【STM32】HAL库 STM32CubeMX教程十二—IIC(读取AT24C02）

前言：

本系列教程将HAL库与STM32CubeMX结合在一起讲解，使您可以更快速的学会各个模块的使用

在之前的标准库中，STM32的硬件IIC非常复杂，更重要的是它**并不稳定**，所以都不推荐使用。
但是在我们的HAL库中，对硬件IIC做了全新的优化，使得之前软件IIC几百行代码，在HAL库中，只需要寥寥几行就可以完成 那么这篇文章将带你去感受下它的优异之处

这可能是目前关于STM32CubeMX的硬件iic 讲的最全面和详细的一篇文章之一了

所用工具：

1、芯片： STM32F103ZET6

2、STM32CubeMx软件

3、IDE： MDK-Keil软件

4、STM32F1xx/STM32F4xxHAL库

5、IIC： 使用硬件IIC1

知识概括：

通过本篇博客您将学到：

IIC的基本原理

STM32CubeMX创建IIC例程

HAL库IIC函数库

AT24C02 芯片原理

IIC 简介

IIC(Inter - Integrated Circuit)总线是一种由NXP（原PHILIPS）公司开发的两线式串行总线，用于连接微控制器及其外围设备。多用于主控制器和从器件间的主从通信，在小数据量场合使用，传输距离短，任意时刻只能有一个主机等特性。

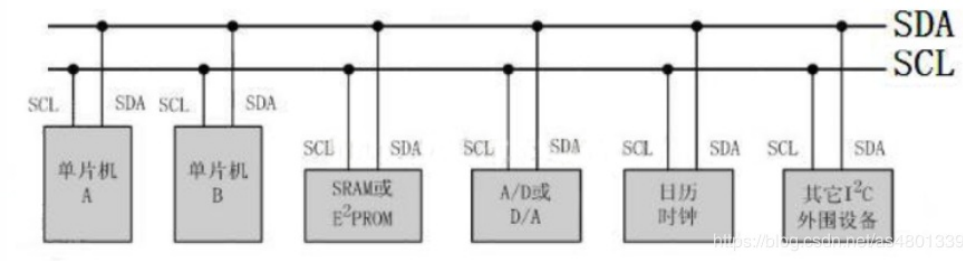
在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送，高速 IIC 总线一般可达 400kbps 以上。

PS： **这里要注意IIC是为了与低速设备通信而发明的，所以IIC的传输速率比不上SPI**

ＩＩＣ的物理层

IIC一共有只有两个总线： 一条是双向的数据线ＳＤＡ，一条是串行时钟线ＳＣＬ

所有接到I2C总线设备上的串行数据SDA都接到总线的SDA上，各设备的时钟线SCL接到总线的SCL上。I2C总线上的每一个设备都对应一个唯一的地址。

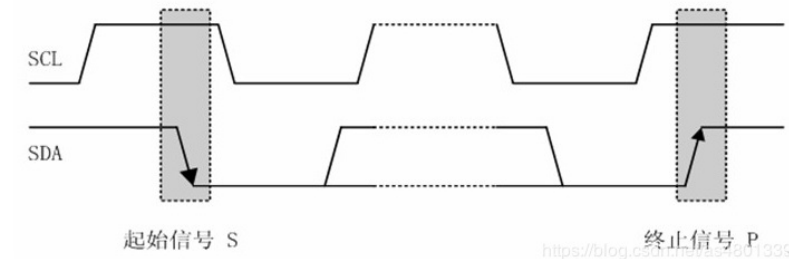


关于IIC的讲解，已经单独整理了一篇文章：

《IIC原理超详细讲解—值得一看》。
如果对IIC还不是太了解的朋友请移步到这篇文章中

IIC起始信号和终止信号：

起始信号：SCL保持高电平，SDA由高电平变为低电平后，延时(>4.7us)，SCL变为低电平。
停止信号：SCL保持高电平。SDA由低电平变为高电平。

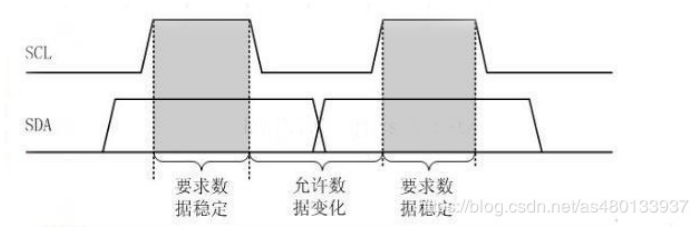


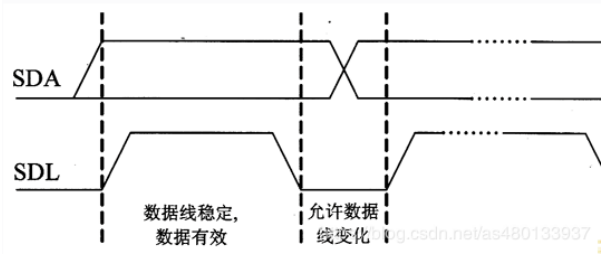
数据有效性

IIC信号在数据传输过程中，当SCL=1高电平时，数据线SDA必须保持稳定状态，不允许有电平跳变，只有在时钟线上的信号为低电平期间，数据线上的高电平或低电平状态才允许变化。

SCL=1时 数据线SDA的任何电平变换会看做是总线的起始信号或者停止信号。

也就是在IIC传输数据的过程中，SCL时钟线会频繁的转换电平，以保证数据的传输



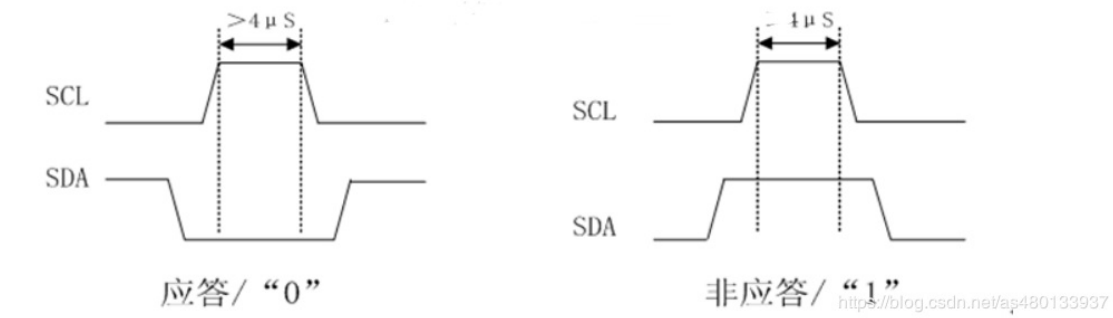


应答信号

每当主机向从机发送完一个字节的数据，主机总是需要等待从机给出一个应答信号，以确认从机是否成功接收到了数据，

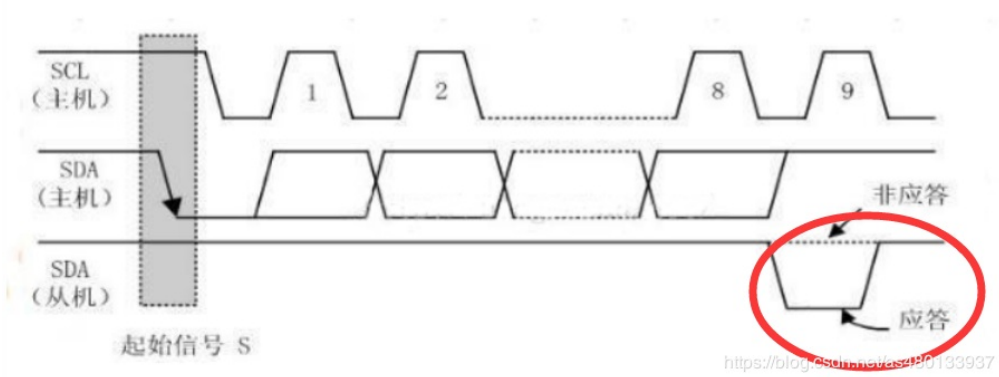
应答信号：主机SCL拉高，读取从机SDA的电平，为低电平表示产生应答

应答信号为低电平时，规定为有效应答位（ACK，简称应答位），表示接收器已经成功地接收了该字节；
应答信号为高电平时，规定为非应答位（NACK），一般表示接收器接收该字节没有成功。



