本机）接收到停止位后，I2C\_SR1的AF位会被置1.

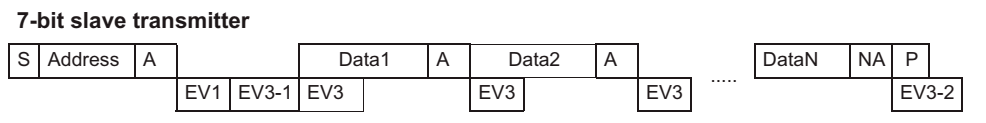
I2C\_SR1

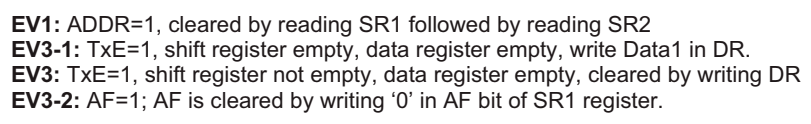


AF：0 正常收到回应位，1：没有收到回应位。

向AF位写0会清除AF位。

* **从机发数据的总线时序总结**

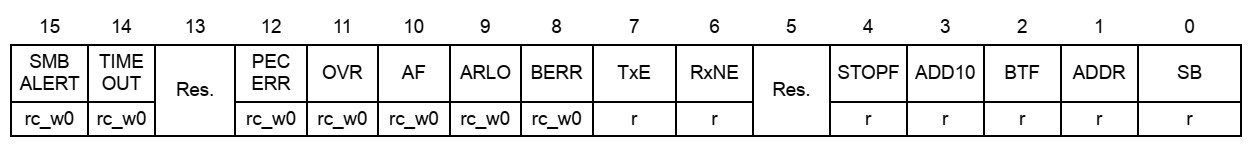




* **从机接收数据**

如果接收到的是**写指令**，从机（本机）需要接收数据。从机收到地址位后，会自动产生ACK信号，然后主机会继续发送一个字节的数据，主机发送完成后，从机（本机）会将接收到的数据保存在DR寄存器中，自动产生一个ACK信号，然后I2C\_SR1的RxNE位会被置1.

I2C\_SR1

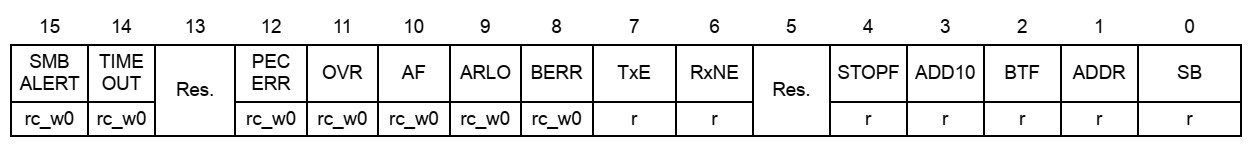


读DR寄存器取出数据，会自动清除RxNE位，并接收下一个数据。

* **结束通信**

通信结束收到主机发送的停止位后，I2C\_SR1的STOPF位会置1。

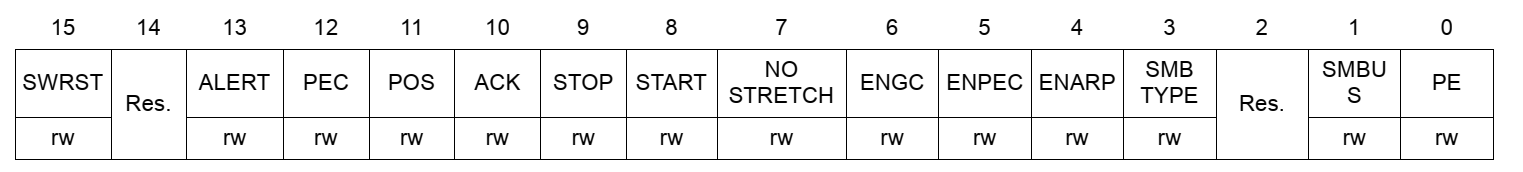
I2C\_SR1



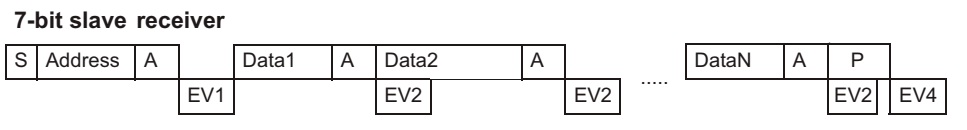
STOPF：0：没有收到停止位，1：收到停止位

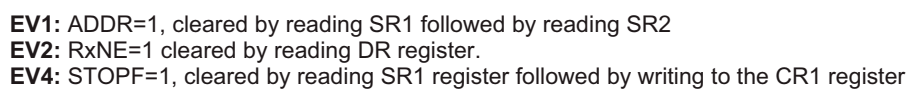
读I2C\_SR1寄存器，然后写I2C\_CR1寄存器，如将PE写为1，会清除STOPF位。

I2C\_CR1



* **从机接收数据的总线时序总结**





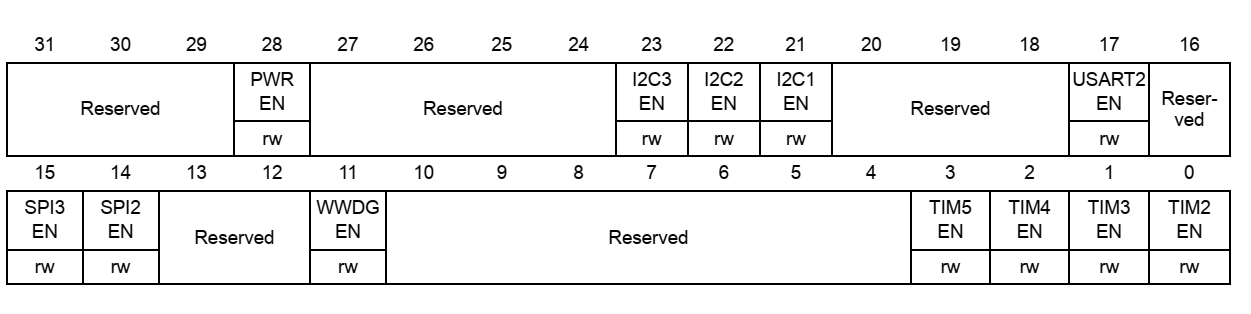
## STM32F401的I2C模块的配置

为了方便学习I2C模块的主、从机应用，我们将I2C1模块设置为主机，将I2C3模块设置为从机，以此为例，说明STM32F401的I2C模块的使用过程。

**I2C模块时钟的使能：**

RCC\_APB1ENR寄存器控制了APB1总线上各个外设的时钟，要使用APB1总线上的外设，必须把开启外设的时钟。

**RCC\_APB1ENR**



STM32F401有三个I2C模块，分别由21位、22位、23位控制。HAL库提供了

\_\_HAL\_RCC\_I2C1\_CLK\_ENABLE();

\_\_HAL\_RCC\_I2C2\_CLK\_ENABLE();

\_\_HAL\_RCC\_I2C3\_CLK\_ENABLE();

三个函数来使能三个I2C模块。其实质就是操作RCC\_APB1ENR寄存器。

#define \_\_HAL\_RCC\_I2C1\_CLK\_ENABLE() do { \

\_\_IO uint32\_t tmpreg = 0x00U; \

SET\_BIT(RCC->APB1ENR, RCC\_APB1ENR\_I2C1EN);\

/\* Delay after an RCC peripheral clock enabling \*/ \

tmpreg = READ\_BIT(RCC->APB1ENR, RCC\_APB1ENR\_I2C1EN);\

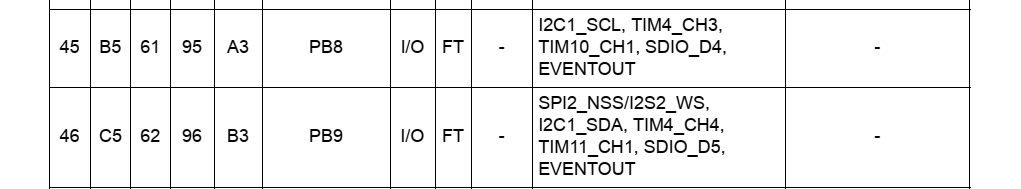
UNUSED(tmpreg); \

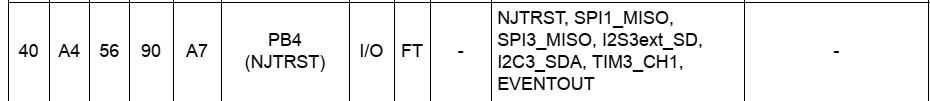
} while(0U)

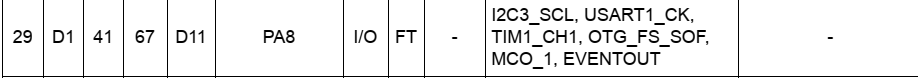
**GPIO的复用设置：**

I2C模块的CLK和SDA信号，是通过某个GPIO的引脚与外界相连的，因此在使用I2C外设之前，还需要将特定的引脚与I2C模块的CLK和SDA信号连接起来。

如果要将某个引脚与I2C模块相连，需要先阅读器件手册，查看哪个引脚与哪个I2C模块相连。







可以看出I2C1模块的SCL信号可以与PB8引脚相连，I2C1模块的SDA信号可以与PB9引脚相连。

可以看出I2C3模块的SCL信号可以与PA8引脚相连，I2C3模块的SDA信号可以与PB4引脚相连。

因此，如果使I2C1模块与I2C3模块通信，需要在实验板上将PB8引脚与PA8引脚相连，将PB9引脚与PB4引脚相连。除此之外，还要将PA8和PB4引脚连接到上拉电阻上，如图所示



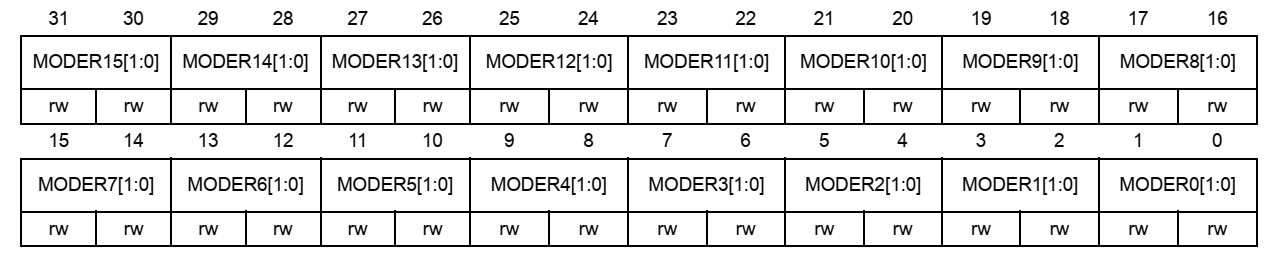
确认了引脚之后，并连接好导线后，还需要通过设置，将这些引脚与对应的I2C模块连接起来。

首先需要使能用到的端口的时钟，即GPIOB和GPIOA的时钟。

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOB\_CLK\_ENABLE();

然后设置这两个端口的MODER寄存器，将引脚设置为复用模式。回顾GPIO的MODER寄存器



每两位控制一个引脚，MODER0[1:0]两位控制Px0引脚，MODER1[1:0]两位控制Px1引脚。

* 这两位为00代表为数字输入模式
* 这两位为01代表为数字输出模式
* 这两位为10代表外设复用模式
* 这两位为11代表模拟信号模式

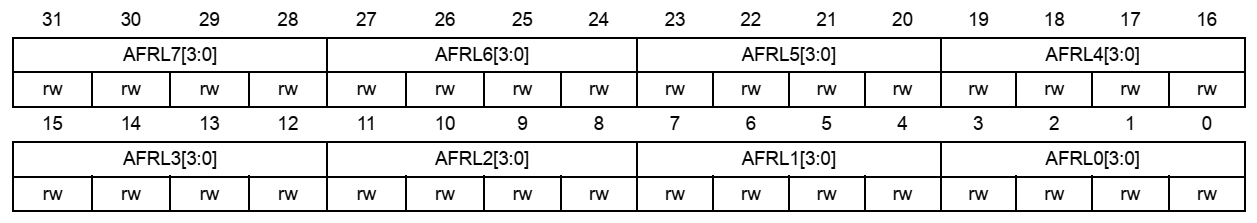
如果我们要使用I2C1模块和I2C3模块，就要把GPIOB的MODER4、MODER8、MODER9设置为10，其他位不变。

要把GPIOA的MODER8设置为10，其他位不变。

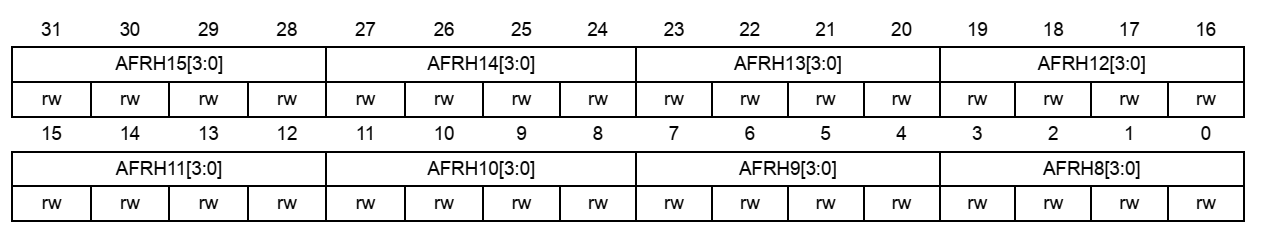
将引脚设置为外设复用模式后，需要设置GPIOx\_AFRL和GPIOx\_AFRH寄存器，告知MCU该引脚被哪个外设复用。

GPIOx\_AFRL是32位寄存器，管理端口中0-7引脚的外设复用情况。GPIOx\_AFRH是32位寄存器，管理端口中8-15引脚的外设复用情况。

GPIOx\_AFRL每四位管理一个引脚的外设复用情况。例如GPIOB的GPIOx\_AFRL的AFRL0这四位，管理PB0的外设复用情况，GPIOB的GPIOx\_AFRL的AFRL4这四位，管理PB4的外设复用情况。

