STM32F767+STM32CubeMX I2C通信读写EEPROM数据（采用轮询、DMA、中断三种方式）

STM32F767+STM32CubeMX I2C通信读写EEPROM数据（采用轮询、DMA、中断三种方式）

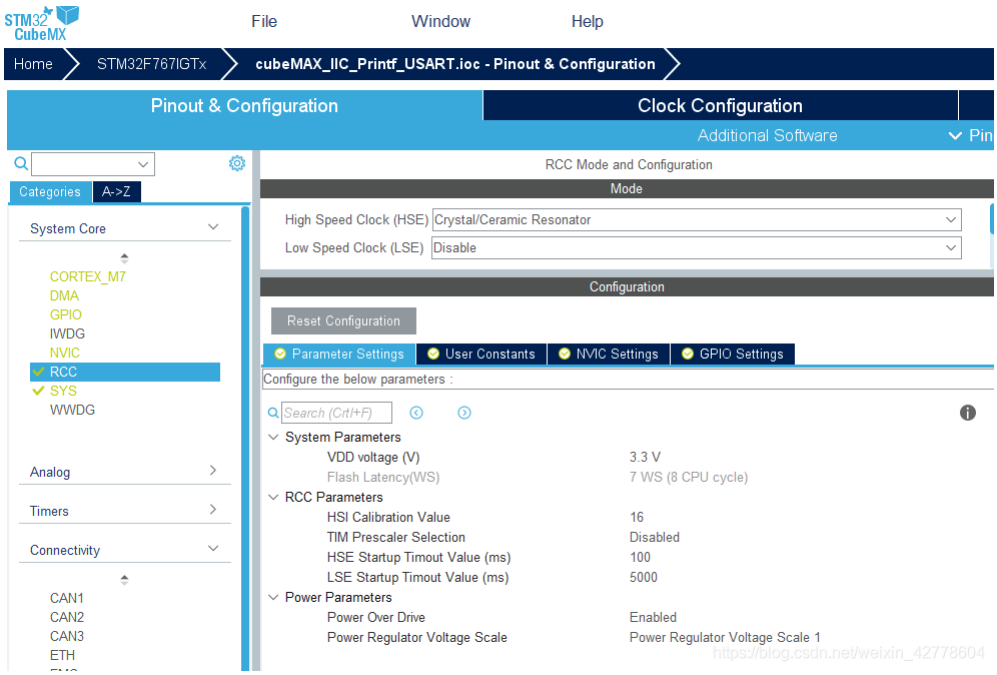
摘要-前言

作为一名STM32的初学者，在学习过程中会遇到很多问题，解决过程中会看到很多博主发过的文章，每次都是零零总总的学习各个大牛的经验。但时间久了就会忘记其中的一些关键点，所以才有了把自己解决问题的过程记录下来的想法，日后回忆起也很方便。
前人们做过很多STM32 I2C通信的努力，但大多都是基于STM32F0、F1、F4这些系列的板子，而众所周知不同系列之间还是有不同的，这就导致初学者学习STM32时，会遇到很多困难。另外I2C通信很多人采取的是软件模拟实现，对硬件并不看好。但是毕竟这么多年过去了，HAL库及CubeMX的出现，能够很大程度上解决I2C宕机的问题。所以本文除了讲解CubeMX I2C通信以外，也顺便做了实验来验证I2C的实际效果。

硬件设施：正点原子STM32F676阿波罗开发板
IDE：KEIL5
STM32CubeMX：5.4.0
STM32CubeMX Firmware Package Name and Version：STM32Cube FW_F7 V1.15.0
Keil STM32F767芯片包：Keil.STM32F7xx_DFP.2.12.0
EEPROM：24C02
I2C2-SCL：PH4
I2C2-SDA：PH5