每发送一个字节（8个bit）在一个字节传输的8个时钟后的第九个时钟期间，接收器接收数据后必须回一个ACK应答信号给发送器，这样才能进行数据传输。

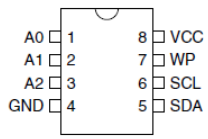
应答出现在每一次主机完成8个数据位传输后紧跟着的时钟周期，低电平0表示应答，1表示非应答，



这里我们仅介绍基于AT24C02的IIC通信

以AT24C02为例子

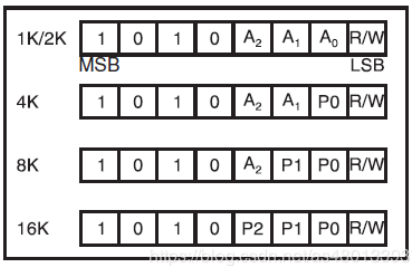
24C02是一个2K Bit的串行EEPROM存储器（掉电不丢失），内部含有256个字节。在24C02里面有一个8字节的页写缓冲器。



A0,A1,A2：硬件地址引脚
WP：写保护引脚，接高电平只读，接地允许读和写
SCL和SDA：IIC总线
可

以看出对于不同大小的24Cxx，具有不同的从器件地址。由于24C02为2k容量，也就是说只需要参考图中第一行的内容：

Device Address



芯片的寻址：
AT24C设备地址如下，前四位固定为1010，**A2~A0为由管脚电平。AT24CXX EEPROM Board模块中默认为接地。所以A2~A0默认为000**，最后一位表示读写操作。所以AT24Cxx的读地址为0xA1,写地址为0xA0。

也就是说如果是
写24C02的时候，从器件地址为10100000（0xA0）；
读24C02的时候，从器件地址为10100001（0xA1）。

片内地址寻址：

芯片寻址可对内部256B中的任一个进行读/写操作，其寻址范围为00~FF，共256个寻址单位。

对应的修改 A2A1A0 三位数据即可

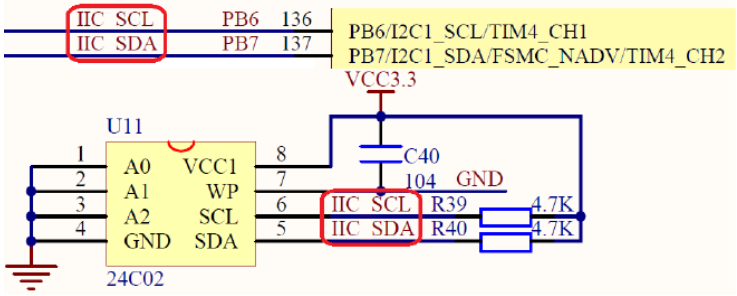
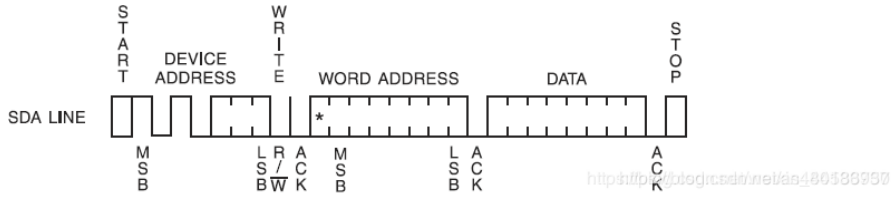


图 28.2.1 STM32 与 24C02 连接图

向AT24C02中写数据

Figure 8. Byte Write



操作时序：

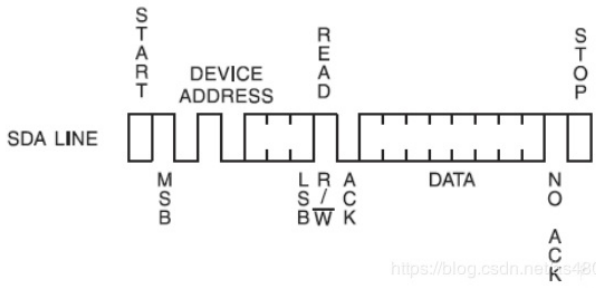
MCU先发送一个开始信号(START)启动总线
接着跟上首字节，发送器件写操作地址(DEVICE ADDRESS)+写数据(0xA0)
等待应答信号(ACK)
发送数据的存储地址。24C02一共有256个字节的存储空间，地址从0x00~0xFF，想把数据存储>在哪个位置，此刻写的就是哪个地址。
发送要存储的数据第一字节、第二字节、…注意在写数据的过程中，E2PROM每个字节都会>回应一个“应答位0”，老告诉我们写E2PROM数据成功，如果没有回应答位，说明写入不成功。
发送结束信号（STOP）停止总线

注意：
在写数据的过程中，每成功写入一个字节，E2PROM存储空间的地址就会自动加1，当加到0xFF后，再写一个字节，地址就会溢出又变成0x00。

写数据的时候需要注意，E2PROM是先写到缓冲区，然后再“搬运到”到掉电非易失区。所以这个过程需要一定的时间，AT24C02这个过程是不超过5ms！
所以，**当我们在写多个字节时，写入一个字节之后，再写入下一个字节之前，必须延时5ms才可以**