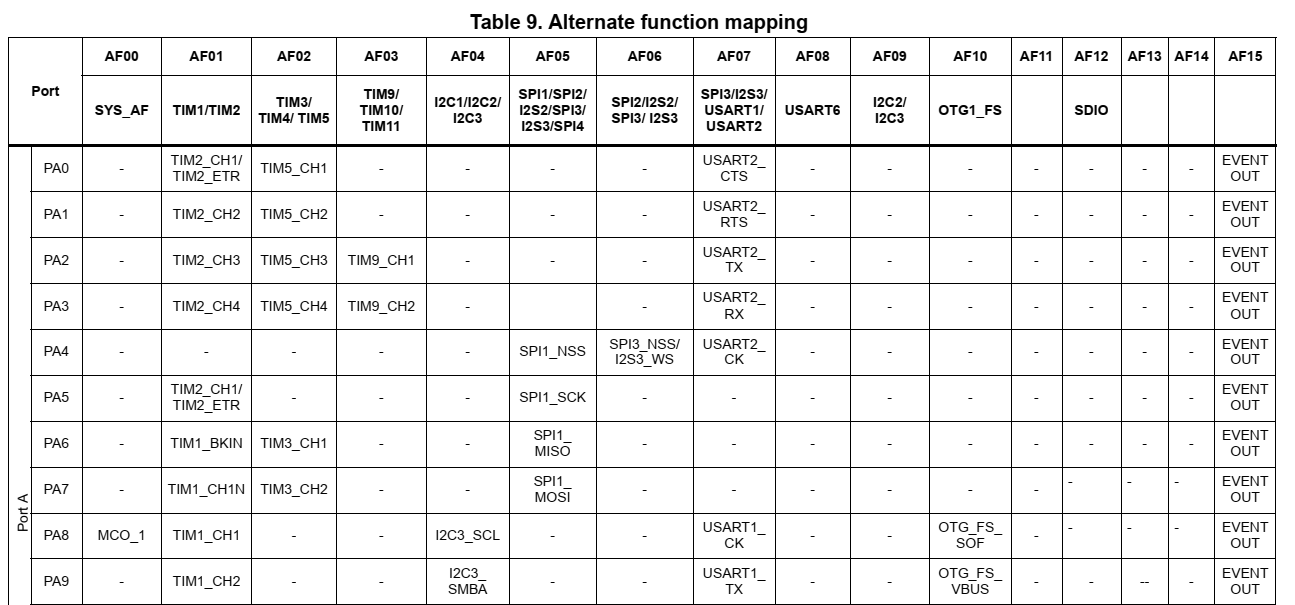


GPIOx\_AFRH也是每四位管理一个引脚的外设复用情况。



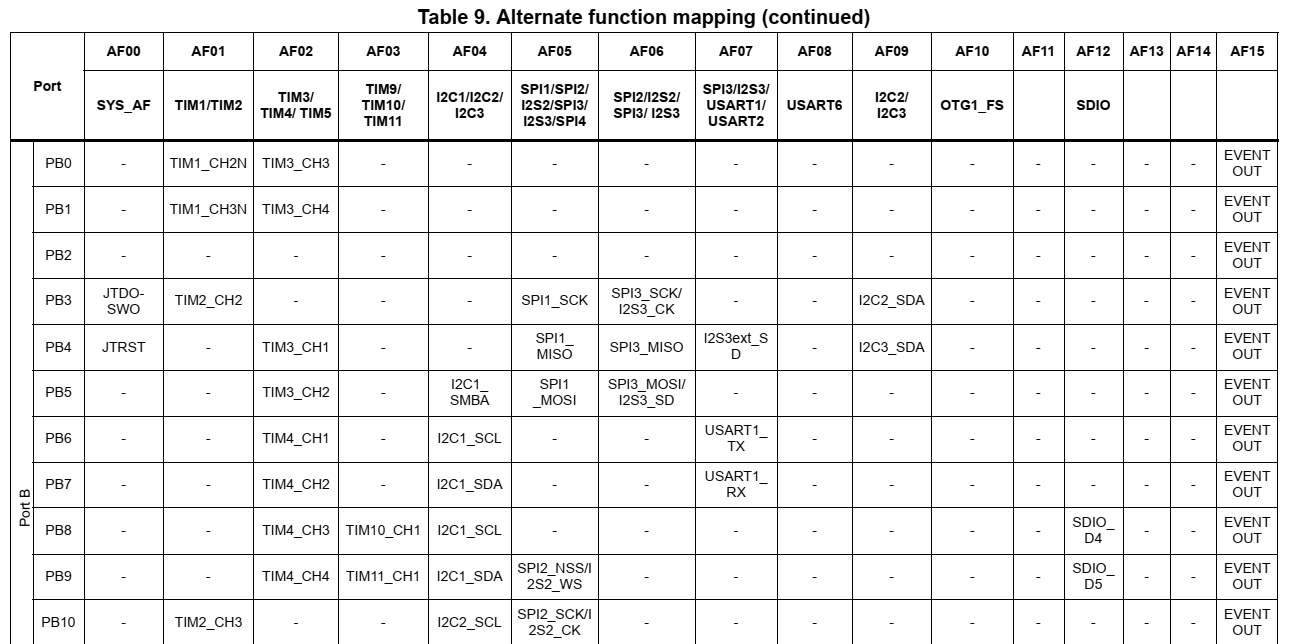
例如GPIOB的GPIOx\_AFRH的AFRH8这四位，管理PB8的外设复用情况，GPIOB的GPIOx\_AFRH的AFRH9这四位，管理PB9的外设复用情况。

AFRLn(n=0-7)或者AFRHn(n=8-15)这四位，可以组成16个数，即0、1、2、3… 15。每一个数对应一个外设。这个GPIO与外设的对应表格，在器件手册中（Table 9）。



这样，我们可以找到，如果要将PA8复用为I2C3的SCL信号，需要把GPIOA的GPIOx\_AFRH寄存器的AFRH8这四位写入4；

同理，寻找PB4、PB8、PB9与外设的对应情况。



同理可知，要将PB4复用为I2C3的SDA信号，可以将GPIOB的GPIOx\_AFRL寄存器的AFRL4这四位写入9；

如果要将PB8复用为I2C1的SCL信号，需要把GPIOB的GPIOx\_AFRH寄存器的AFRH8这四位写入4；

如果要将PB9复用为I2C1的SDA信号，需要把GPIOB的GPIOx\_AFRH寄存器的AFRH9这四位写入4；

这样设置完成后，I2C模块的SCL和SDA信号，就与MCU的引脚连在一起了。

## STM32F401的I2C模块的使用(主从通信)

**通过直接操作寄存器实现主从通信**

1. **配置GPIO**

在进入主循环之前，先配置GPIO

在 /\* USER CODE BEGIN 2 \*/之后，先开启GPIOB的时钟。GPIOA的时钟已经在MX\_GPIO\_Init();函数内开启过了。

\_\_HAL\_RCC\_GPIOB\_CLK\_ENABLE();

然后设置PB8、PB9、PB4为复用模式：

tmp=GPIOB->MODER;

tmp&=~(0x03<<(8\*2));

tmp|=(0x02<<(8\*2));

tmp&=~(0x03<<(9\*2));

tmp|=(0x02<<(9\*2));

tmp&=~(0x03<<(4\*2));

tmp|=(0x02<<(4\*2));

GPIOB->MODER=tmp;

设置的PB8、PB9、PB4输出模式为开漏输出

tmp=GPIOB->OTYPER;

tmp&=~(0x01<<8);

tmp|=0x01<<8;

tmp&=~(0x01<<9);

tmp|=0x01<<9;

tmp&=~(0x01<<4);

tmp|=0x01<<4;

GPIOB->OTYPER=tmp;

设置PA8为复用模式

tmp=GPIOA->MODER;

tmp&=~(0x03<<(8\*2));

tmp|=0x02<<(8\*2);

GPIOA->MODER=tmp;

设置PA8的输出模式为开漏输出

tmp=GPIOA->OTYPER;

tmp&=~(0x01<<8);

tmp|=0x01<<8;

GPIOA->OTYPER=tmp;

设置GPIOB的GPIOx\_AFRH寄存器的AFRH8和AFRH9为4.

tmp=GPIOB->AFR[1];

tmp&=~(0xf<<((8-8)\*4));

tmp|=(0x4<<((8-8)\*4));

tmp&=~(0xf<<((9-8)\*4));

tmp|=(0x4<<((9-8)\*4));

GPIOB->AFR[1]=tmp;

设置GPIOB的GPIOx\_AFRL寄存器的AFRL4这四位为9

tmp=GPIOB->AFR[0];

tmp&=~(0xf<<((4)\*4));

tmp|=(0x9<<((4)\*4));

GPIOB->AFR[0]=tmp;

设置GPIOA的GPIOx\_AFRH寄存器的AFRH8为4

tmp=GPIOA->AFR[1];

tmp&=~(0xf<<((8-8)\*4));

tmp|=(0x4<<((8-8)\*4));

GPIOA->AFR[1]=tmp;

1. **I2C模块寄存器的访问**

在头文件stm32f401xc.h文件中，定一个了一个结构体类型I2C\_TypeDef，用于方便的访问I2C模块的各个寄存器

typedef struct

{

\_\_IO uint32\_t CR1; /\*!< I2C Control register 1, Address offset: 0x00 \*/

\_\_IO uint32\_t CR2; /\*!< I2C Control register 2, Address offset: 0x04 \*/

\_\_IO uint32\_t OAR1; /\*!< I2C Own address register 1, Address offset: 0x08 \*/

\_\_IO uint32\_t OAR2; /\*!< I2C Own address register 2, Address offset: 0x0C \*/

\_\_IO uint32\_t DR; /\*!< I2C Data register, Address offset: 0x10 \*/

\_\_IO uint32\_t SR1; /\*!< I2C Status register 1, Address offset: 0x14 \*/

\_\_IO uint32\_t SR2; /\*!< I2C Status register 2, Address offset: 0x18 \*/

\_\_IO uint32\_t CCR; /\*!< I2C Clock control register, Address offset: 0x1C \*/

\_\_IO uint32\_t TRISE; /\*!< I2C TRISE register, Address offset: 0x20 \*/

\_\_IO uint32\_t FLTR; /\*!< I2C FLTR register, Address offset: 0x24 \*/

} I2C\_TypeDef;

然后又通过宏定义了三个I2C模块的基地址

#define PERIPH\_BASE 0x40000000UL /\*!< Peripheral base address in the alias region \*/

#define APB1PERIPH\_BASE PERIPH\_BASE

#define I2C1\_BASE (APB1PERIPH\_BASE + 0x5400UL)

#define I2C2\_BASE (APB1PERIPH\_BASE + 0x5800UL)

#define I2C3\_BASE (APB1PERIPH\_BASE + 0x5C00UL)

最后通过宏定义，将I2C模块的基地址I2C1\_BASE、I2C2\_BASE、I2C3\_BASE转换为I2C\_TypeDef类型的指针，用I2C1、I2C2、I2C3代替，就可以轻松的访问每个I2C模块的寄存器了。如I2C1->CR1、I2C1->CCR等。

#define I2C1 ((I2C\_TypeDef \*) I2C1\_BASE)

#define I2C2 ((I2C\_TypeDef \*) I2C2\_BASE)

#define I2C3 ((I2C\_TypeDef \*) I2C3\_BASE)

1. **初始化主机和从机**

首先开启I2C1模块的时钟

\_\_HAL\_RCC\_I2C1\_CLK\_ENABLE();

配置I2C1模块的I2C\_CR2寄存器的FREQ区域为42，其他位为0。即TPLCK1为1/(42MHz)。配置I2C1模块的I2C\_CCR寄存器的CCR位为42\*5，Thigh= Tlow=5us，即波特率为100kHz。设置上升时间最大值为1us，然后使能I2C1模块，即设置I2C\_CR1寄存器的PE位为1。

I2C1->CR2=42;

I2C1->CCR=5\*42;

I2C1->TRISE=42;

I2C1->CR1|=0x01; //PE=1

同理，开启I2C3模块的时钟。

\_\_HAL\_RCC\_I2C3\_CLK\_ENABLE();

配置I2C3模块的I2C\_CR2寄存器的FREQ区域为42，其他位为0。即TPLCK1为1/(42MHz)。配置I2C3模块的I2C\_CCR寄存器的CCR位为42\*5，Thigh= Tlow=5us，即波特率为100kHz。设置上升时间最大值为1us。设置I2C3模块的从机地址为0x1D。然后使能I2C3模块，即设置I2C\_CR1寄存器的PE位为1。然后启动自动回应，即设置I2C\_CR1寄存器的ACK位为1

I2C3->CR2=42;

I2C3->CCR=5\*42;

I2C3->TRISE=42;

I2C3->OAR1=0x1D<<1;

I2C3->CR1|=0x01; //PE=1

I2C3->CR1|=(0x01<<10);//ACK=1;

1. **主机发数据，从机收数据**

首先明确I2C1模块为主机，I2C3模块为从机。开启从机I2C3的自动回应。

//////////////////

I2C3->CR1|=(1<<10);//ACK=1

//////////////////

主机I2C1发送起始位，等待起始位发送完成后，接着发送从机地址和写指令，等待从机I2C3回应后，清除I2C1的I2C\_SR1寄存器的ADDR位。

I2C1->CR1|=(1<<8); // START=1

**while**((I2C1->SR1 &0x01)==0); //wait for SB==1

I2C1->DR=(0x1D<<1)+0;// write to 0x1D

**while**((I2C1->SR1 &0x02)==0); //wait for ADDR==1

tmp=I2C1->SR2;

从机I2C3收到地址，并自动回应后，从机I2C3的I2C\_SR1寄存器的ADDR位置1，需要清除ADDR位。连续读取I2C\_SR1寄存器和I2C\_SR2寄存器可以清除ADDR位。

//////////////////////

**while**((I2C3->SR1 &0x02)==0); //wait for ADDR==1

tmp=I2C3->SR2;

/////////////////////

由于主机I2C1发送的是写指令，因此收到从机的回应后，主机I2C1的I2C\_SR1寄存器的TxE位会置1，向主机I2C1的DR寄存器中写入要发送的数据，主机I2C1的I2C\_SR1寄存器的TxE位清零，并开始发送数据。

**while**((I2C1->SR1 & (1<<7))==0); //wait for TxE==1

I2C1->DR=0x32;