轮询方式（普通方式）读写EEPROM（24C02）

配置RCC

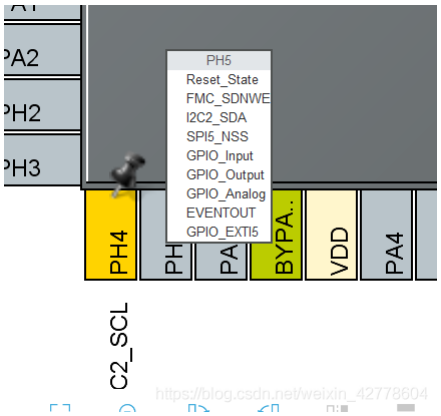


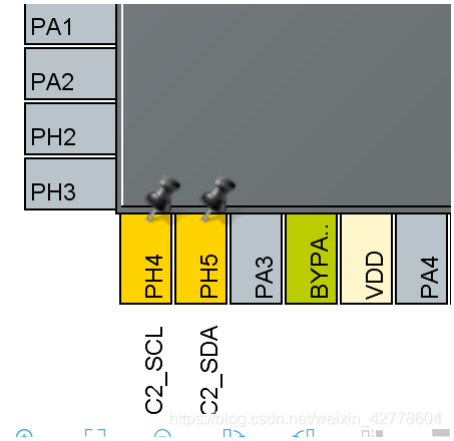
选择HSE的Crystal/Ceramic Resonator其余默认

配置I2C2

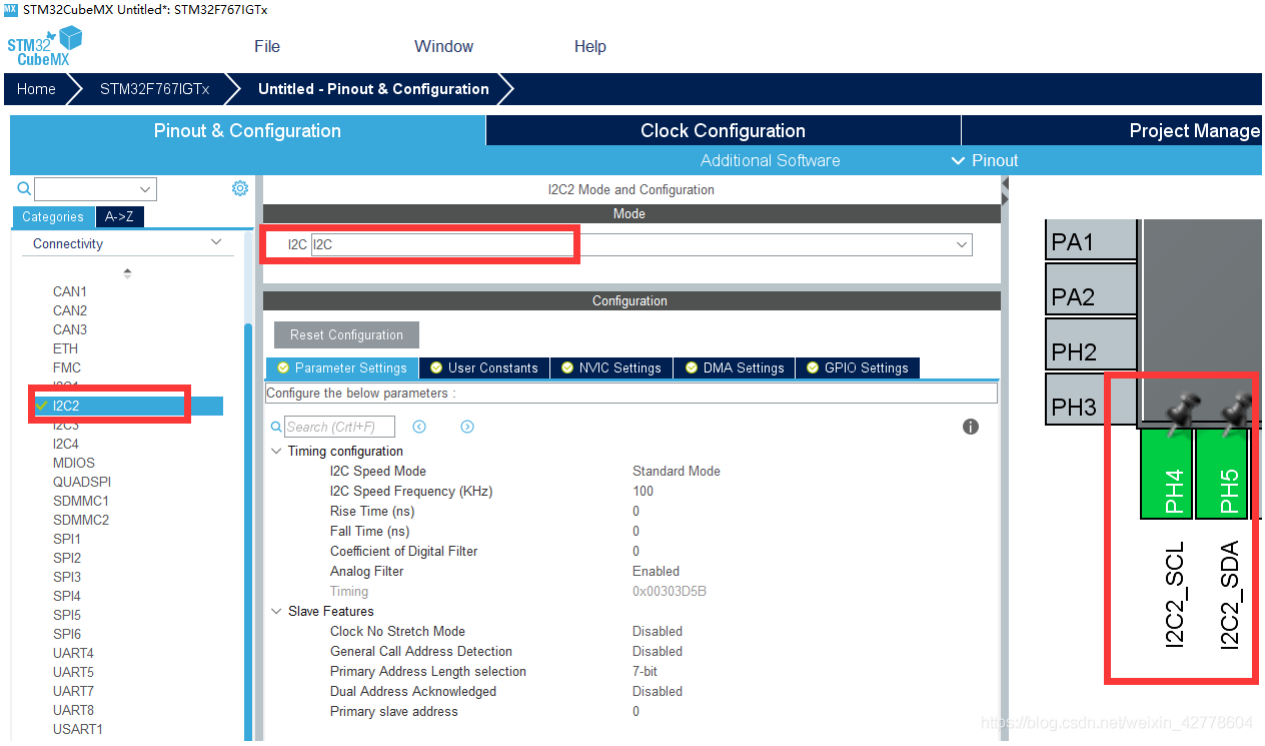
这里需要注意一下，CubeMX默认的I2C2不是PH4和PH5，是PF0和PF1。如果直接点击左侧选项中的I2C2，就自动成了默认PF0和PF1。正点原子开发板资料中虽然写了24C02连接在I2C2上，但是粗心的我并没有注意引脚号，在配置工程时，选择了默认，这一个小小的问题，浪费了我两天时间。