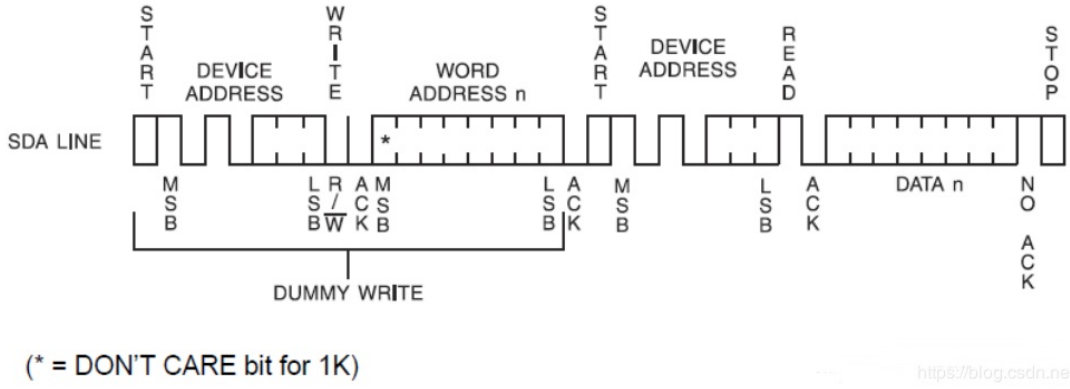
从AT24C02中读数据

读当前地址的数据



2、读随机地址的数据

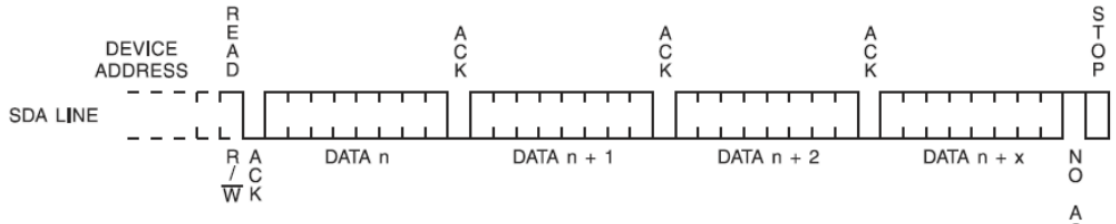
Figure 11. Random Read



MCU先发送一个开始信号(START)启动总线
接着跟上首字节，发送器件写操作地址(DEVICE ADDRESS)+写数据(0xA0)
注意：这里写操作是为了要把所要读的数据的存储地址先写进去，告诉E2PROM要读取哪个地址的数据。
发送要读取内存的地址(WORD ADDRESS)，通知E2PROM读取要哪个地址的信息。
重新发送开始信号(START)
发送设备读操作地址(DEVICE ADDRESS)对E2PROM进行读操作 (0xA1)
E2PROM会自动向主机发送数据，主机读取从器件发回的数据，在读一个字节后，MCU会回应一个应答信号(ACK)后，E2PROM会继续传输下一个地址的数据，MCU不断回应应答信号可以不断读取内存的数据
如果不想读了，告诉E2PROM不想要数据了，就发送一个“非应答位NAK(1)”。发送结束信号（STOP）停止总线

3、连续读数据

Figure 12. Sequential Read

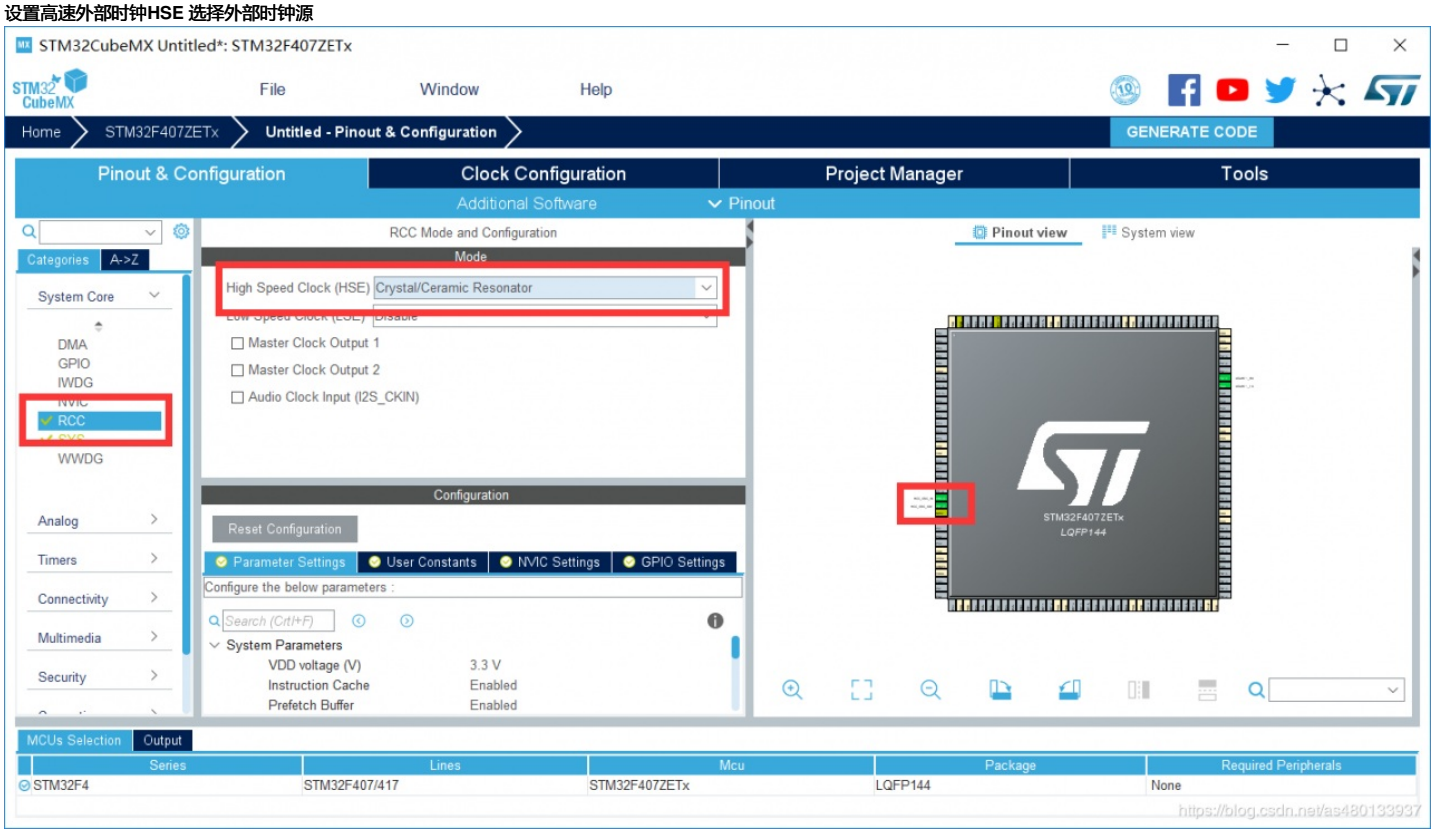


E2PROM支持连续写操作，操作和单个字节类似，先发送设备写操作地址(DEVICE ADDRESS)，然后发送内存起始地址(WORD ADDRESS)，MCU会回应一个应答信号(ACK)后，E2PROM会继续传输下一个地址的数据，MCU不断回应应答信号可以不断读取内存的数据。E2PROM的地址指针会自动递增，数据会依次保存在内存中。不应答发送结束信号后终止传输。