从机I2C3收到数据后自动回应，并且从机I2C3的I2C\_SR1寄存器的RxNE位置1。读从机I2C3的DR寄存器，可以将数据取出，并使RxNE位清零

/////////////////////

**while**((I2C3->SR1 & (1<<6))==0); //wait for RxNE==1

b=I2C3->DR;

/////////////////////

主机I2C1收到从机I2C3的回应信号后，TxE位会置1，如果要继续发送数据，则继续向DR寄存器中写入数据。

如果要停止通信，则将主机I2C1的I2C\_CR1寄存器的STOP位置1，发送停止位。

I2C1->CR1|=(0x01<<9); // STOP=1

从机I2C3收到停止位后，从机I2C3的I2C\_SR1寄存器STOPF位置1。读从机I2C3的I2C\_SR1寄存器并写I2C\_CR1寄存器清除STOPF位。

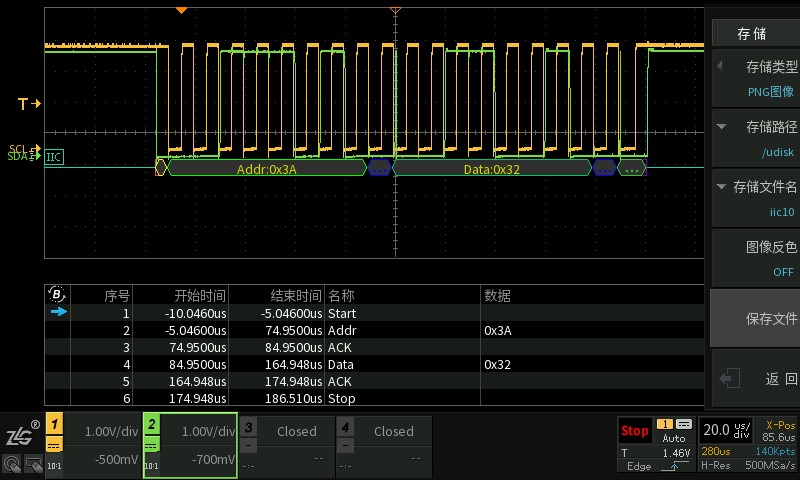
///////////////////////////

**while**((I2C3->SR1 & (1<<4))==0); //wait for STOPF==1

I2C3->CR1|=0x01; //PE=1 clear STOPF

/////////////////////////////////

用示波器捕获总线上的信号，如图所示。可见总线上传输的数据，与预期的一致。



1. **主机读数据，从机发数据**

首先开启从机I2C3的自动回应，使从机I2C3能够回应主机I2C1的地址位

///////////////////////

I2C3->CR1|=(1<<10);//ACK=1

///////////////////////

然后主机I2C1发送起始位，等起始位发送完成后，发送地址位和读指令。主机I2C1收到从机I2C3的自动回应后，ADDR位置1，读主机I2C1的I2C\_SR1寄存器和I2C\_SR2寄存器可以清除ADDR位。如果主机I2C1只读一个字节的数据，收到从机I2C3的地址回应信号后，应该关闭主机I2C1的自动回应，并设置主机I2C1的I2C\_CR1寄存器的STOP位为1，即在传输完下一个字节后，发送停止位。

I2C1->CR1|=(1<<8); // START=1

**while**((I2C1->SR1 &0x01)==0); //wait for SB==1

I2C1->DR=(0x1D<<1)+1;// read from 0x1D

**while**((I2C1->SR1 &0x02)==0); //wait for ADDR==1

tmp=I2C1->SR2;

I2C1->CR1|=(0x01<<9); // STOP=1

I2C1->CR1&=~(1<<10); //ACK=0

从机I2C3收到主机I2C1的地址位后，地址如果与从机I2C3的地址匹配，则自动产生回应信号。产生回应信号后，从机I2C3的I2C\_SR1寄存器的ADDR位置1。连续读取读从机I2C3的I2C\_SR1寄存器和I2C\_SR2寄存器可以清除ADDR位。

如果从机I2C3收到的是读指令，那么从机I2C3的I2C\_SR1寄存器的TxE位会置1，时钟被拉低，等待从机I2C3向DR寄存器中写入数据。从机I2C3向DR寄存器中写入数据0x45后，继续数据传输，TxE位清零。

//////////////////////

**while**((I2C3->SR1 &0x02)==0); //wait for ADDR==1

tmp=I2C3->SR2;

**while**((I2C3->SR1 & (1<<7))==0); //wait for TxE==1

I2C3->DR=0x45;

/////////////////////

主机I2C1收到从机I2C3传来的数据后，主机I2C1的I2C\_SR1寄存器的RxNE位置1，读主机I2C1的DR寄存器，可以清除RxNE位。

**while**((I2C1->SR1 & (1<<6))==0); //wait for RxNE==1

b=I2C1->DR;

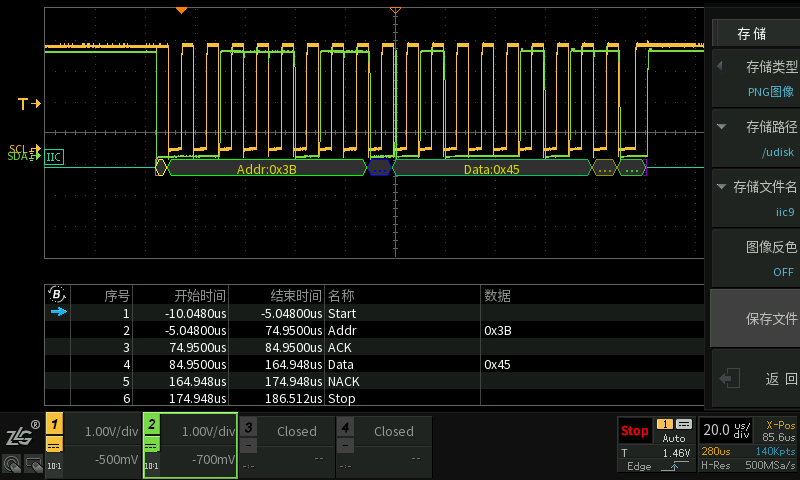
由于前面关闭了主机I2C1的自动回应，并在传输一个字节后产生停止位，因此主机I2C1接收完数据后，产生NACK信号，并产生停止位，结束数据传输。

如果从机I2C3收到的是NACK信号，那么从机I2C3的I2C\_SR1寄存器的AF位会置1，把AF位写0会清除AF位。

**while**((I2C3->SR1 & (1<<10))==0); //wait for AF==1

I2C3->SR1&=~(1<<10); //AF=0 clear AF

用示波器捕获总线上的信号，如图所示。可见总线上传输的数据，与预期的一致。



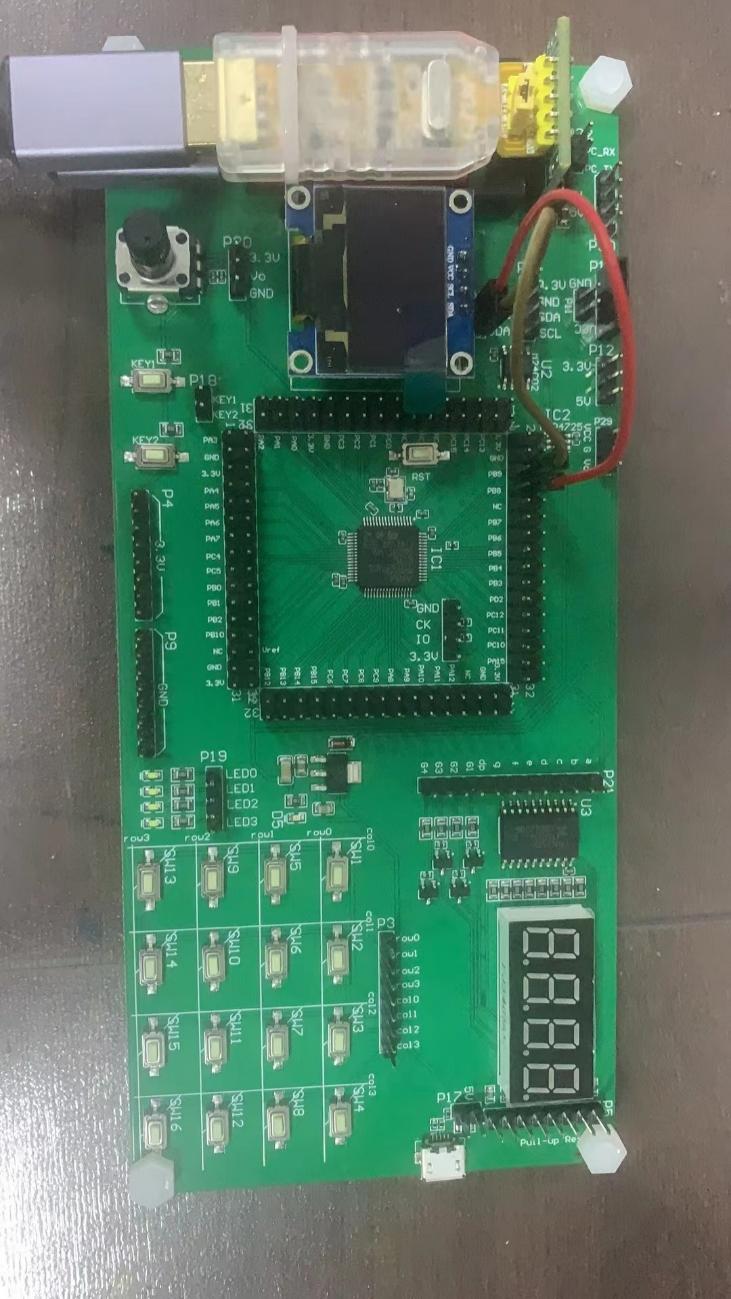
## STM32F401的I2C模块的应用：读写EEPROM(寄存器方式)

**通过直接操作寄存器读写EEPROM**

1. **开发板上的I2C设备**

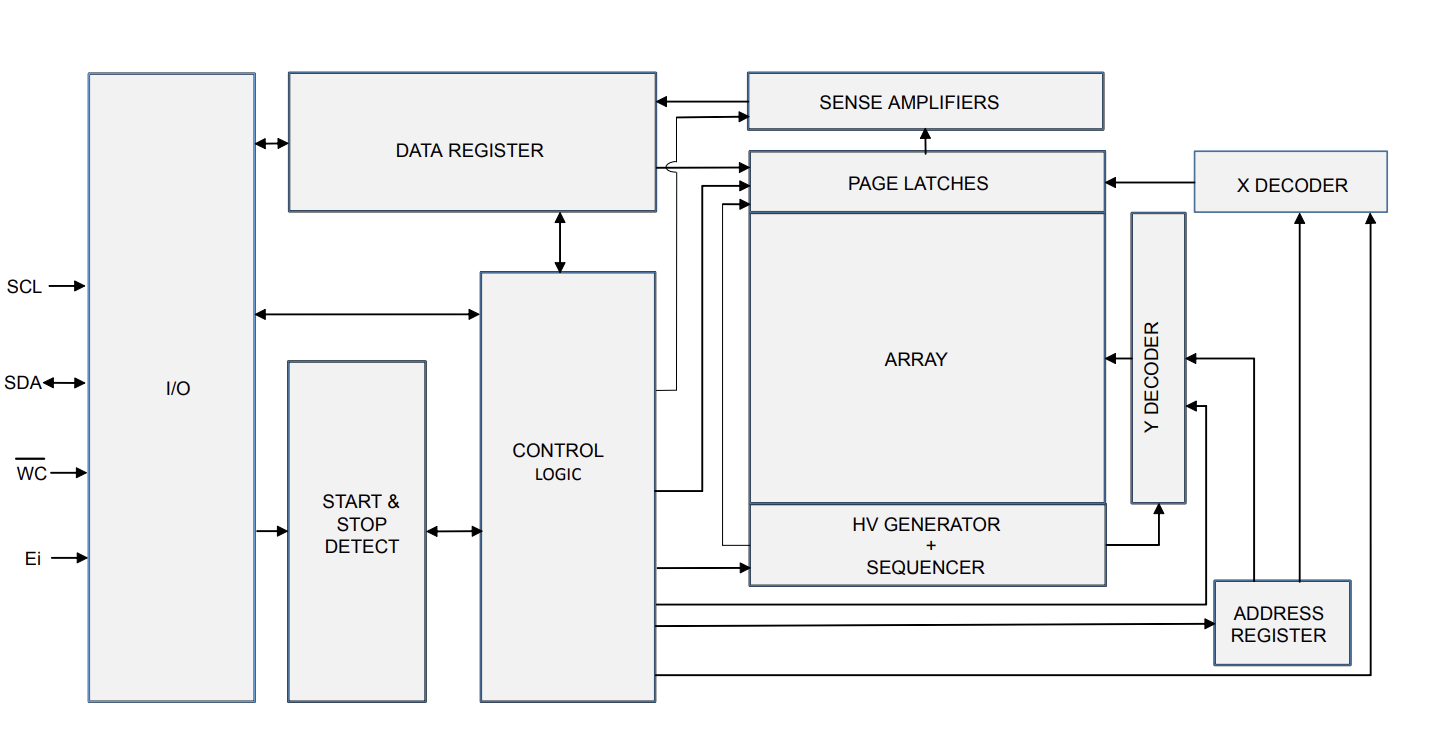


开发板上有多个I2C接口的模块，包括OLED屏和EEPROM M24C02。他们的SCL和SDA先都连在了一起。同时SDA和SCL信号有排针引出。如果需要使用I2C总线上的设备，需要将MCU的引脚连接到SCL和SDA上。本例中，使用PB8引脚作为I2C1模块的SCL引脚，PB9作为SDA引脚。因此需要将PB8连接到SCL上，将PB9连接到SDA上，如图所示。