配置时，在右侧的芯片上找到PH4与PH5，左键引脚，选择I2C2_SCL和I2C2-SDA，此时两个引脚会变成黄色。

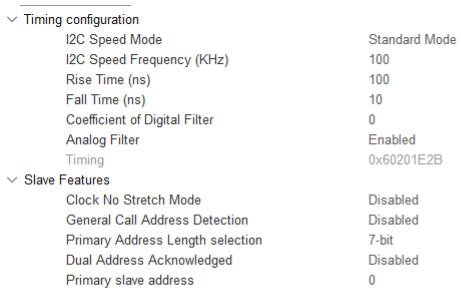




这个时候再去选中左侧的I2C2，就会定位到PH4和PH5了。



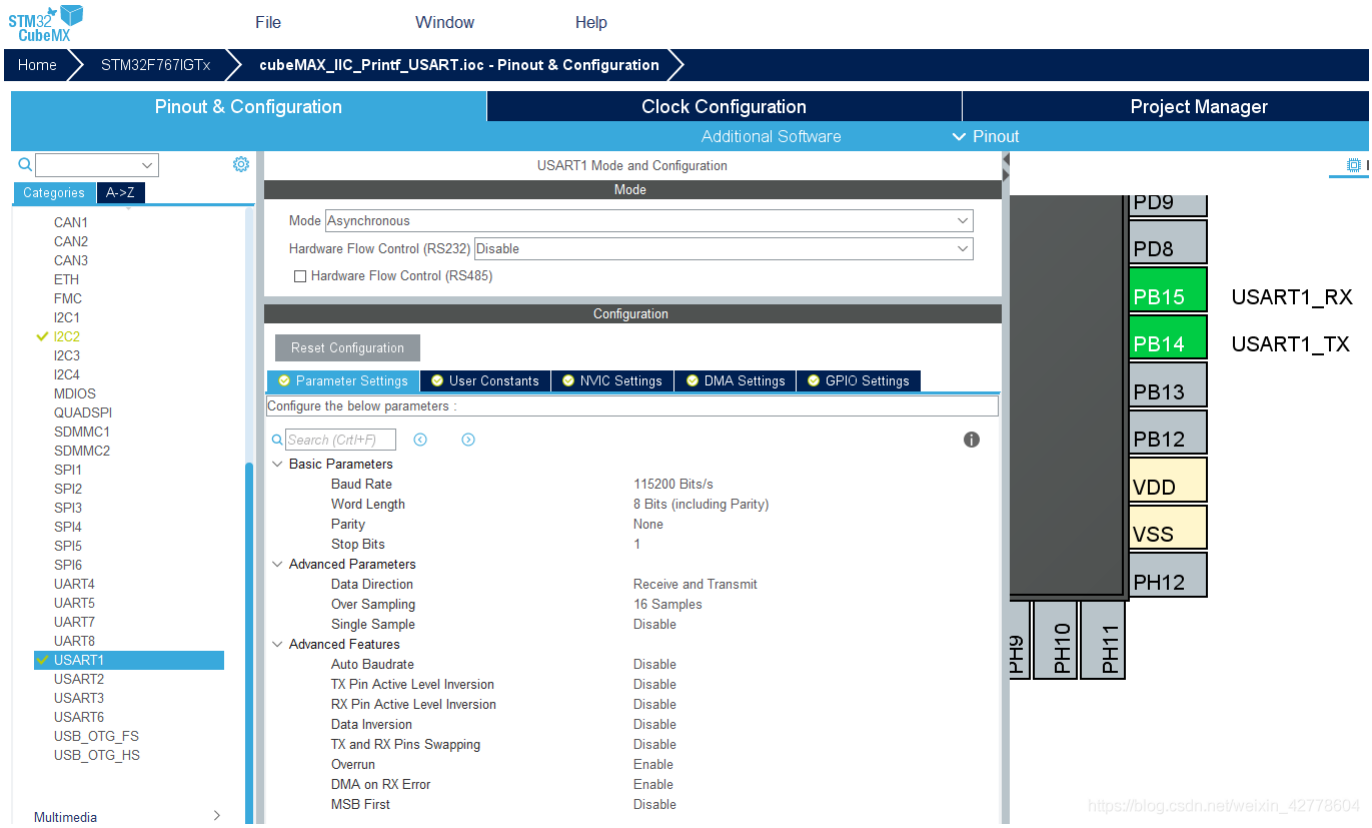
接下来再看下面的配置参数：



选择了标准模式，那么频率对应100KHZ。Rise Time、Fall Time、Coefficient of Digital Filter 实际上是要遵循一套非常复杂的时序计算方法的，也和对应的外设有关系，在设置前要阅读相关的外设资料此处暂时不展开。Timing由软件自动计算好，这也是CubeMX方便之处。