基于CubeMx的讲解

1设置RCC时钟

设置高速外部时钟HSE 选择外部时钟源



2 IIC设置

点击I2C1 设置为I2C 因为我们的硬件IIC 芯片一般都是主设备，也就是一般情况设置主模式即可

Master features 主模式特性

I2C Speed Mode： IIC模式设置 快速模式和标准模式。实际上也就是速率的选择。

I2C Clock Speed：I2C传输速率，默认为100KHz

Slave features 从模式特性

Clock No Stretch Mode： 时钟没有扩展模式

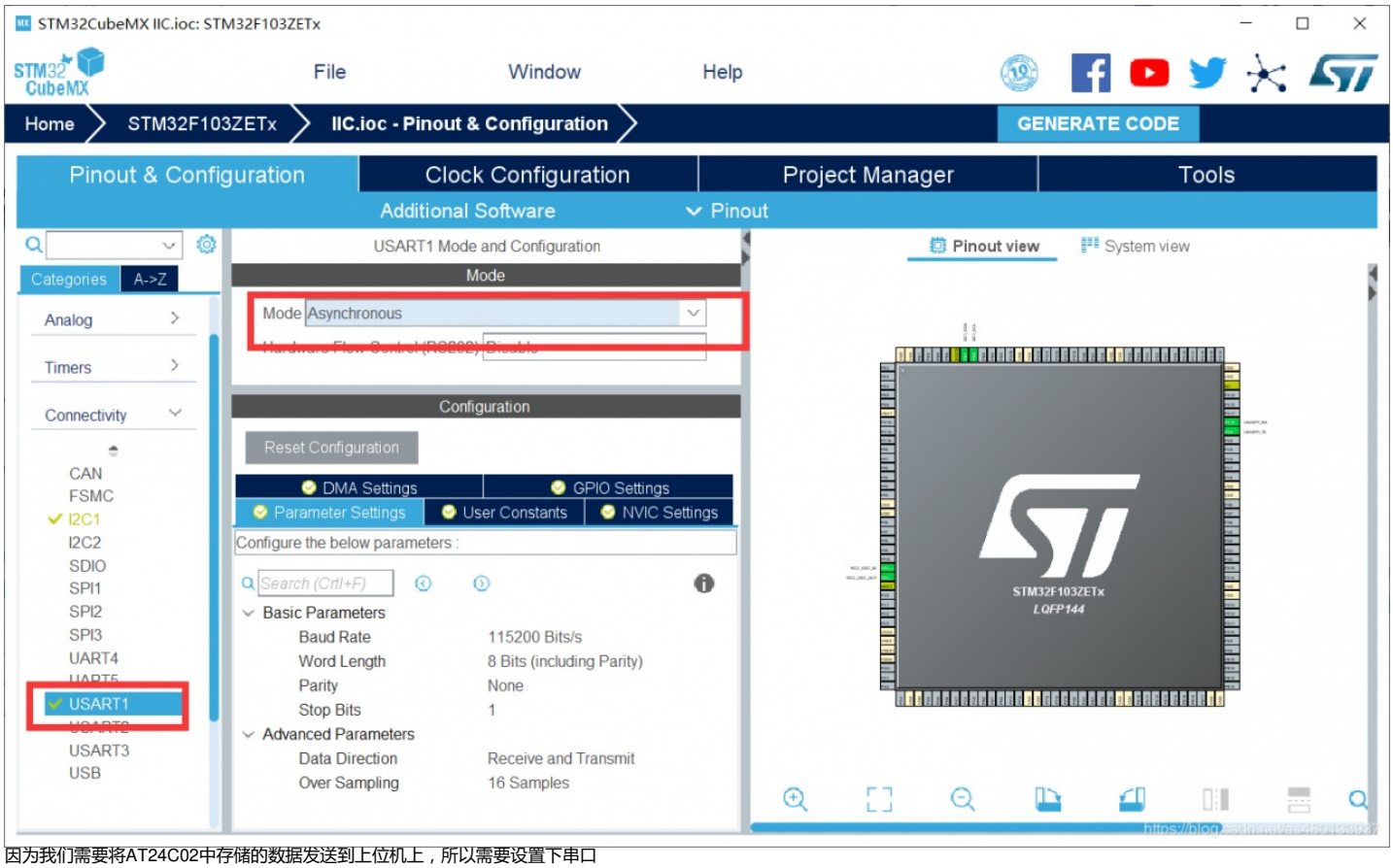
IIC时钟拉伸(Clock stretching)
clock stretching通过将SCL线拉低来暂停一个传输.直到释放SCL线为高电平,传输才继续进行.clock stretching是可选的,实际上大多数从设备不包括SCL驱动,所以它们不能stretch时钟.

Primary Address Length selection： 从设备地址长度 设置从设备的地址是7bit还是10bit 大部分为7bit
–Dual Address Acknowledged： 双地址确认