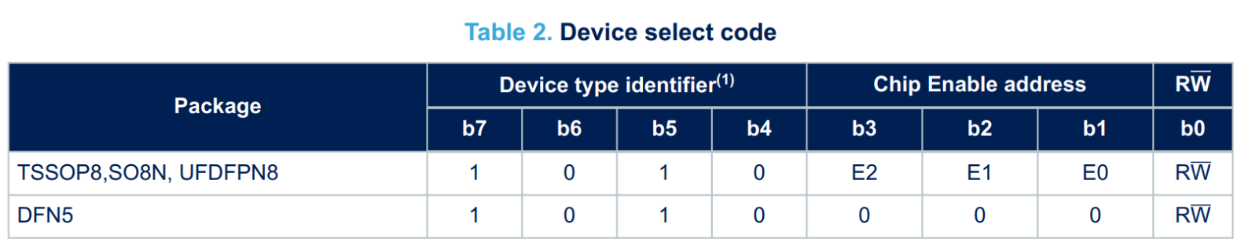


1. **EEPROM存储器M24C02**

M24C02是一个存储容量为2kbit的EEPROM。共存储256个字节。通过I2C总线与MCU通信。

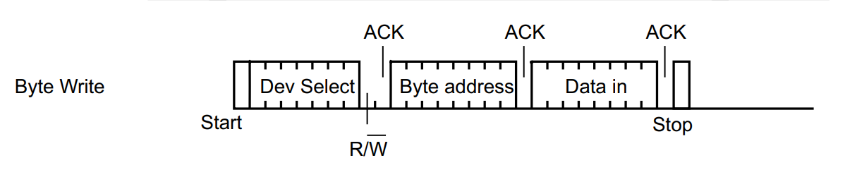


**M24C02的器件地址：**



器件地址的高四位为1010b，地址的低三位由引脚E0、E1、E2决定。在实验板上，引脚E0、E1、E2都接地，因此器件地址为0x50。

**M24C02的写时序：**

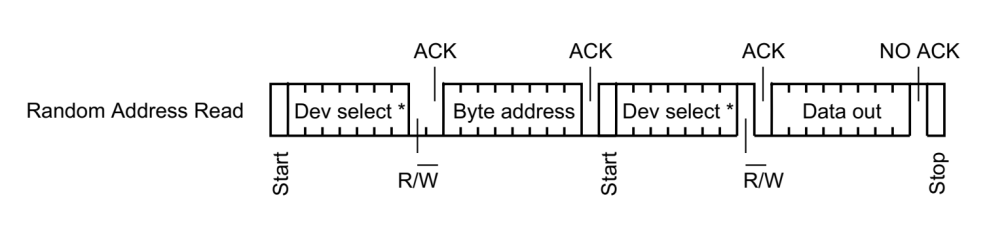


先通过发送器件地址和写指令选中EEPROM，然后发送一个字节，指明要写入的数据的地址，再把要写入的数据发送到EEPROM。

如果要连续写多个字节，可以连续写入数据，每写一个字节，器件内部地址会自动加1。因此如果连续写入，只需要设置一次地址。



**M24C02的读时序：**



先发送器件地址和写指令选中EEPROM，然后发送一个字节，指明要读取的地址。然后重新发送起始位和器件地址加读指令，接受一个字节的数据。



也可以接收多个字节的数据，每读一个字节的数据，器件内部地址会自动加1，因此如果是连续读取，只需要设置一次地址。

1. **使用I2C1读写EEPROM**

**初始化I2C1模块**

首先在/\* USER CODE BEGIN 2 \*/和/\* USER CODE END 2 \*/之间，初始化I2C1模块。

首先设置GPIO，由于I2C1模块用到的PB8（SCL）和PB9(SDA)引脚，因此需要先使能GPIOB模块。

\_\_HAL\_RCC\_GPIOB\_CLK\_ENABLE();

然后设置MODER寄存器，将PB8和PB9设置为外设复用模式

tmp=GPIOB->MODER;

tmp&=~(0x03<<(8\*2));

tmp|=(0x02<<(8\*2));

tmp&=~(0x03<<(9\*2));

tmp|=(0x02<<(9\*2));

GPIOB->MODER=tmp;

设置PB8和PB9的输出模式为漏极开路

tmp=GPIOB->OTYPER;

tmp&=~(0x01<<8);

tmp|=0x01<<8;

tmp&=~(0x01<<9);

tmp|=0x01<<9;

GPIOB->OTYPER=tmp;

设置GPIOB的GPIOx\_AFRH寄存器的AFRH8和AFRH9为4，将这两个引脚与I2C1模块连接

tmp=GPIOB->AFR[1];

tmp&=~(0xf<<((8-8)\*4));

tmp|=(0x4<<((8-8)\*4));

tmp&=~(0xf<<((9-8)\*4));

tmp|=(0x4<<((9-8)\*4));

GPIOB->AFR[1]=tmp;

设置完引脚后，初始化I2C1模块。开启I2C1模块的时钟。然后配置I2C\_CR2寄存器的FREQ区域为42，其他位为0。即TPLCK1为1/(42MHz)。配置I2C\_CCR寄存器的CCR位为42\*5，Thigh= Tlow=5us，即波特率为100kHz。设置上升时间最大值为1us，然后使能I2C1模块，即设置I2C\_CR1寄存器的PE位为1。

\_\_HAL\_RCC\_I2C1\_CLK\_ENABLE();

I2C1->CR2=42;

I2C1->CCR=5\*42;

I2C1->TRISE=42;

I2C1->CR1|=0x01; //PE=1

**I2C1模块读写EEPROM**

配置完成后，发送一个字节0x33到EEPROM的0x01地址中：

首先发送起始位，等待起始位发送完成。然后写DR寄存器发送给从机地址0x50加写指令，等待收到回应后，ADDR位置1。连续读取SR1寄存器和SR2寄存器可以清除ADDR位。由于发送的是写指令，收到从机的回应后，TxE位置1。向DR寄存器中写入要读取的EEPROM中数据的地址0x01。写入DR寄存器清除了TxE位。如果收到从机的回应，TxE位又置1，然后向DR寄存器中写入要向0x01地址写入的数据0x33。等待从机接收数据并回应，收到从机回应后TxE位又置1，发送停止位结束通信。

//write data from I2C1 to EERPOM @ address 0x01

I2C1->CR1|=(1<<8); // START=1

**while**((I2C1->SR1 &0x01)==0); //wait for SB==1

I2C1->DR=(0x50<<1)+0;// write to 0x50

**while**((I2C1->SR1 &0x02)==0); //wait for ADDR==1

tmp=I2C1->SR2;

**while**((I2C1->SR1 & (1<<7))==0); //wait for TxE==1

I2C1->DR=0x01;

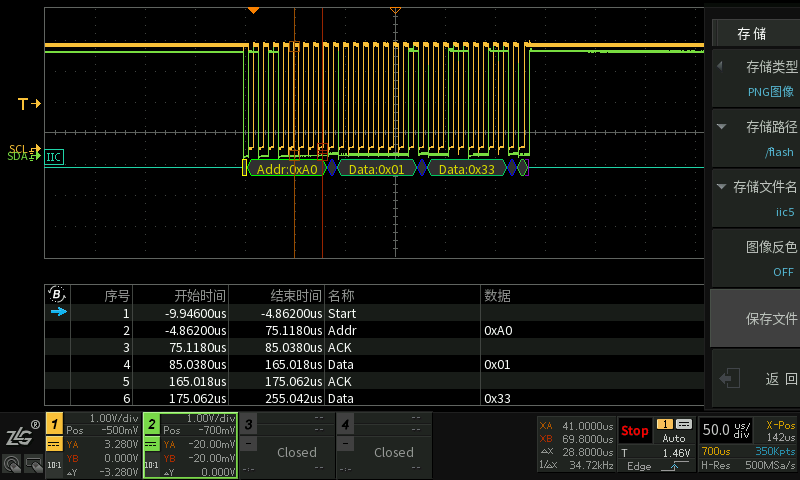
**while**((I2C1->SR1 & (1<<7))==0); //wait for TxE==1

I2C1->DR=0x33;

**while**((I2C1->SR1 & (1<<7))==0); //wait for TxE==1

I2C1->CR1|=(0x01<<9); // STOP=1

用示波器捕获总线上的信号，如图所示。可见总线上传输的数据，与预期的一致。



如果要读取EEPROM某一存储空间的数据，如读取EEPROM中地址为0x01处的数据。如果刚才写入数据成功，那么读出的数据应该也是0x33。

首先发送起始位，等待起始位发送完成。然后写DR寄存器发送给从机地址0x50加写指令，等待收到回应后，ADDR位置1。连续读取SR1寄存器和SR2寄存器可以清除ADDR位。由于发送的是写指令，收到从机的回应后，TxE位置1。向DR寄存器中写入要访问的EEPROM中数据的地址0x01。写入DR寄存器清除了TxE位并开始传输数据。如果收到从机的回应，TxE位又置1，然后再发送一个起始位，等待起始位发送完成。然后向DR寄存器中写入从机地址0x50加读指令。等待收到回应后，ADDR位置1。连续读取SR1寄存器和SR2寄存器可以清除ADDR位。清除ADDR位后，主机会继续发送时钟信号，从从机读取数据。由于只读一个字节的数据，因此需要将主机ACK置0，将STOP置1，即再收到从机的数据后不回应并产生一个停止信号结束通信。接收完从机传输的一个字节的数据后，RxNE置1，可以通过读取DR寄存器获取数据并同时清除RxNE位。

I2C1->CR1|=(1<<8); // START=1

**while**((I2C1->SR1 &0x01)==0); //wait for SB==1

I2C1->DR=(0x50<<1)+0;// write to 0x50 write address first

**while**((I2C1->SR1 &0x02)==0); //wait for ADDR==1

tmp=I2C1->SR2;

**while**((I2C1->SR1 & (1<<7))==0); //wait for TxE==1

I2C1->DR=0x01;

**while**((I2C1->SR1 & (1<<7))==0); //wait for TxE==1

I2C1->CR1|=(1<<8); // START=1

**while**((I2C1->SR1 &0x01)==0); //wait for SB==1

I2C1->DR=(0x50<<1)+1;// write to 0x50

**while**((I2C1->SR1 &0x02)==0); //wait for ADDR==1

tmp=I2C1->SR2;

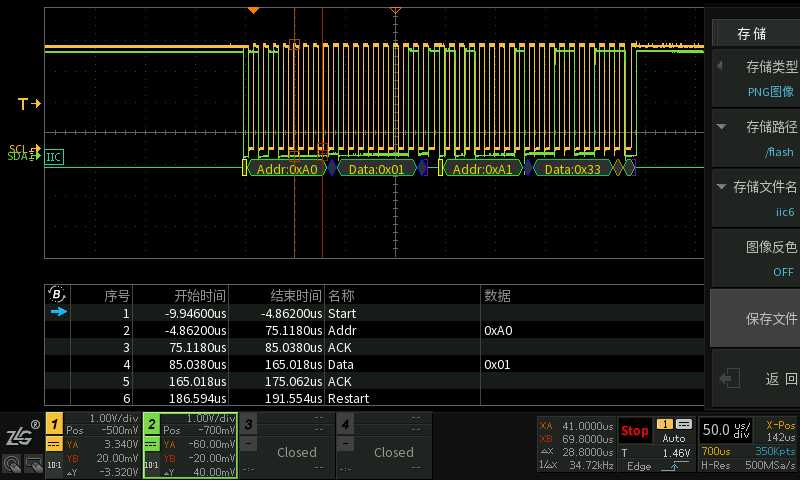
I2C1->CR1&=~(1<<10); //ACK=0

I2C1->CR1|=(0x01<<9); // STOP=1

**while**((I2C1->SR1 & (1<<6))==0); //wait for RxNE==1

b=I2C1->DR;

用示波器捕获总线上的信号，如图所示。可见总线上传输的数据，与预期的一致。



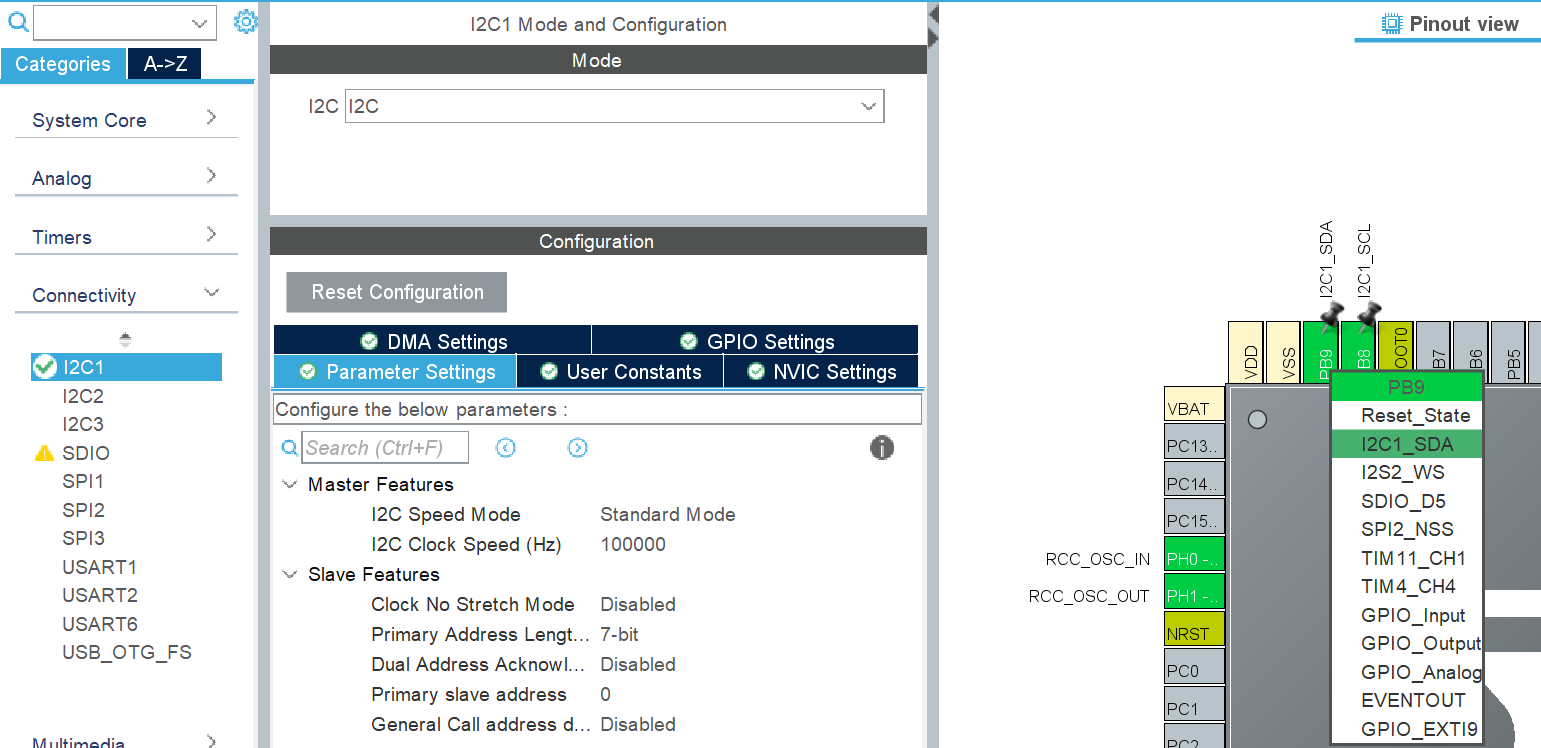
## STM32F401的I2C模块的应用：读写EEPROM(HAL库方式)