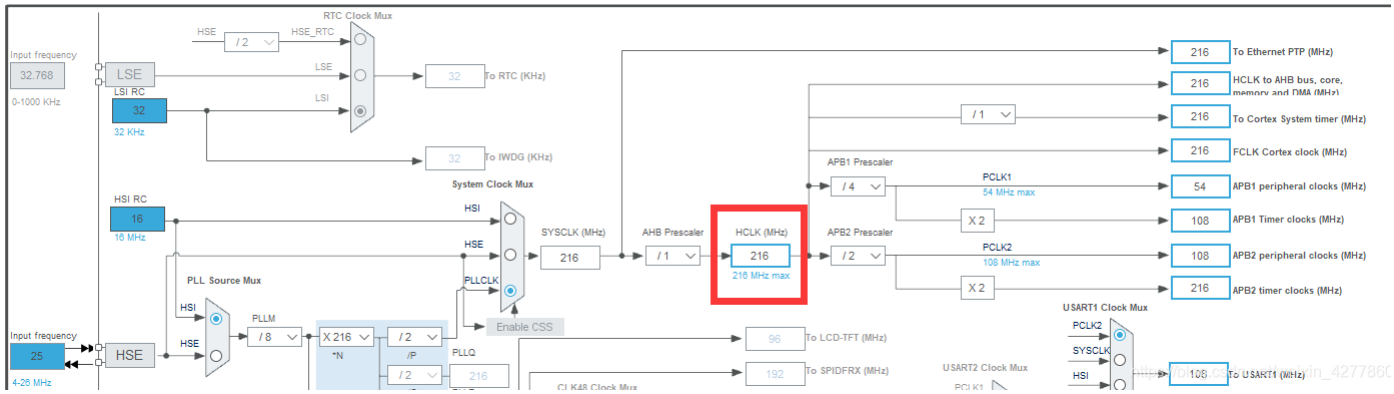
配置USART1

配置串口的目的，是为了能够把从EEPROM读出来的数据“打印”在串口调试助手上，方便检验。



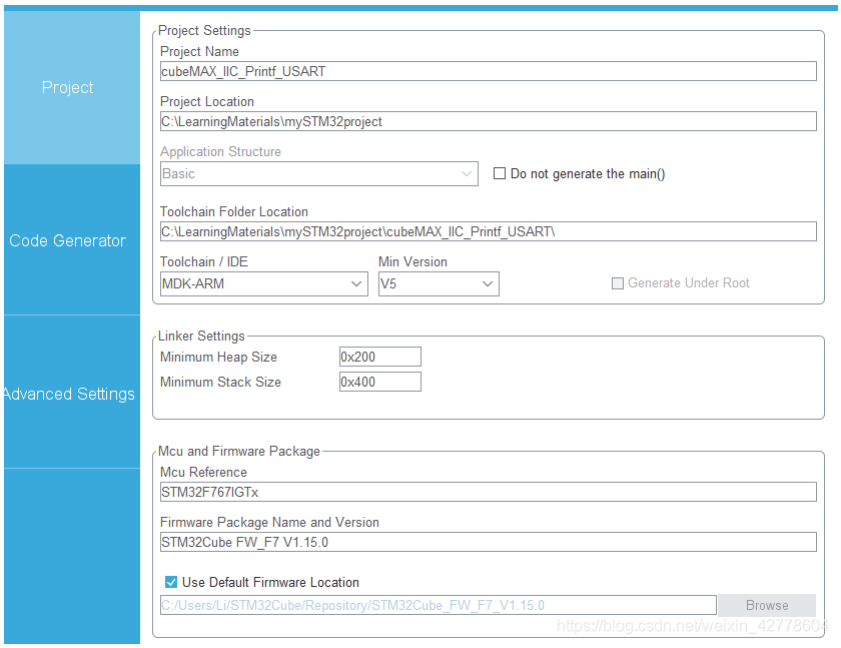
选择USART1,在选择Asynchronous，其余默认即可。

配置时钟树



直接把红色地方拉满即可。

工程管理



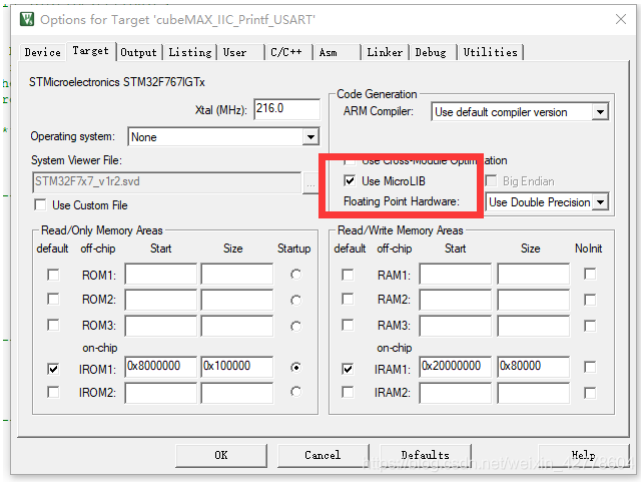
从串口打印汉字、英文、数字等可读性信息

配置完后，打开工程，在main.c中添加如下代码，对prinf函数进行重定位。

```
#include <stdio.h>

/* USER CODE BEGIN PFP */
#ifdef __GNUC__
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */
PUTCHAR_PROTOTYPE
{
 HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0x0001);
 return ch;
}
/* USER CODE END PFP */
```

勾选微库



然后就可以开开心心用Printf函数了。

I2C轮询方式的代码

```
/* USER CODE BEGIN PD */
#define ADDR_24LCxx_Write 0xA0
#define ADDR_24LCxx_Read 0xA1
#define BufferSize 0X100
/* USER CODE END PD */
```

24C02的写地址是0XA0,读地址为0XA1，总存储空间为256字节。

```
/* USER CODE BEGIN PV */
uint8_t WriteBuffer[BufferSize], ReadBuffer[BufferSize];
uint16_t i,j;
uint16_t recv;//保存I2C读写函数的返回值，方便DEBUG
/* USER CODE END PV */
```