Primary slave address： 从设备初始地址

这里我们保持默认即可

3 串口设置

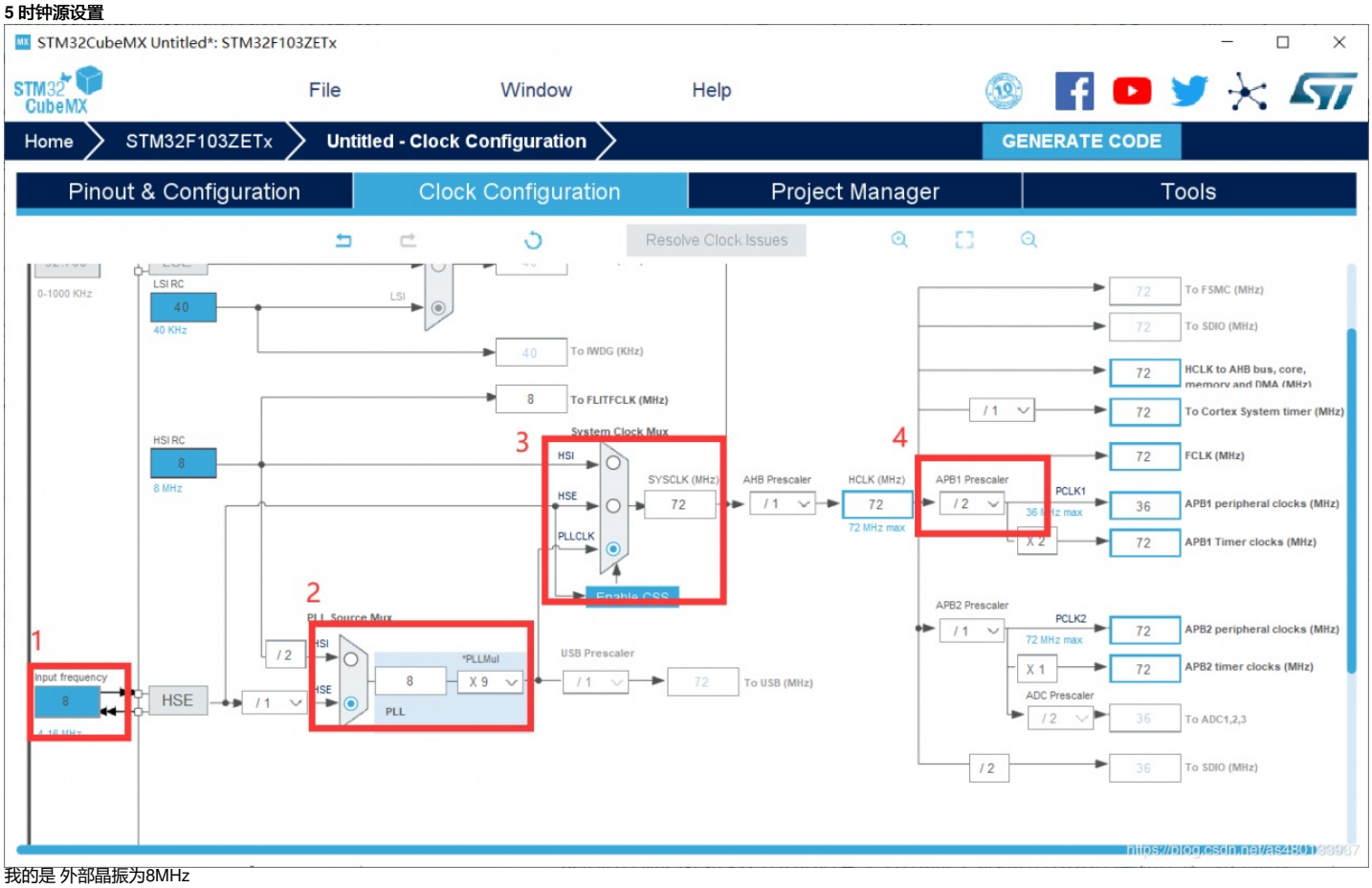


因为我们需要将AT24C02中存储的数据发送到上位机上，所以需要设置下串口

这里设置为异步通信，其他的默认即可

串口如有不懂，请看这篇文章
【STM32】HAL库 STM32CubeMX教程四—UART串口通信详解

5 时钟源设置

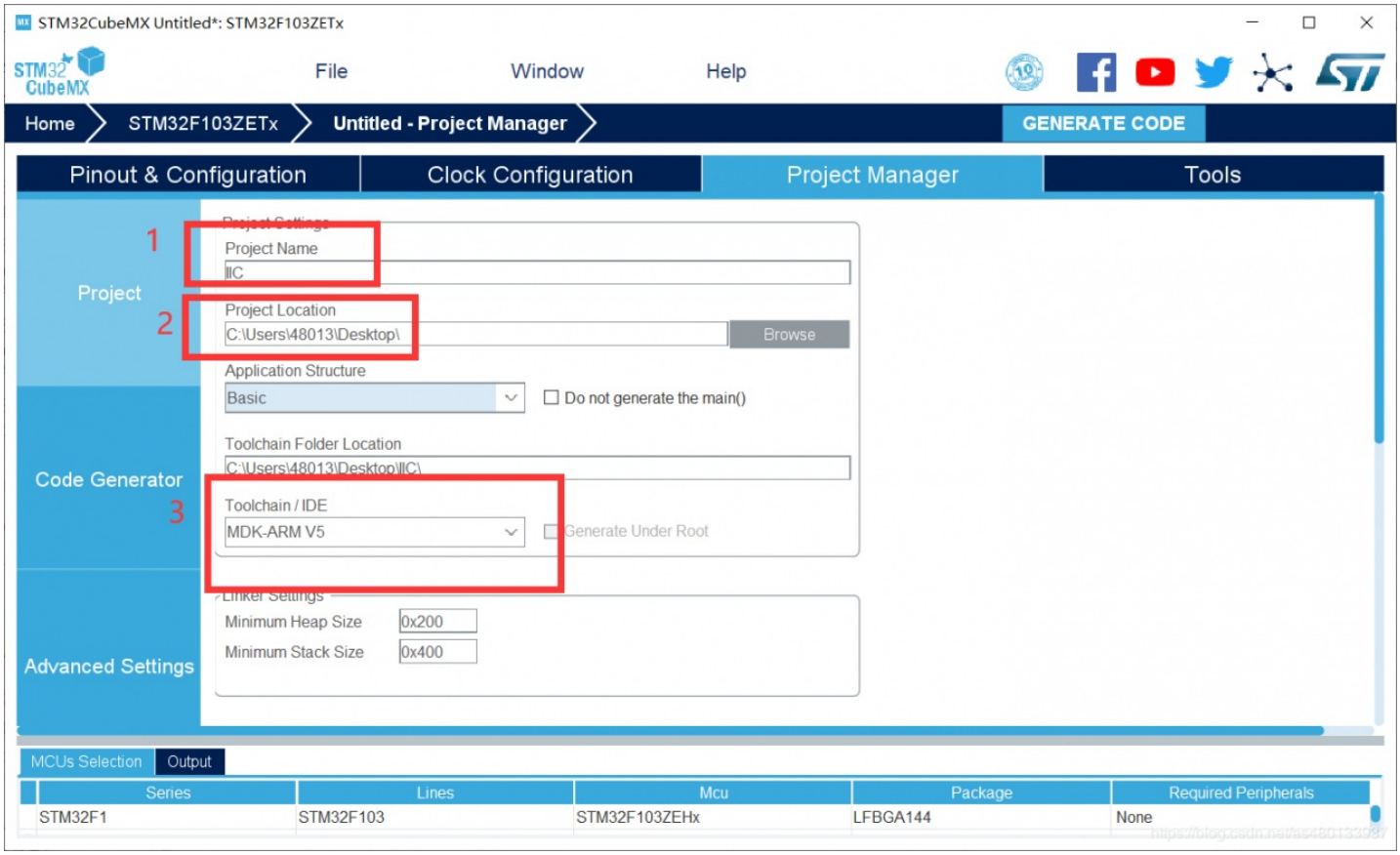


我的是 外部晶振为8MHz

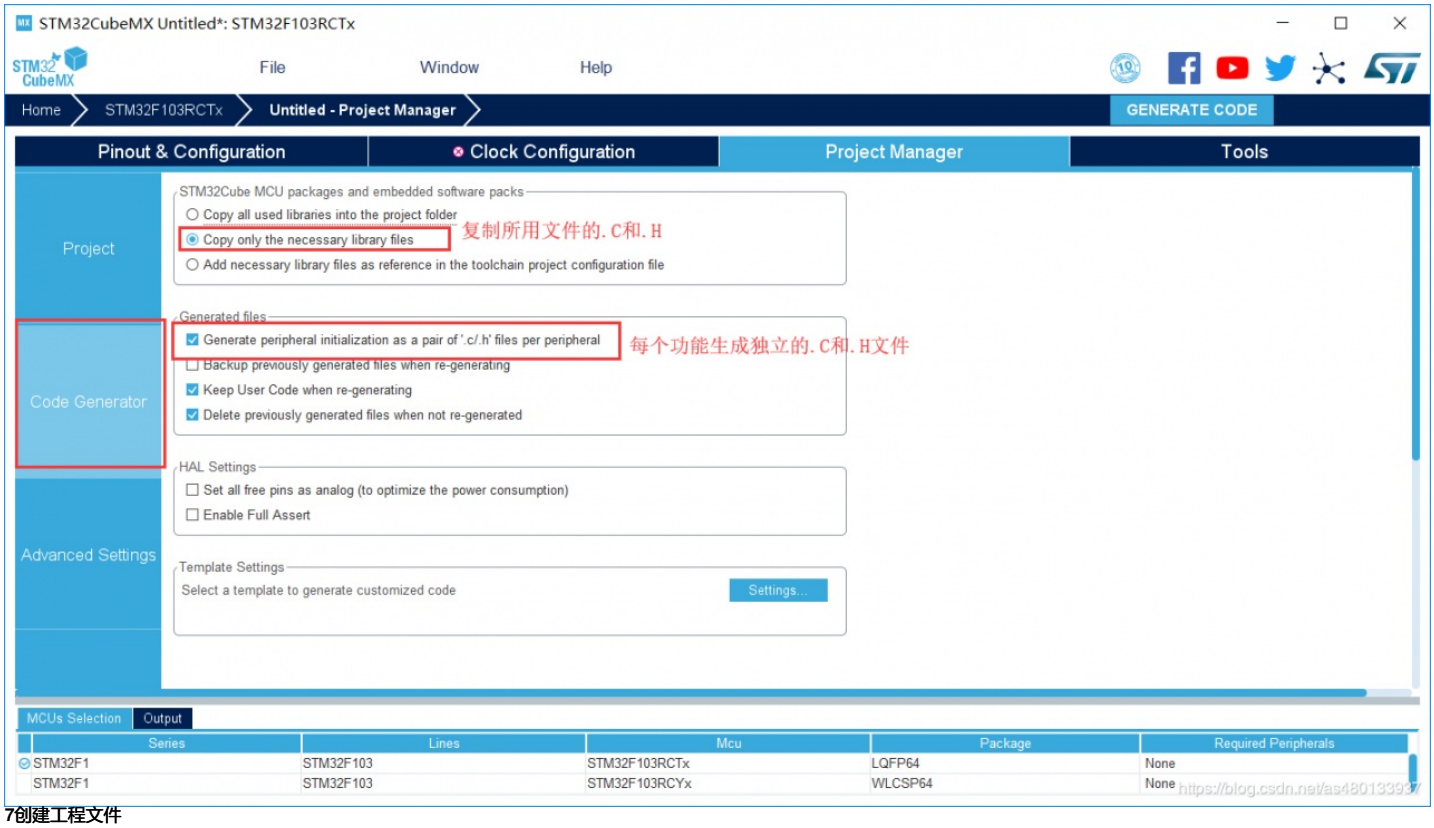
1选择外部时钟HSE 8MHz
2PLL锁相环倍频9倍
3系统时钟来源选择为PLL
4设置APB1分频器为 /2
5 使能CSS监视时钟