**通过HAL库函数读写EEPROM**

使用寄存器读写EEPROM可以完成所需要的功能，但是可读性强，编程比较繁琐。HAL库提供了更加简便的编程方式完成存储器类I2C器件的读写。

1. **图形界面初始化设置：**

首先新建一个工程，或者复制HelloWorld工程，取名为F401\_I2C1\_EEPROM\_HAL。完成新工程的正确配置，包括调试引脚，时钟等。双击打开F401\_I2C1\_EEPROM\_HAL.ioc文件。



选择左侧Connectivity菜单中的I2C1模块，在弹出的配置界面中，将I2C的工作模式选择为I2C。其他参数如通信速率确认为标准模式100kHz。使用7bit地址等。在右侧的引脚配置界面中，单击PB9引脚，将其设置为I2C1\_SDA。同样将PB8引脚设置为I2C1\_SCL。设置完毕后保存并生成代码。

查看生成的代码，我们会发现在MX\_GPIO\_Init();函数下面多了MX\_I2C1\_Init();函数。同时，多了一个全局变量的定义：

I2C\_HandleTypeDef hi2c1;

首先查看自动生成的C1\_Init()函数：

**static** **void** **MX\_I2C1\_Init**(**void**)

{

hi2c1.Instance = I2C1;

hi2c1.Init.ClockSpeed = 100000;

hi2c1.Init.DutyCycle = I2C\_DUTYCYCLE\_2;

hi2c1.Init.OwnAddress1 = 0;

hi2c1.Init.AddressingMode = I2C\_ADDRESSINGMODE\_7BIT;

hi2c1.Init.DualAddressMode = I2C\_DUALADDRESS\_DISABLE;

hi2c1.Init.OwnAddress2 = 0;

hi2c1.Init.GeneralCallMode = I2C\_GENERALCALL\_DISABLE;

hi2c1.Init.NoStretchMode = I2C\_NOSTRETCH\_DISABLE;

**if** (HAL\_I2C\_Init(&hi2c1) != *HAL\_OK*)

{

Error\_Handler();

}

}

可见，函数中，首先把I2C1的指针赋值给了hi2c1变量的Instance元素。这样访问I2C1和访问hi2c1.Instance样的。然后在hi2c1.Init素中，填入图形界面中I2C1模块的设置值，如波特率，地址位数等。注意，hi2c1.Init也是一个结构体，其类型为I2C\_InitTypeDef。

**typedef** **struct**

{

uint32\_t ClockSpeed; /\*!< Specifies the clock frequency.

This parameter must be set to a value lower than 400kHz \*/

uint32\_t DutyCycle; /\*!< Specifies the I2C fast mode duty cycle.

This parameter can be a value of @ref I2C\_duty\_cycle\_in\_fast\_mode \*/

uint32\_t OwnAddress1; /\*!< Specifies the first device own address.

This parameter can be a 7-bit or 10-bit address. \*/

uint32\_t AddressingMode; /\*!< Specifies if 7-bit or 10-bit addressing mode is selected.

This parameter can be a value of @ref I2C\_addressing\_mode \*/

uint32\_t DualAddressMode; /\*!< Specifies if dual addressing mode is selected.

This parameter can be a value of @ref I2C\_dual\_addressing\_mode \*/

uint32\_t OwnAddress2; /\*!< Specifies the second device own address if dual addressing mode is selected

This parameter can be a 7-bit address. \*/

uint32\_t GeneralCallMode; /\*!< Specifies if general call mode is selected.

This parameter can be a value of @ref I2C\_general\_call\_addressing\_mode \*/

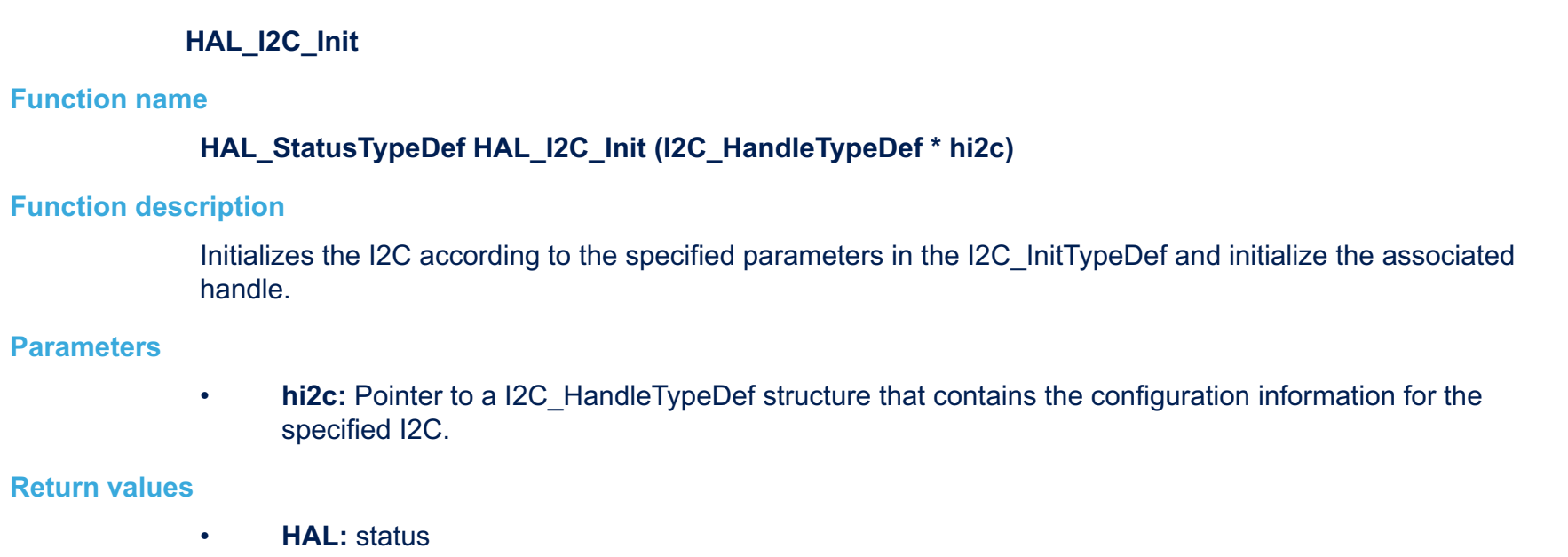
uint32\_t NoStretchMode; /\*!< Specifies if nostretch mode is selected.

This parameter can be a value of @ref I2C\_nostretch\_mode \*/

} I2C\_InitTypeDef;

最后MX\_I2C1\_Init函数调用了HAL\_I2C\_Init函数，传入hi2c1变量的地址，初始化I2C1模块。

查看HAL\_I2C\_Init函数的说明如下：



根据说明，HAL\_I2C\_Init函数用于根据设置的参数初始化，而设定的参数存在一个I2C\_InitTypeDef数据类型中，也就是hi2c1.Init。

进入HAL\_I2C\_Init函数查看其如何对I2C模块进行初始化，截取HAL\_I2C\_Init函数其中的一段：

HAL\_I2C\_MspInit(hi2c);

/\* Disable the selected I2C peripheral \*/

\_\_HAL\_I2C\_DISABLE(hi2c);

/\*Reset I2C\*/

hi2c->Instance->CR1 |= I2C\_CR1\_SWRST;

hi2c->Instance->CR1 &= ~I2C\_CR1\_SWRST;

/\* Get PCLK1 frequency \*/

pclk1 = HAL\_RCC\_GetPCLK1Freq();

/\* Check the minimum allowed PCLK1 frequency \*/

if (I2C\_MIN\_PCLK\_FREQ(pclk1, hi2c->Init.ClockSpeed) == 1U)

{

return HAL\_ERROR;

}

/\* Calculate frequency range \*/

freqrange = I2C\_FREQRANGE(pclk1);

/\*---------------------------- I2Cx CR2 Configuration ----------------------\*/

/\* Configure I2Cx: Frequency range \*/

MODIFY\_REG(hi2c->Instance->CR2, I2C\_CR2\_FREQ, freqrange);

/\*---------------------------- I2Cx TRISE Configuration --------------------\*/

/\* Configure I2Cx: Rise Time \*/

MODIFY\_REG(hi2c->Instance->TRISE, I2C\_TRISE\_TRISE, I2C\_RISE\_TIME(freqrange, hi2c->Init.ClockSpeed));

/\*---------------------------- I2Cx CCR Configuration ----------------------\*/

/\* Configure I2Cx: Speed \*/

MODIFY\_REG(hi2c->Instance->CCR, (I2C\_CCR\_FS | I2C\_CCR\_DUTY | I2C\_CCR\_CCR), I2C\_SPEED(pclk1, hi2c->Init.ClockSpeed, hi2c->Init.DutyCycle));

/\*---------------------------- I2Cx CR1 Configuration ----------------------\*/

/\* Configure I2Cx: Generalcall and NoStretch mode \*/

MODIFY\_REG(hi2c->Instance->CR1, (I2C\_CR1\_ENGC | I2C\_CR1\_NOSTRETCH), (hi2c->Init.GeneralCallMode | hi2c->Init.NoStretchMode));

/\*---------------------------- I2Cx OAR1 Configuration ---------------------\*/

/\* Configure I2Cx: Own Address1 and addressing mode \*/

MODIFY\_REG(hi2c->Instance->OAR1, (I2C\_OAR1\_ADDMODE | I2C\_OAR1\_ADD8\_9 | I2C\_OAR1\_ADD1\_7 | I2C\_OAR1\_ADD0), (hi2c->Init.AddressingMode | hi2c->Init.OwnAddress1));

/\*---------------------------- I2Cx OAR2 Configuration ---------------------\*/

/\* Configure I2Cx: Dual mode and Own Address2 \*/

MODIFY\_REG(hi2c->Instance->OAR2, (I2C\_OAR2\_ENDUAL | I2C\_OAR2\_ADD2), (hi2c->Init.DualAddressMode | hi2c->Init.OwnAddress2));

/\* Enable the selected I2C peripheral \*/

\_\_HAL\_I2C\_ENABLE(hi2c);

可以看出，该函数为了初始化I2C模块，首先调用了HAL\_I2C\_MspInit函数。该函数在stm32f4xx\_hal\_msp.c文件中定义：

**void** **HAL\_I2C\_MspInit**(I2C\_HandleTypeDef\* hi2c)

{

GPIO\_InitTypeDef GPIO\_InitStruct = {0};

**if**(hi2c->Instance==I2C1)

{

/\* USER CODE BEGIN I2C1\_MspInit 0 \*/

/\* USER CODE END I2C1\_MspInit 0 \*/

\_\_HAL\_RCC\_GPIOB\_CLK\_ENABLE();

/\*\*I2C1 GPIO Configuration

PB8 ------> I2C1\_SCL

PB9 ------> I2C1\_SDA

\*/

GPIO\_InitStruct.Pin = GPIO\_PIN\_8|GPIO\_PIN\_9;

GPIO\_InitStruct.Mode = GPIO\_MODE\_AF\_OD;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_VERY\_HIGH;

GPIO\_InitStruct.Alternate = GPIO\_AF4\_I2C1;

HAL\_GPIO\_Init(GPIOB, &GPIO\_InitStruct);

/\* Peripheral clock enable \*/

\_\_HAL\_RCC\_I2C1\_CLK\_ENABLE();

/\* USER CODE BEGIN I2C1\_MspInit 1 \*/

/\* USER CODE END I2C1\_MspInit 1 \*/

}

}

可见，HAL\_I2C\_MspInit函数主要打开了GPIOB的时钟，然后初始化了所用到的PB8、PB9引脚，然后打开了I2C1模块的时钟。

运行完HAL\_I2C\_MspInit函数后，回到HAL\_I2C\_Init函数中，通过\_\_HAL\_I2C\_DISABLE(hi2c);关闭I2C1模块的使能位，然后重启I2C1模块。然后根据设置的波特率和系统的时钟频率，设置CR2寄存器的FREQ位、TRISE寄存器和CCR寄存器，用与获得正确的波特率。然后设置CR1、OAR1、OAR2等寄存器。

最后通过\_\_HAL\_I2C\_ENABLE(hi2c)语句使能I2C1模块。

根据定义可知，\_\_HAL\_I2C\_ENABLE语句是将CR1寄存器的PE位置1，而\_\_HAL\_I2C\_DISABLE语句是将CR1寄存器的PE位置0。此功能与寄存器的描述一致。