查阅24C02的资料可知，该EEPROM总存储量256字节，按照页来存储，每页8个字节。因此每次写入只可以写8个字节，写满的话总共要分32次执行。参考资料同时指出，每次写入需要等待5ms之后才能进行下次写入。因此下面的程序我分了32次执行，每次存储8字节。
使用函数HAL_I2C_Mem_Write进行写入操作，该函数在用户手册中定义如下：

HAL_I2C_Mem_Write

Function name	**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)**
Function description	Write an amount of data in blocking mode to a specific memory address.
Parameters	• **hi2c:** Pointer to a I2C_HandleTypeDef structure that contains the configuration information for the specified I2C. • **DevAddress:** Target device address: The device 7 bits address value in datasheet must be shift at right before call interface • **MemAddress:** Internal memory address • **MemAddSize:** Size of internal memory address • **pData:** Pointer to data buffer • **Size:** Amount of data to be sent • **Timeout:** Timeout duration
Return values	• **HAL:** status

从描述中可以看出，这种传递方式是阻塞的，形象的说，CPU等在这里Timeout ms啥都不敢，就等写入或者读取数据，只有读写结束后程序才往下进行。这种方式令人很不舒服。尤其是在严格控制节拍的地方。

```
/* USER CODE BEGIN 2 */
  printf("Write data in EEPROM\r\n");
  for (i = 0; i < 256; i++)
   WriteBuffer[i] = i;
  for (j = 0; j < 32; j++)
  {
   if ((recv = HAL_I2C_Mem_Write(&hi2c2, ADDR_24LCxx_Write, 8 * j, I2C_MEMADD_SIZE_8BIT, WriteBuffer + 8 * j, 8)) == HAL_OK)
   {
    printf("\r\n EEPROM 24C02 Write Test OK \r\n");
    HAL_Delay(5);
   }
   else
   {
    HAL_Delay(5);
    printf("\r\n EEPROM 24C02 Write Test False \r\n");
    printf("\r\n recv = %d \r\n",recv);
   }
  }
   /* USER CODE END 2 */
```

在主函数中添加：使用HAL_I2C_Mem_Read函数进行一次性读取操作。参数意义和写函数一样。读取完后，打印出读到的结果。

```
  if ((recv = HAL_I2C_Mem_Read(&hi2c2, ADDR_24LCxx_Read, 0, I2C_MEMADD_SIZE_8BIT, ReadBuffer, BufferSize)) == HAL_OK)
       {
        printf("This is Data in EEPROM!!\r\n");
        for (i = 0; i < 256; i++)
         printf("%d", ReadBuffer[i]);
        printf("\r\nGOT Finished!!\r\n");
       }
       else
       {
        printf("Failed to get data\r\n");
        printf("recv =  %d !\r\n", recv);
       }
```

I2C轮询方式的实验结果

下载到单片机中，连接好串口线，打开串口调试助手，选好端口等参数，打开串口，给单片机上电，助手中出现以下内容：

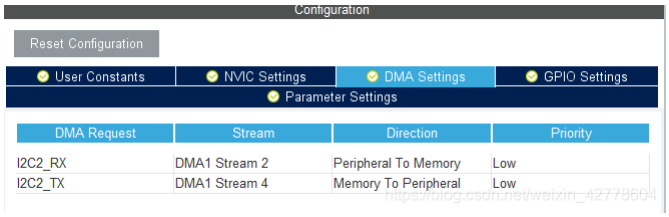
成功写入32次，然后读取到结果。

用DMA的方式，实现I2C通信

DMA方式实现I2C通信是非阻塞的，DMA控制器化身成为数据的搬运工，专门负责管理数据在内存外设之间传递。CPU把数据扔给DMA后，就撒手不管了，继续该干嘛干嘛。DMA则接盘数据，进行搬运工作。大大节约CPU的运行效率。