32的时钟树框图 如果不懂的话请看《【STM32】系统时钟RCC详解(超详细，超全面)》

6 项目文件设置



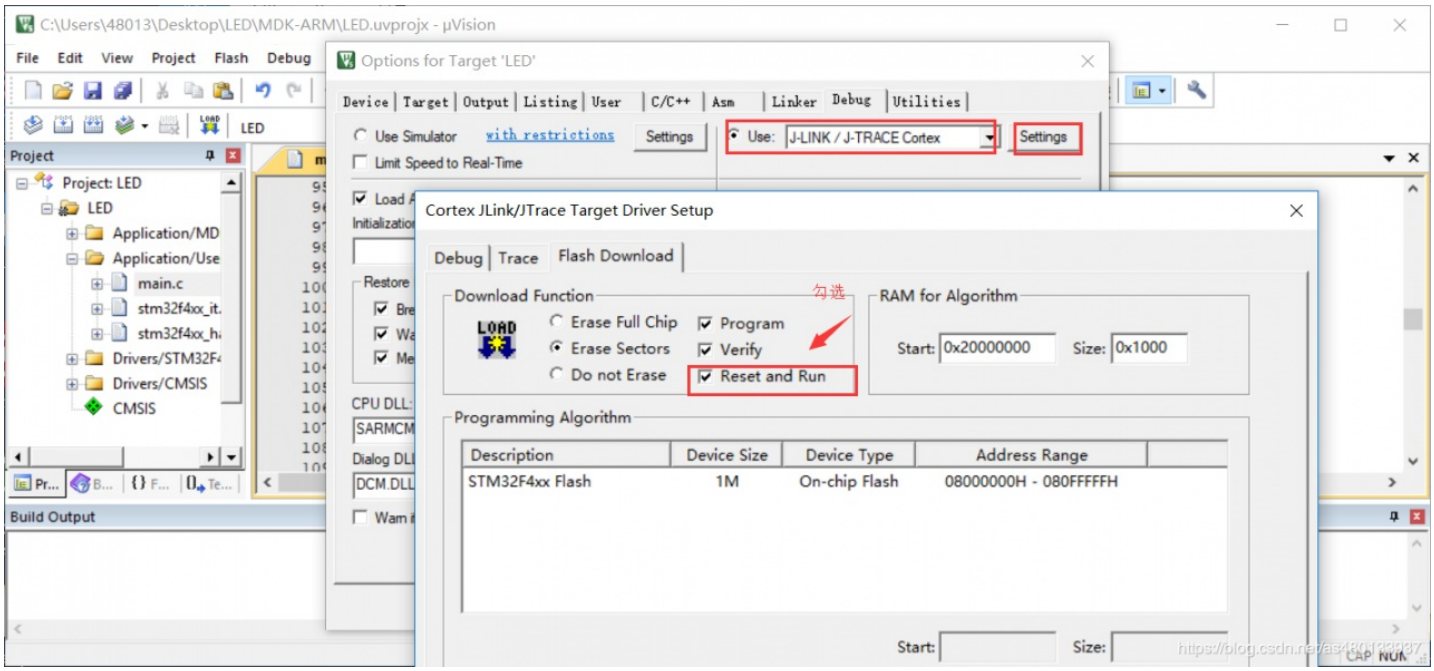
1 设置项目名称
2 设置存储路径
3 选择所用IDE



7创建工程文件

然后点击GENERATE CODE 创建工程

配置下载工具
新建的工程所有配置都是默认的 我们需要自行选择下载模式，勾选上下载后复位运行



IIC HAL库代码部分

在i2c.c文件中可以看到IIC初始化函数。在stm32f1xx_hal_i2c.h头文件中可以看到I2C的操作函数。分别对应**轮询，中断和DMA三种控制方式**

main.c | usart.c | stm32f1xx_hal.c | stm32f1xx_hal.h | usart.h | i2c.c | stm32f1xx_hal_i2c.c | stm32f1xx_hal_i2c.h

```
/** @addtogroup I2C_Exported_Functions_Group2 Input and Output operation functions
  * @{
  */
/******* Blocking mode: Polling */
HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);
HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);
HAL_StatusTypeDef HAL_I2C_Slave_Transmit(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size, uint32_t Timeout);
HAL_StatusTypeDef HAL_I2C_Slave_Receive(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size, uint32_t Timeout);
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);
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);
HAL_StatusTypeDef HAL_I2C_IsDeviceReady(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint32_t Trials, uint32_t Timeout);

/******* Non-Blocking mode: Interrupt */
HAL_StatusTypeDef HAL_I2C_Master_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Master_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Slave_Transmit_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Slave_Receive_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Mem_Write_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Mem_Read_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData, uint16_t Size);

HAL_StatusTypeDef HAL_I2C_Master_Seq_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t XferOptions);
HAL_StatusTypeDef HAL_I2C_Master_Seq_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t XferOptions);
HAL_StatusTypeDef HAL_I2C_Slave_Seq_Transmit_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size, uint32_t XferOptions);
HAL_StatusTypeDef HAL_I2C_Slave_Seq_Receive_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size, uint32_t XferOptions);
HAL_StatusTypeDef HAL_I2C_EnableListen_IT(I2C_HandleTypeDef *hi2c);
HAL_StatusTypeDef HAL_I2C_DisableListen_IT(I2C_HandleTypeDef *hi2c);
HAL_StatusTypeDef HAL_I2C_Master_Abort_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress);

/******* Non-Blocking mode: DMA */
HAL_StatusTypeDef HAL_I2C_Master_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Master_Receive_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Slave_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Slave_Receive_DMA(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Mem_Write_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Mem_Read_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData, uint16_t Size);

HAL_StatusTypeDef HAL_I2C_Master_Seq_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t XferOptions);
HAL_StatusTypeDef HAL_I2C_Master_Seq_Receive_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t XferOptions);
HAL_StatusTypeDef HAL_I2C_Slave_Seq_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size, uint32_t XferOptions);
HAL_StatusTypeDef HAL_I2C_Slave_Seq_Receive_DMA(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size, uint32_t XferOptions);
/**
  * @}
  */
```