**#define** \_\_HAL\_I2C\_ENABLE(\_\_HANDLE\_\_) SET\_BIT((\_\_HANDLE\_\_)->Instance->CR1, I2C\_CR1\_PE)

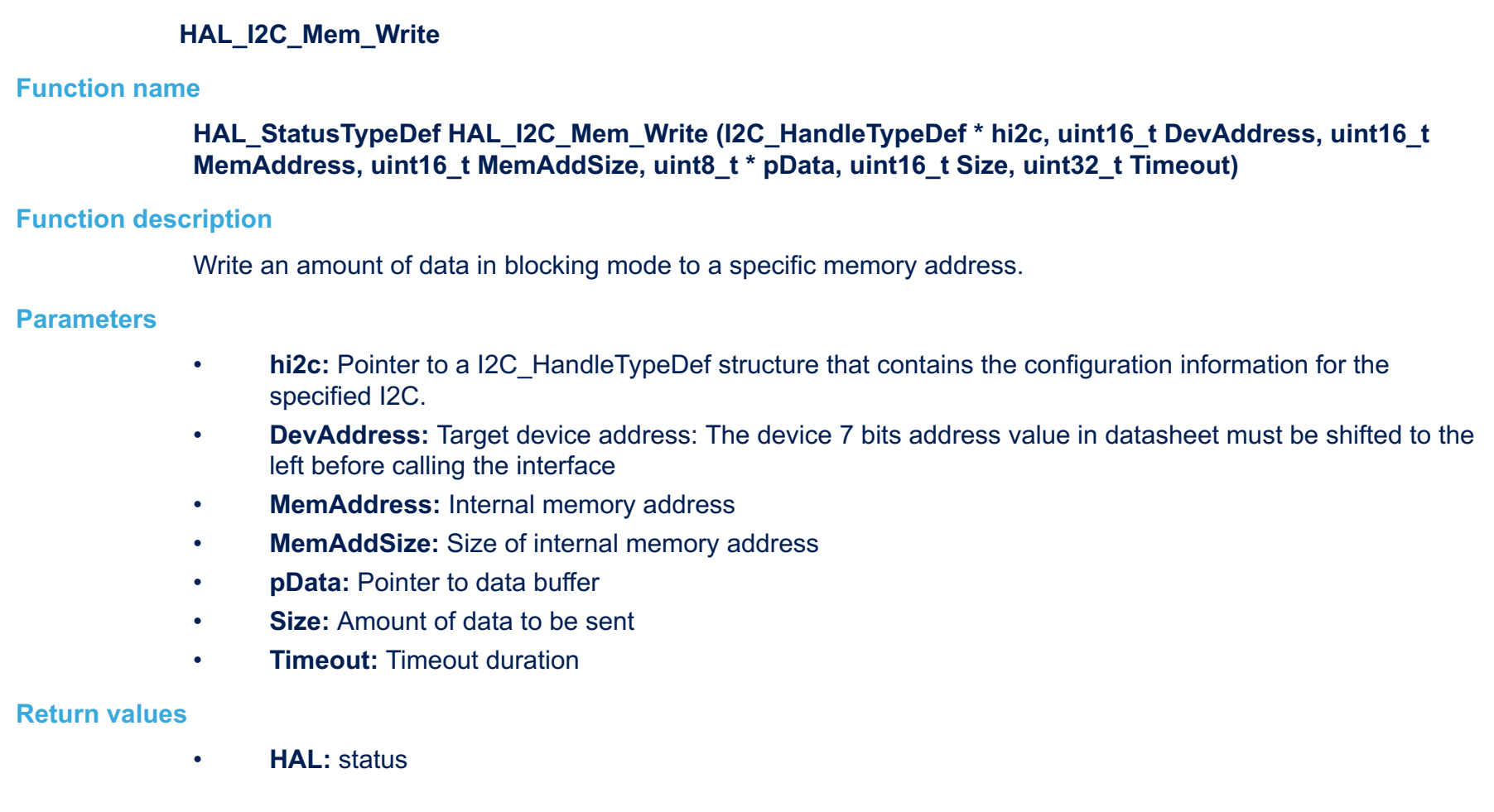
**#define** \_\_HAL\_I2C\_DISABLE(\_\_HANDLE\_\_) CLEAR\_BIT((\_\_HANDLE\_\_)->Instance->CR1, I2C\_CR1\_PE)

1. **写数据：**

完成I2C1模块的初始化之后，就需要使用I2C1模块对EEPROM进行读写。HAL库提供了两个函数

HAL\_I2C\_Mem\_Write和HAL\_I2C\_Mem\_Read来读写I2C总线上的存储器。

其中HAL\_I2C\_Mem\_Write函数的说明如下：



其功能描述为：采用阻塞通信的方式，向指定存储地址中写一段数据。

其输入参数较多，第一个\*hi2c参数是I2C\_HandleTypeDef类型的指针，如果要操作I2C1模块，则需要将&hi2c1传入到HAL\_I2C\_Mem\_Write函数。

第二个参数DevAddress为EEPROM的器件地址，即该器件在I2C总线上的地址，实验板上的地址是0x50。注意，该函数要求将器件地址左移一位后再传入。也就是说第二个参数应该设置为0x50<<1.

第三个参数MemAddress为要访问的数据在EEPROM中的地址，根据要求，这里要填0x01。

第四个参数MemAddSize为数据地址的长度，EEPROM M24C02写时序中存储器地址的长度，M24C02使用一个字节的地址位，因此此处填1。

第五个参数\*pData为指向要写入的数据的指针。如果数据存在数组中，只要填入数组的名字即可。但注意数组的类型要是uint8\_t类型的。

第六个参数Size是要写入的数据的个数，以字节为单位，写一个字节，就填1。

第七个参数为超时时间，以ms为单位。如果经过这个设置的时间后还没有完成写入操作，就停止I2C总线的读写并返回HAL\_ERROR表示写入失败。如果再设置的时间内完成了数据写入操作，则返回HAL\_OK表示写入成功。

例如，要向EEPROM中写入数据，首先定义一个数组全局变量，存放要传输的数据：

uint8\_t i2cdataW[10];

然后将要发送的数据填入到该数组中，调用HAL\_I2C\_Mem\_Write，就可以将数据写入到EEPROM中。

i2cdataW[0]=0x33;

HAL\_I2C\_Mem\_Write(&hi2c1,0x50<<1,0x01,1,i2cdataW,1,10);

查看HAL\_I2C\_Mem\_Write函数的实现方式：

HAL\_StatusTypeDef HAL\_I2C\_Mem\_Write(I2C\_HandleTypeDef \*hi2c, uint16\_t DevAddress, uint16\_t MemAddress, uint16\_t MemAddSize, uint8\_t \*pData, uint16\_t Size, uint32\_t Timeout)

{

……

/\* Check if the I2C is already enabled \*/

if ((hi2c->Instance->CR1 & I2C\_CR1\_PE) != I2C\_CR1\_PE)

{

/\* Enable I2C peripheral \*/

\_\_HAL\_I2C\_ENABLE(hi2c);

}

……

/\* Send Slave Address and Memory Address \*/

if (I2C\_RequestMemoryWrite(hi2c, DevAddress, MemAddress, MemAddSize, Timeout, tickstart) != HAL\_OK)

{

return HAL\_ERROR;

}

while (hi2c->XferSize > 0U)

{

/\* Wait until TXE flag is set \*/

if (I2C\_WaitOnTXEFlagUntilTimeout(hi2c, Timeout, tickstart) != HAL\_OK)

{

if (hi2c->ErrorCode == HAL\_I2C\_ERROR\_AF)

{

/\* Generate Stop \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_STOP);

}

return HAL\_ERROR;

}

/\* Write data to DR \*/

hi2c->Instance->DR = \*hi2c->pBuffPtr;

/\* Increment Buffer pointer \*/

hi2c->pBuffPtr++;

/\* Update counter \*/

hi2c->XferSize--;

hi2c->XferCount--;

if ((\_\_HAL\_I2C\_GET\_FLAG(hi2c, I2C\_FLAG\_BTF) == SET) && (hi2c->XferSize != 0U))

{

/\* Write data to DR \*/

hi2c->Instance->DR = \*hi2c->pBuffPtr;

/\* Increment Buffer pointer \*/

hi2c->pBuffPtr++;

/\* Update counter \*/

hi2c->XferSize--;

hi2c->XferCount--;

}

}

/\* Wait until BTF flag is set \*/

if (I2C\_WaitOnBTFFlagUntilTimeout(hi2c, Timeout, tickstart) != HAL\_OK)

{

if (hi2c->ErrorCode == HAL\_I2C\_ERROR\_AF)

{

/\* Generate Stop \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_STOP);

}

return HAL\_ERROR;

}

/\* Generate Stop \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_STOP);

……

return HAL\_OK;

}

else

{

return HAL\_BUSY;

}

}

可知，该函数首先通过\_\_HAL\_I2C\_ENABLE(hi2c);使能I2C模块，然后通过I2C\_RequestMemoryWrite发送起始位、从机地址和数据地址

而I2C\_RequestMemoryWrite的实现方式如下。

static HAL\_StatusTypeDef I2C\_RequestMemoryWrite(I2C\_HandleTypeDef \*hi2c, uint16\_t DevAddress, uint16\_t MemAddress, uint16\_t MemAddSize, uint32\_t Timeout, uint32\_t Tickstart)

{

/\* Generate Start \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_START);

/\* Wait until SB flag is set \*/

if (I2C\_WaitOnFlagUntilTimeout(hi2c, I2C\_FLAG\_SB, RESET, Timeout, Tickstart) != HAL\_OK)

{

if (READ\_BIT(hi2c->Instance->CR1, I2C\_CR1\_START) == I2C\_CR1\_START)

{

hi2c->ErrorCode = HAL\_I2C\_WRONG\_START;

}

return HAL\_TIMEOUT;

}

/\* Send slave address \*/

hi2c->Instance->DR = I2C\_7BIT\_ADD\_WRITE(DevAddress);

/\* Wait until ADDR flag is set \*/

if (I2C\_WaitOnMasterAddressFlagUntilTimeout(hi2c, I2C\_FLAG\_ADDR, Timeout, Tickstart) != HAL\_OK)

{

return HAL\_ERROR;

}

/\* Clear ADDR flag \*/

\_\_HAL\_I2C\_CLEAR\_ADDRFLAG(hi2c);

/\* Wait until TXE flag is set \*/

if (I2C\_WaitOnTXEFlagUntilTimeout(hi2c, Timeout, Tickstart) != HAL\_OK)

{

if (hi2c->ErrorCode == HAL\_I2C\_ERROR\_AF)

{

/\* Generate Stop \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_STOP);

}

return HAL\_ERROR;

}

/\* If Memory address size is 8Bit \*/

if (MemAddSize == I2C\_MEMADD\_SIZE\_8BIT)

{

/\* Send Memory Address \*/

hi2c->Instance->DR = I2C\_MEM\_ADD\_LSB(MemAddress);

}

……

return HAL\_OK;

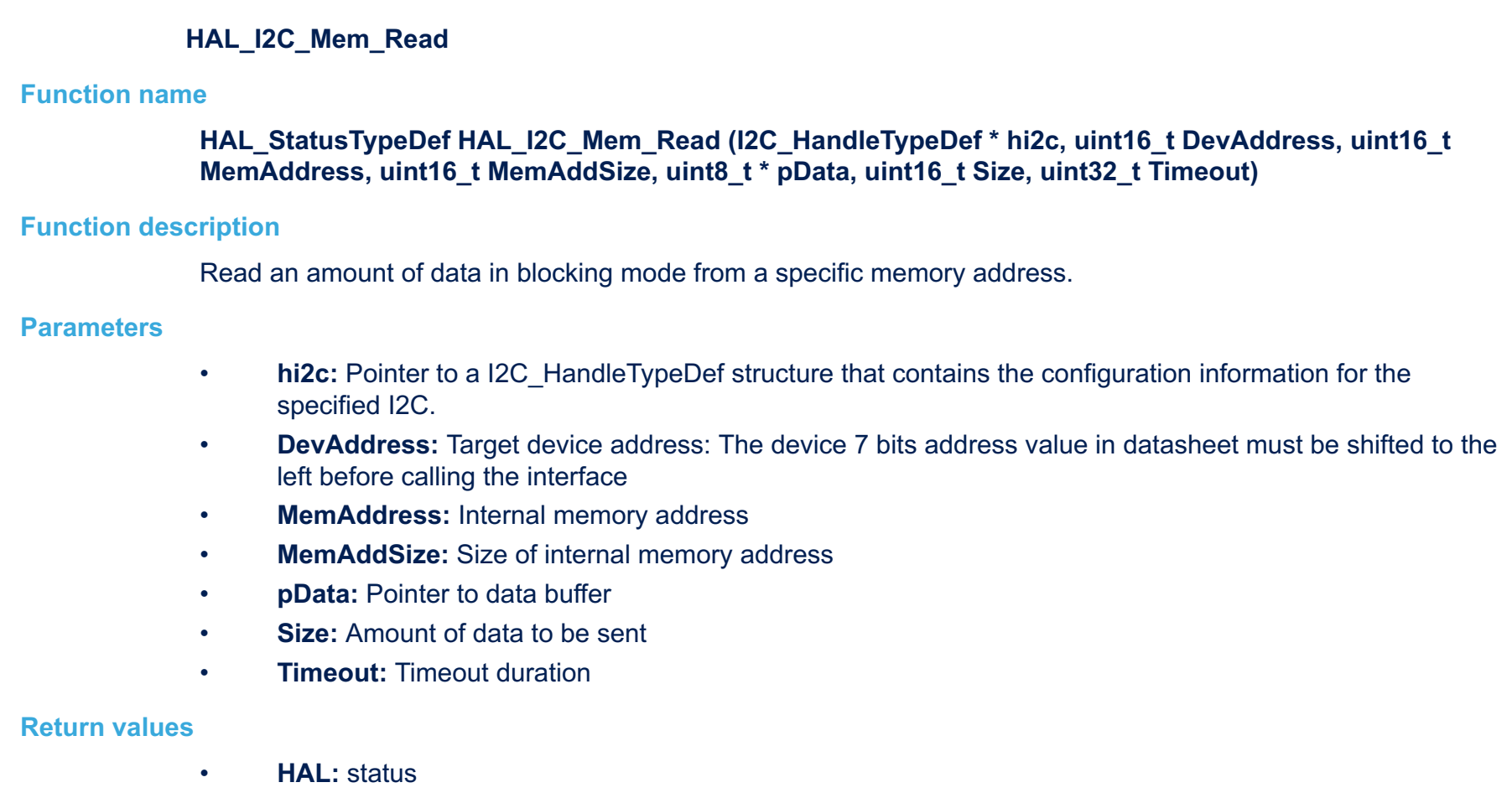
}

可见，I2C\_RequestMemoryWrite函数首先通过 SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_START);语句发送一个起始位，然后等待起始位发送完成后通过写DR寄存器发送从机地址。等待从机回应后TxE置1，然后再将数据地址写入到DR寄存器中发送到从机。

回到HAL\_I2C\_Mem\_Write函数中，待发送完从机地址和数据地址后，开始根据数据指针向DR寄存器中写入数据，每发送一个字节的数据，都要修改数据指针和发送数据的个数。数据发送完后，调用SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_STOP);语句发送结束位，完成数据传输。传输过程中，如果某个环节的操作时间过长，达到了Timeout参数设定的时间，测发送停止位并返回HAL\_ERROR。综上所述，HAL库函数HAL\_I2C\_Mem\_Write只是帮我们封装好了发送数据所需的操作寄存器的流程，让我们能够更方便的写代码，其本质还是通过操作寄存器来实现。

1. **读数据：**

再看HAL\_I2C\_Mem\_Read函数，其函数说明如下：



其功能描述为：从指定存储地址中读出一段数据。

其输入参数较多，第一个\*hi2c参数是I2C\_HandleTypeDef类型的指针，如果要操作I2C1模块，则需要将&hi2c1传入到HAL\_I2C\_Mem\_Read函数。

第二个参数DevAddress为EEPROM的器件地址，即该器件在I2C总线上的地址，实验板上的地址是0x50。注意，该函数要求将器件地址左移一位后再传入。也就是说第二个参数应该设置为0x50<<1.