上面的函数看起来多，但是只是发送和接收的方式改变了，函数的参数和本质功能并没有改变
比方说IIC发送函数 还是发送函数，只不过有普通发送，DMA传输，中断 的几种发送模式

这里我们仅介绍下普通发送，其他的只是改下函数名即可

IIC写函数

HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);

功能：**IIC写数据**
参数：

***hi2c** 设置使用的是那个IIC 例：&hi2c2
DevAddress 写入的地址 设置写入数据的地址 例 0xA0
***pData** 需要写入的数据
Size 要发送的字节数

Timeout 最大传输时间，超过传输时间将自动退出传输函数

IIC读函数

HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);

功能：**IIC读一个字节**
参数：

***hi2c**： 设置使用的是那个IIC 例：&hi2c2

DevAddress： 写入的地址 设置写入数据的地址 例 0xA0

***pDat：a 存储读取到的数据**

Size： 发送的字节数

Timeout： **最大读取时间**，超过时间将自动退出读取函数

举例：

```
HAL_I2C_Master_Transmit(&hi2c1,0xA1,(uint8_t*)TxData,2,1000) ; ;
```

发送两个字节数据

IIC写数据函数

```
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);
/* 第1个参数为I2C操作句柄
第2个参数为从机设备地址
第3个参数为从机寄存器地址
第4个参数为从机寄存器地址长度
第5个参数为发送的数据的起始地址
第6个参数为传输数据的大小
第7个参数为操作超时时间       */
```

功能：**IIC写多个数据** 该函数适用于IIC外设里面还有子地址寄存器的设备，比方说E2PROM,除了设备地址，每个存储字节都有其对应的地址

参数：

***hi2c：** I2C设备号指针，设置使用的是那个IIC 例：&hi2c2

DevAddress： 从设备地址 从设备的IIC地址 例E2PROM的设备地址 0xA0

MemAddress： 从机寄存器地址 ，每写入一个字节数据，地址就会自动+1

MemAddSize： 从机寄存器地址字节长度 8位或16位

写入数据的字节类型 **8位还是16位**

I2C_MEMADD_SIZE_8BIT

I2C_MEMADD_SIZE_16BIT

在stm32f1xx_hal_i2c.h中有定义

```
/** @defgroup I2C_Memory_Address_Size I2C Memory Address Size
  * @{
  */
#define I2C_MEMADD_SIZE_8BIT            0x00000001U
#define I2C_MEMADD_SIZE_16BIT           0x00000010U
/**
```

***pData：** **需要写入的的数据的起始地址**

Size： 传输数据的大小 多少个字节

Timeout： **最大读取时间**，超过时间将自动退出函数

使用**HAL_I2C_Mem_Write**等于先使用**HAL_I2C_Master_Transmit**传输第一个寄存器地址，再用**HAL_I2C_Master_Transmit**传输写入第一个寄存器的数据。可以传输多个数据

```
void Single_WriteI2C(uint8_t REG_Address,uint8_t REG_data)
{
uint8_t TxData[2] = {REG_Address,REG_data};
while(HAL_I2C_Master_Transmit(&hi2c1,I2C1_WRITE_ADDRESS,(uint8_t*)TxData,2,1000) != HAL_OK)
{
if (HAL_I2C_GetError(&hi2c1) != HAL_I2C_ERROR_AF)
{
Error_Handler();
}
}
}
```

在传输过程，寄存器地址和源数据地址是会自加的。

至于读函数也是如此，因此用HAL_I2C_Mem_Write和HAL_I2C_Mem_Read，来写读指定设备的指定寄存器数据是十分方便的，让设计过程省了好多步骤。

举例：

8位：

```
HAL_I2C_Mem_Write(&hi2c2, ADDR, i, I2C_MEMADD_SIZE_8BIT,&(I2C_Buffer_Write[i]),8, 1000);
```

HAL_I2C_Mem_Read(&hi2c2, ADDR, i, I2C_MEMADD_SIZE_8BIT,&(I2C_Buffer_Write[i]),8, 1000);

16位：

```
HAL_I2C_Mem_Write(&hi2c2, ADDR, i, I2C_MEMADD_SIZE_16BIT,&(I2C_Buffer_Write[i]),8, 1000);
```

HAL_I2C_Mem_Read(&hi2c2, ADDR, i, I2C_MEMADD_SIZE_16BIT,&(I2C_Buffer_Write[i]),8, 1000);

如果只往某个外设中写数据，则用Master_Transmit。 如果是外设里面还有子地址，例如我们的E2PROM，有设备地址，还有每个数据的寄存器存储地址。则用Mem_Write。Mem_Write是2个地址，Master_Transmit只有从机地址

硬件IIC读取AT24C02

在**mian.c**文件前面声明，AT24C02 写地址和读地址 ，定义写数据数组，和读数据数组

```
/* USER CODE BEGIN PV */
#include

#define ADDR_24LCxx_Write 0xA0
#define ADDR_24LCxx_Read 0xA1
#define BufferSize 256
uint8_t WriteBuffer[BufferSize],ReadBuffer[BufferSize];
uint16_t i;
/* USER CODE END PV */
```

重新定义printf函数

在 **stm32f4xx_hal.c**中包含#include

```
#include "stm32f4xx_hal.h"
#include
extern UART_HandleTypeDef huart1; //声明串口
```

在 **stm32f4xx_hal.c** 中重写fget和fput函数

```
/**
* 函数功能: 重定向c库函数printf到DEBUG_USARTx
* 输入参数: 无
* 返 回 值: 无
* 说 明 : 无
*/
int fputc(int ch, FILE *f)
{
HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xffff);
return ch;
}

/**
* 函数功能: 重定向c库函数getchar,scanf到DEBUG_USARTx
* 输入参数: 无
* 返 回 值: 无
* 说 明 : 无
*/
int fgetc(FILE *f)
{
uint8_t ch = 0;
HAL_UART_Receive(&huart1, &ch, 1, 0xffff);
return ch;
}
```

在**main.c**中添加

```
/* USER CODE BEGIN 2 */
for(i=0; i<256; i++)
WriteBuffer[i]=i; /* WriteBuffer init */

printf("\r\n***************I2C Example Z小旋测试*********************************\r\n");
for (int j=0; j<32; j++)
{
if(HAL_I2C_Mem_Write(&hi2c1, ADDR_24LCxx_Write, 8*j, I2C_MEMADD_SIZE_8BIT,WriteBuffer+8*j,8, 1000) == HAL_OK)
{
printf("\r\n EEPROM 24C02 Write Test OK \r\n");
HAL_Delay(20);
}
else
{
HAL_Delay(20);
printf("\r\n EEPROM 24C02 Write Test False \r\n");
}
}
/*
// wrinte date to EEPROM 如果要一次写一个字节，写256次，用这里的代码
for(i=0;i<BufferSize;i++)
{
HAL_I2C_Mem_Write(&hi2c1, ADDR_24LCxx_Write, i, I2C_MEMADD_SIZE_8BIT,&WriteBuffer[i],1，0xff);//使用I2C块读，出错。因此采用此种方式，逐个单字节写入
HAL_Delay(5);//此处延时必加，与AT24C02写时序有关
}
printf("\r\n EEPROM 24C02 Write Test OK \r\n");
*/

HAL_I2C_Mem_Read(&hi2c1, ADDR_24LCxx_Read, 0, I2C_MEMADD_SIZE_8BIT,ReadBuffer,BufferSize, 0xff);

for(i=0; i<256; i++)
printf("0x%02X ",ReadBuffer[i]);

/* USER CODE END 2 */
```

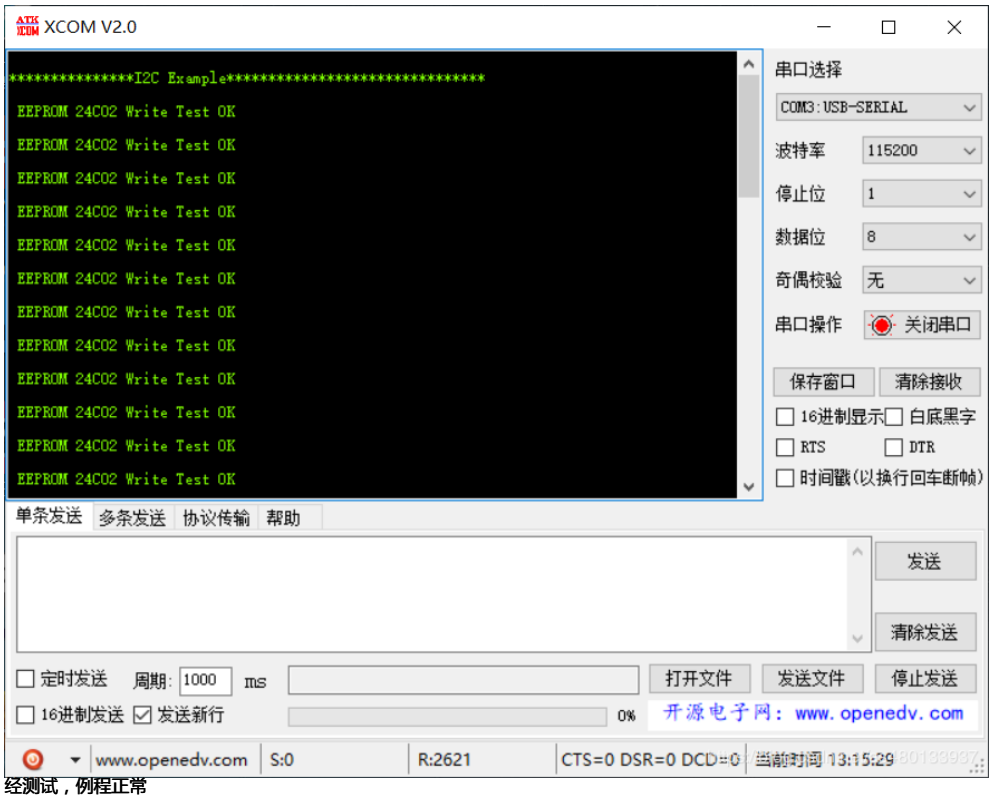
注意事项：

AT24C02的IIC每次写之后要延时一段时间才能继续写 每次写之后要delay 5ms左右 **不管硬件IIC采用何种形式（DMA，IT），都要确保两次写入的间隔大于5ms;**
读写函数最后一个超时调整为1000以上 因为我们一次写8个字节，延时要久一点
AT24C02页写入只支持8个byte，所以需要分32次写入。这不是HAL库的bug，而是AT24C02的限制，其他的EEPROM可以支持更多byte的写入。

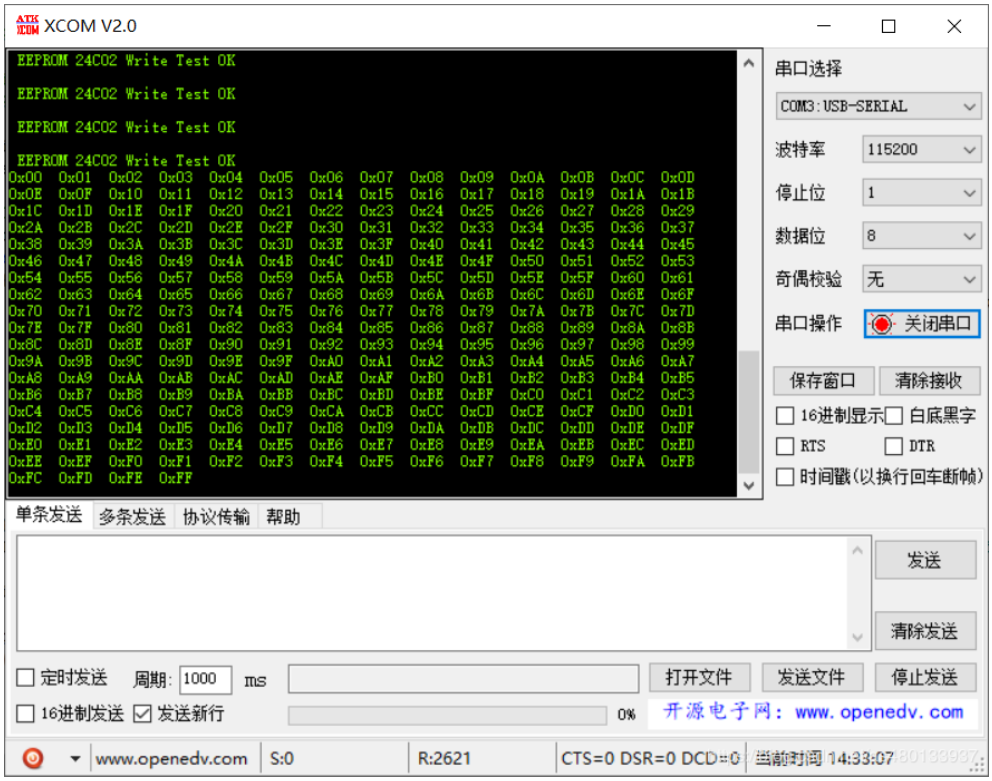
当然，你也可以每次写一个字节，分成256次写入，也是可以的 那就用注释了的代码即可

```
/*
// wrinte date to EEPROM 如果要一次写一个字节，写256次，用这里的代码
for(i=0;i<BufferSize;i++)
{
HAL_I2C_Mem_Write(&hi2c1, ADDR_24LCxx_Write, i, I2C_MEMADD_SIZE_8BIT,&WriteBuffer[i],1，0xff);//使用I2C块读，出错。因此采用此种方式，逐个单字节写入
HAL_Delay(5);//此处延时必加，与AT24C02写时序有关
}
printf("\r\n EEPROM 24C02 Write Test OK \r\n");
*/
```

注意读取AT24C02数据的时候延时也要久一点，否则会造成读的数据不完整



经测试，例程正常



点个赞 再走吧！

作者：Z小旋