第三个参数MemAddress为要访问的数据再EEPROM中的地址，根据要求，这里要填0x01。

第四个参数MemAddSize为地址的长度，EEPROM M24C02写时序中存储器地址的长度，M24C02使用一个字节的地址位，因此此处填1。

第五个参数\*pData为指向存储读出数据的数组的指针。如果要把读出的数据存在数组中，只要填入数组的名字即可。但注意数组的类型要是uint8\_t类型的。

第六个参数Size是要读出的数据的个数，以字节为单位，写一个字节，就填1。

第七个参数Timeout为超时时间，以ms为单位。如果经过这个设置的时间后还没有完成读出操作，就停止I2C总线的读写并返回HAL\_ERROR表示写入失败。如果再设置的时间内完成了数据写入操作，则返回HAL\_OK表示写入成功。

例如，要从EEPROM中读出数据，首先定义一个数组全局变量，用来存储读出的数据：

uint8\_t i2cdataR[10];

然后调用HAL\_I2C\_Mem\_Read，就可以将EEPROM中指定地址的数据读出到i2cdataR数组中。

HAL\_I2C\_Mem\_Read(&hi2c1,0x50<<1,0x01,1,i2cdataR,1,10);

查看HAL\_I2C\_Mem\_Read函数的实现方式，并摘取主要代码：

HAL\_StatusTypeDef HAL\_I2C\_Mem\_Read(I2C\_HandleTypeDef \*hi2c, uint16\_t DevAddress, uint16\_t MemAddress, uint16\_t MemAddSize, uint8\_t \*pData, uint16\_t Size, uint32\_t Timeout)

{

……

/\* Check if the I2C is already enabled \*/

if ((hi2c->Instance->CR1 & I2C\_CR1\_PE) != I2C\_CR1\_PE)

{

/\* Enable I2C peripheral \*/

\_\_HAL\_I2C\_ENABLE(hi2c);

}

……

/\* Send Slave Address and Memory Address \*/

if (I2C\_RequestMemoryRead(hi2c, DevAddress, MemAddress, MemAddSize, Timeout, tickstart) != HAL\_OK)

{

return HAL\_ERROR;

}

……

else if (hi2c->XferSize == 1U)

{

/\* Disable Acknowledge \*/

CLEAR\_BIT(hi2c->Instance->CR1, I2C\_CR1\_ACK);

/\* Clear ADDR flag \*/

\_\_HAL\_I2C\_CLEAR\_ADDRFLAG(hi2c);

/\* Generate Stop \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_STOP);

}

……

else

{

/\* Clear ADDR flag \*/

\_\_HAL\_I2C\_CLEAR\_ADDRFLAG(hi2c);

}

while (hi2c->XferSize > 0U)

{

if (hi2c->XferSize <= 3U)

{

/\* One byte \*/

if (hi2c->XferSize == 1U)

{

/\* Wait until RXNE flag is set \*/

if (I2C\_WaitOnRXNEFlagUntilTimeout(hi2c, Timeout, tickstart) != HAL\_OK)

{

return HAL\_ERROR;

}

/\* Read data from DR \*/

\*hi2c->pBuffPtr = (uint8\_t)hi2c->Instance->DR;

/\* Increment Buffer pointer \*/

hi2c->pBuffPtr++;

/\* Update counter \*/

hi2c->XferSize--;

hi2c->XferCount--;

}

……

else

{

/\* Wait until RXNE flag is set \*/

if (I2C\_WaitOnRXNEFlagUntilTimeout(hi2c, Timeout, tickstart) != HAL\_OK)

{

return HAL\_ERROR;

}

/\* Read data from DR \*/

\*hi2c->pBuffPtr = (uint8\_t)hi2c->Instance->DR;

/\* Increment Buffer pointer \*/

hi2c->pBuffPtr++;

/\* Update counter \*/

hi2c->XferSize--;

hi2c->XferCount--;

……

}

}

}

……

return HAL\_OK;

}

else

{

return HAL\_BUSY;

}

}

可见，该函数也是在使能I2C模块后，调用I2C\_RequestMemoryRead发送起始位、从机地址等信息。查看I2C\_RequestMemoryRead函数的代码：

static HAL\_StatusTypeDef I2C\_RequestMemoryRead(I2C\_HandleTypeDef \*hi2c, uint16\_t DevAddress, uint16\_t MemAddress, uint16\_t MemAddSize, uint32\_t Timeout, uint32\_t Tickstart)

{

/\* Enable Acknowledge \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_ACK);

/\* Generate Start \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_START);

/\* Wait until SB flag is set \*/

if (I2C\_WaitOnFlagUntilTimeout(hi2c, I2C\_FLAG\_SB, RESET, Timeout, Tickstart) != HAL\_OK)

{

……

}

/\* Send slave address \*/

hi2c->Instance->DR = I2C\_7BIT\_ADD\_WRITE(DevAddress);

/\* Wait until ADDR flag is set \*/

if (I2C\_WaitOnMasterAddressFlagUntilTimeout(hi2c, I2C\_FLAG\_ADDR, Timeout, Tickstart) != HAL\_OK)

{

return HAL\_ERROR;

}

/\* Clear ADDR flag \*/

\_\_HAL\_I2C\_CLEAR\_ADDRFLAG(hi2c);

/\* Wait until TXE flag is set \*/

if (I2C\_WaitOnTXEFlagUntilTimeout(hi2c, Timeout, Tickstart) != HAL\_OK)

{

……

}

/\* If Memory address size is 8Bit \*/

if (MemAddSize == I2C\_MEMADD\_SIZE\_8BIT)

{

/\* Send Memory Address \*/

hi2c->Instance->DR = I2C\_MEM\_ADD\_LSB(MemAddress);

}

……

/\* Wait until TXE flag is set \*/

if (I2C\_WaitOnTXEFlagUntilTimeout(hi2c, Timeout, Tickstart) != HAL\_OK)

{

……

}

/\* Generate Restart \*/

SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_START);

/\* Wait until SB flag is set \*/

if (I2C\_WaitOnFlagUntilTimeout(hi2c, I2C\_FLAG\_SB, RESET, Timeout, Tickstart) != HAL\_OK)

{

……

}

/\* Send slave address \*/

hi2c->Instance->DR = I2C\_7BIT\_ADD\_READ(DevAddress);

/\* Wait until ADDR flag is set \*/

if (I2C\_WaitOnMasterAddressFlagUntilTimeout(hi2c, I2C\_FLAG\_ADDR, Timeout, Tickstart) != HAL\_OK)

{

return HAL\_ERROR;

}

return HAL\_OK;

}

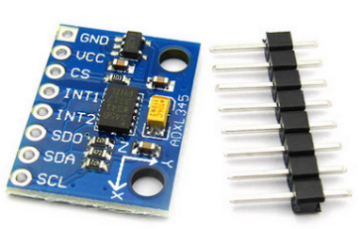
在I2C\_RequestMemoryRead函数中，首先设置ACK=1，打开了主机的自动回应。然后使用SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_START);发送起始位。等起始位发送完成后，使用hi2c->Instance->DR = I2C\_7BIT\_ADD\_WRITE(DevAddress);代码发送从机地址和写指令，等待从机回应后清除ADDR位，同时TxE位置1。向DR寄存器中填入数据地址，等待数据传送完成，收到回应后，TxE置1。之后，使用SET\_BIT(hi2c->Instance->CR1, I2C\_CR1\_START);代码重新发送起始位，等待起始位发送完成后，使用hi2c->Instance->DR = I2C\_7BIT\_ADD\_READ(DevAddress);代码发送从机地址和读指令，待收到从机回应后，返回到HAL\_I2C\_Mem\_Read函数读取数据。

继续看HAL\_I2C\_Mem\_Read函数，执行完I2C\_RequestMemoryRead函数后，如果判断要读取的数据只有一个字节，则直接将ACK设置为0，关闭自动回应，然后清除ADDR，读取一个字节后，发送停止位结束通信。然后等待RxNE置1后，读取DR寄存器，取出数据。

同理，HAL库函数HAL\_I2C\_Mem\_Read函数也只是帮我们封装好了读取数据所需的操作寄存器的流程，让我们能够更方便的写代码，其本质也还是通过操作寄存器来实现。

## STM32F401的I2C模块的应用：ADXL345三轴加速度传感器

ADXL345三轴加速度传感器：能够在预先不知道物体运动方向的场合下，准确且全面的测量出物体的空间加速度，并且体积小



**常见应用领域：**

（1）**汽车电子**：

以车身安全系统为例，当车身受到撞击时，冲击传感器会在几微秒内将信号发送至该电子控制器。随后电子控制器会立即根据碰撞的强度、乘客数量及座椅/安全带的位置等参数，配合分布在整个车厢的传感器传回的数据进行计算和做出相应评估，并在最短的时间内通过电爆驱动器启动安全气囊保证乘客的生命安全。

（2）**便携式设备的抗冲击防护**：

便携式设备由于其应用场合的原因，经常会意外跌落或受到碰撞，而造成对内部元器件的巨大冲击。当跌落发生时，系统会检测到加速的突然变化，并执行相应的自我保护操作，如关闭抗震性能差的电子或机械器件，从而避免其受损，或发生硬盘磁头损坏或刮伤盘片等可能造成数据永久丢失的情况。

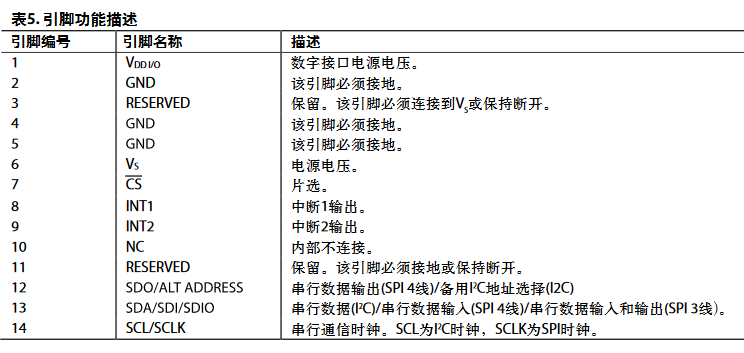
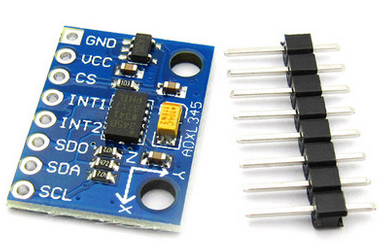
（3）**卫星导航**：

当进入卫星信号接收不良的区域或环境中就会因失去信号而丧失导航功能。基于MEMS技术的3轴加速度传感器配合陀螺仪或电子罗盘等元件一起可创建方位推算系统，对GPS系统实现互补性应用。

（4）**虚拟现实**：

例如AR/VR，游戏机手柄等，通过穿戴或手持的游戏装备、手柄，传感器同步感应人体加速度，根据加速度分析人体的动作，进而在游戏屏幕上同步呈现出相应的游戏动作，一些体感游戏甚至能达到与户外运动相同锻炼的效果。

1. **接线方式：**



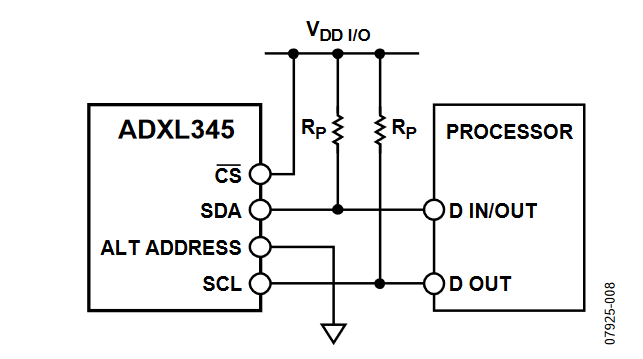
ADXL345使用I2C通信，只需要连接4根线：

VCC：电源线3.3V-5V

GND：地线

SDA：I2C总线的数据线，上拉

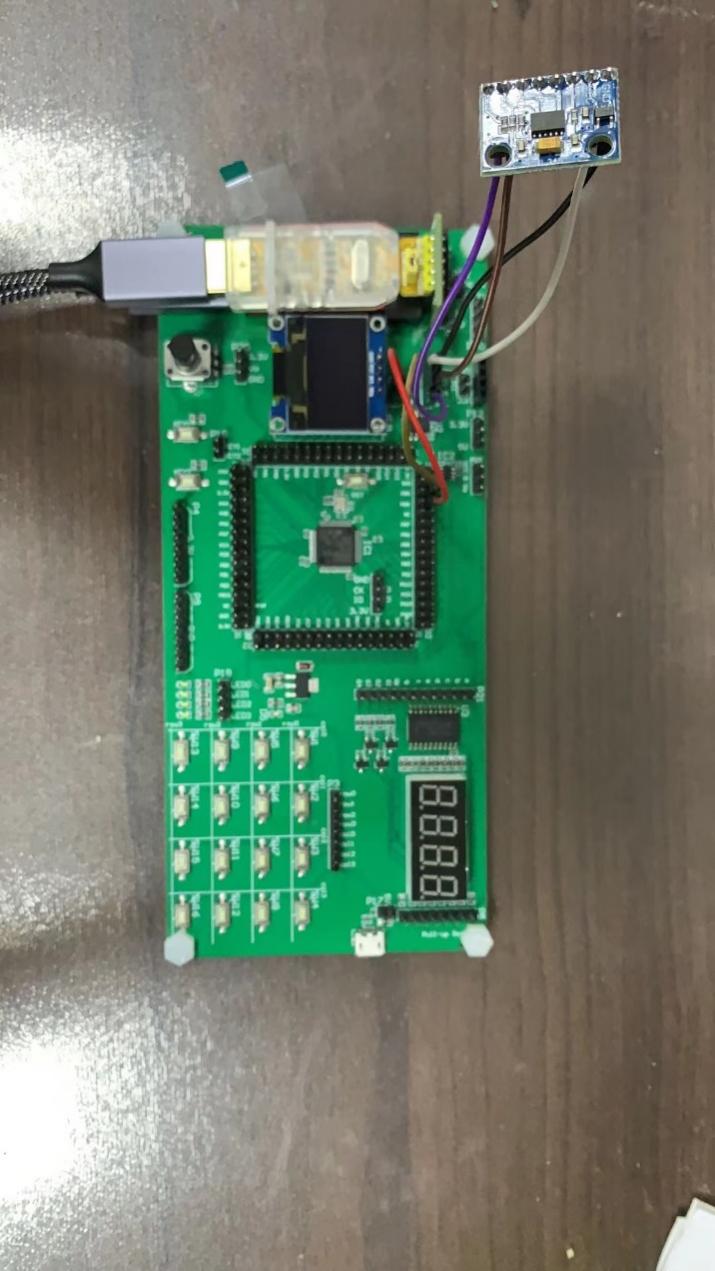
SCL：I2C总线的时钟线，上拉



将CS引脚拉高，ADXL345处于I2C模式，需要简单2线式连接。支持标准(100 kHz)和快速(400 kHz)数据传输模式

ALT ADDRESS引脚处于高电平，器件的7位I2C地址是0x1D，随后为R / W位。通过ALT ADDRESS引脚(引脚12)接地，可以选择备用I2C地址0x53(随后为R / W位)。

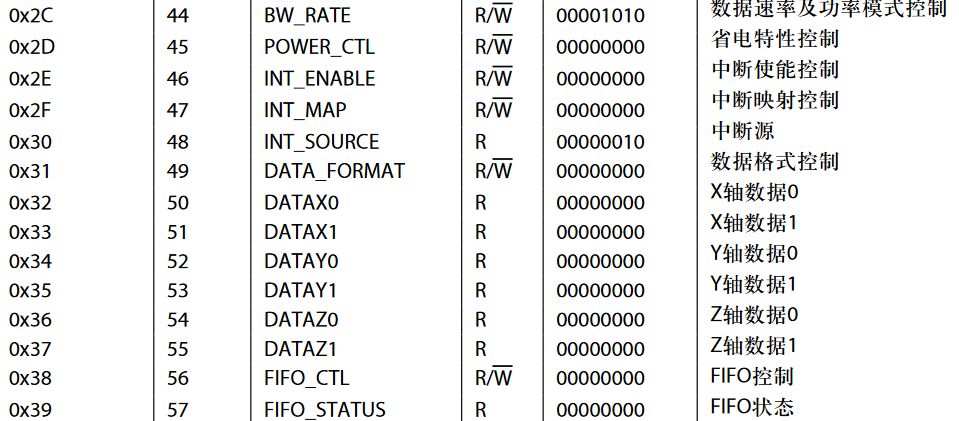
因此需要将传感器的VCC接实验板的3.3V，将传感器的GND接实验板的GND。将传感器的SCL接实验板的SCL，将传感器的SDA接实验板的SDA。同时将PB8、PB9用导线连接到实验板的SCL和SDA上。



1. **ADXL345的操作方式**

ADXL345的通过内部寄存器控制芯片并读写数据：



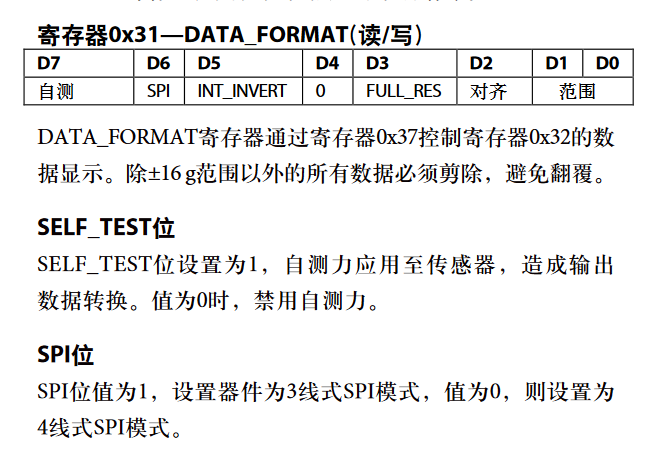


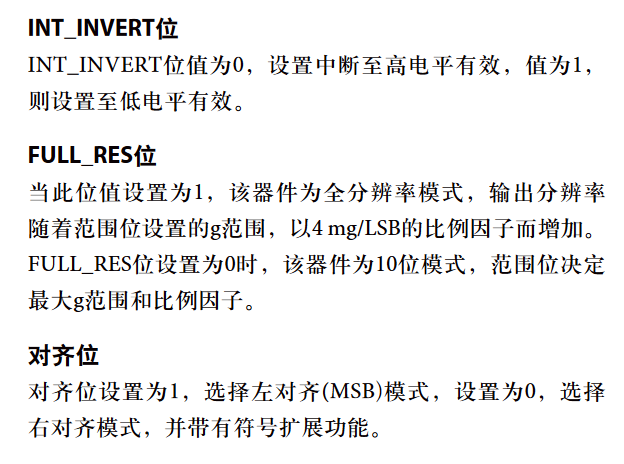
**几个关键的寄存器：**

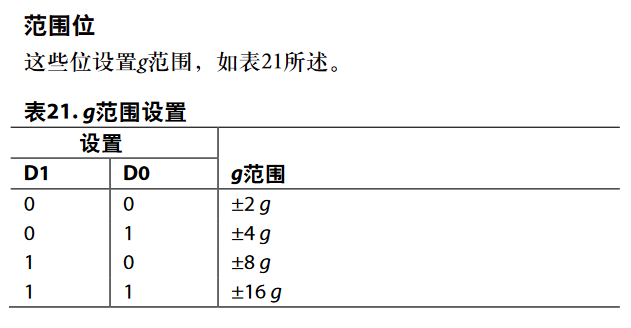
1. **DEVID 器件编号**

此寄存器里的数据位固定值0b11100101，即0xE5。如果从这个寄存器里读出的数据正确，说明I2C通信能够正常工作。

1. **DATA\_FORMAT数据格式寄存器**

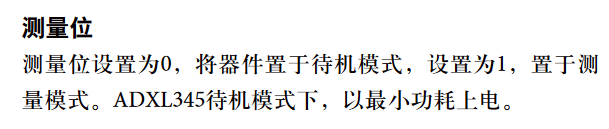




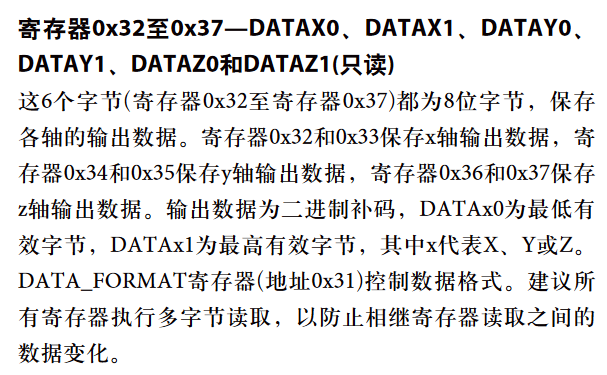


1. **省电特性控制POWER\_CTL：**





1. **输出加速度数据：DATAX0、DATAX1、DATAY0、DATAY1、DATAZ0和DATAZ1**

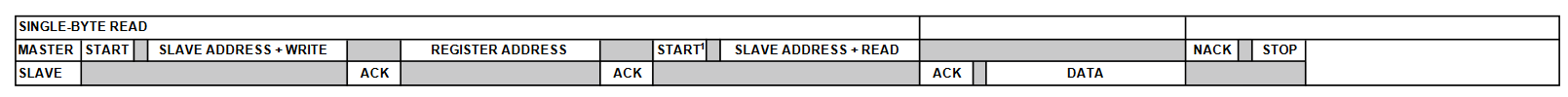


**ADXL345三轴加速度传感器的操作流程：**

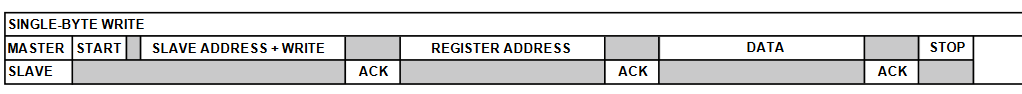
1. 读取DEVID 器件编号，确认I2C总线正常工作
2. 设置测量范围（写DATA\_FORMAT寄存器，只设置一次）
3. 进入测量模式（写POWER\_CTL寄存器，只设置一次）
4. 读取数据（DATAX0、DATAX1、DATAY0、DATAY1、DATAZ0和DATAZ1寄存器）

**通过I2C总线操作寄存器的方式：**

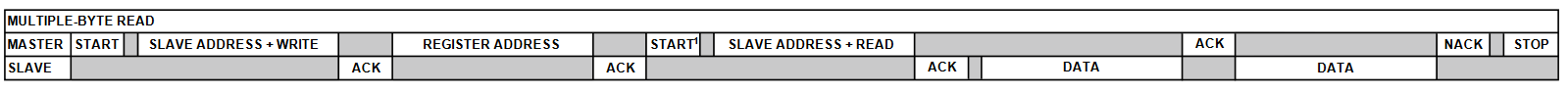
读单个寄存器：



写单个寄存器



读多个寄存器：



1. **使用STM32F401的I2C模块读写2. ADXL345**

* **读单个寄存器：**

读DEVID 器件编号，即读取0x00处的数据。

HAL\_I2C\_Mem\_Read(&hi2c1,0x53<<1,0x00,1,i2cdataR,1,10);



查看SDA和SCL的波形可以看到，读出的数据为0xE5，与预设的器件编号一致。这表示MCU与传感器的通信正常。

* **写单个寄存器**

即写0x31和0x2D处的数据

确认通信正常后，需要设置测量范围。本例采用全分辨率模式，即将0x31寄存器的第三位D3设置为1，其他位都为零。

i2cdataW[0]= (1<<3);

HAL\_I2C\_Mem\_Write(&hi2c1,0x53<<1,0x31,1,i2cdataW,1,10);

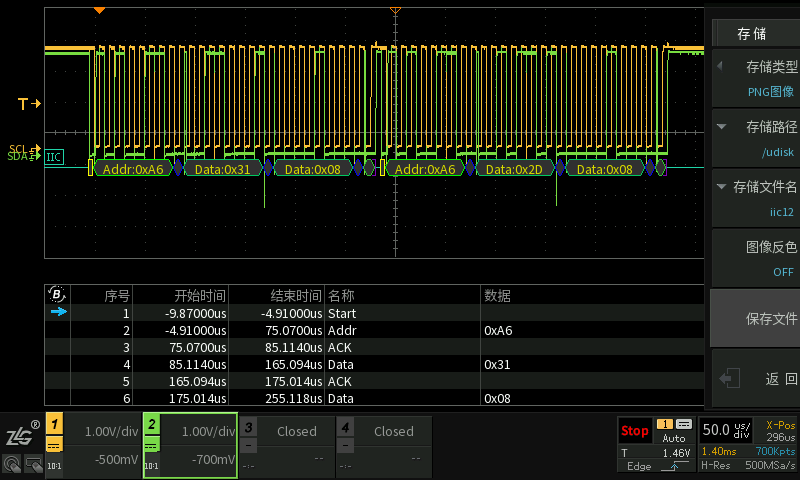
然后需要开启测量，即将0x2D的第三位D3设置为1;

i2cdataW[0]=(1<<3);

HAL\_I2C\_Mem\_Write(&hi2c1,0x53<<1,0x2D,1,i2cdataW,1,10);

这样传感器就处于测量状态了，可以读取数据获得测量结果。

查看SDA和SCL引脚上的波形，也可以看到总线上传输了正确的信息。



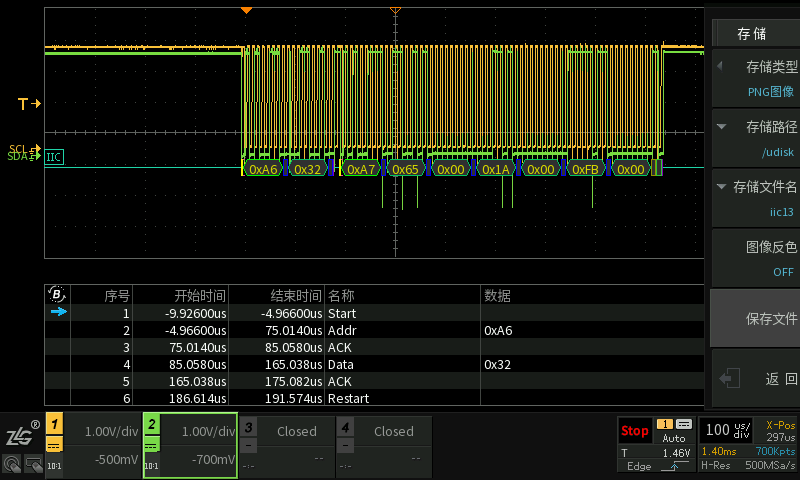
* **读多个寄存器：**

**即读0x32至0x37处的数据。**

保存到dataBuffer数组中。由于读取的是连续的6个字节，因此只需要设置一次地址，然后连续读取六个字节的数据，存到readBuffer数组中。

HAL\_I2C\_Mem\_Read(&hi2c1,0x53<<1,0x32,1,readBuffer,6,10);

查看SDA和SCL引脚上的波形，也可以看到总线上传输了正确的信息。



将数据读出之后，需要将DATAX0、DATAX1两个八位的数据，合成一个16位的数据：

x = ((short)readBuffer[1] << 8) + readBuffer[0];

y = ((short)readBuffer[3] << 8) + readBuffer[2];

z = ((short)readBuffer[5] << 8) + readBuffer[4];

再将这个16位的数据转换为加速度，便可以根据重力加速度的分布，获知传感器的倾角状态。

由于采用全分辨率模式，x、y、z每增加1，对应加速度增加0.004g=0.004\*9.8。

根据这个关系，定义浮点类型变量gx、gy、gz，然后将x、y、z转为为加速度。