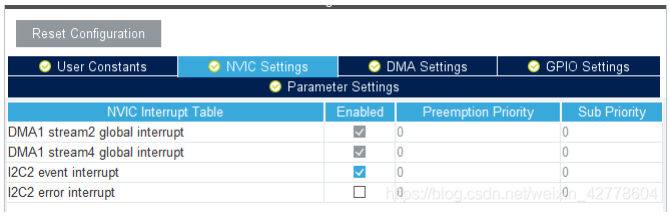
CubeMX配置I2C-DMA

在刚才的基础上，只需要修改两个地方：



Add接受和发送的DMA Request，参数默认就行了。
接下来就是关键的地方，一定要勾选I2C2 event interrupt，否则你只能进行一次不超过 256 Bytes 的 DMA，之后就得手动去清空标志位，再手动启动一次 DMA 发送。

> 参考这位大佬的原话：使用硬件 I2C + DMA 操作液晶屏 (STM32)



勾选 I2C2事件中断

代码修改-DMA方式实现I2C通信

重新生成代码，在新的代码中，只需要换一下函数。
写入操作：

```
(recv = HAL_I2C_Mem_Write_DMA(&hi2c2, ADDR_24LCxx_Write, 8 * j, I2C_MEMADD_SIZE_8BIT, WriteBuffer + 8 * j, 8)) == HAL_OK
```

读取操作：

```
(recv = HAL_I2C_Mem_Read_DMA(&hi2c2, ADDR_24LCxx_Read, 0, I2C_MEMADD_SIZE_8BIT, ReadBuffer, BufferSize)) == HAL_OK
```

HAL_I2C_Mem_Write_DMA函数和HAL_I2C_Mem_Read_DMA函数的参数含义和之前轮询方式的一致。只是少了Timeout这一项，因为CPU从此再也不需要等待了。

实验结果-DMA方式实现I2C通信

成功写入32次，然后读取到EEPROM中的数据。

用中断的方式，实现I2C通信

中断方式实现I2C通信也是非阻塞的，每次执行完一些特定事件之后，CPU会自动调取对应的中断回调函数，STM32的I2C定义的回调函数有：
void HAL_I2C_EV_IRQHandler (I2C_HandleTypeDef * hi2c);
void HAL_I2C_ER_IRQHandler (I2C_HandleTypeDef * hi2c);
void HAL_I2C_MasterTxCpltCallback (I2C_HandleTypeDef *hi2c);
void HAL_I2C_MasterRxCpltCallback (I2C_HandleTypeDef * hi2c);
void HAL_I2C_SlaveTxCpltCallback (I2C_HandleTypeDef * hi2c);
void HAL_I2C_SlaveRxCpltCallback (I2C_HandleTypeDef * hi2c);
void HAL_I2C_AddrCallback (I2C_HandleTypeDef * hi2c, uint8_t TransferDirection, uint16_t AddrMatchCode);
void HAL_I2C_ListenCpltCallback (I2C_HandleTypeDef * hi2c);
void HAL_I2C_MemTxCpltCallback (I2C_HandleTypeDef * hi2c);
void HAL_I2C_MemRxCpltCallback (I2C_HandleTypeDef * hi2c);
void HAL_I2C_ErrorCallback (I2C_HandleTypeDef * hi2c);
void HAL_I2C_AbortCpltCallback (I2C_HandleTypeDef * hi2c);
除了前两个函数已经被HAL库实现，其余的都为弱函数，用户可以根据自己的需求，重新定义声明这些函数。每一种回调函数的执行时间和条件都很清楚的说明在STM32F7的用户手册412页到415页的IO操作中。
这是STM32F7用户手册的下载地址
查阅P412 Polling mode IO MEM operation 和 Interrupt mode IO MEM operation可知，HAL_I2C_Mem_Write/Read_系列的函数，是对内存读写。执行HAL_I2C_Mem_Write/Read_IT函数后，HAL_I2C_MemTx/RxCpltCallback() 函数被调用。其实仔细读一下会发现，DMA实现方法也能使HAL_I2C_MemTx/RxCpltCallback() 函数被调用。

CubeMX配置I2C-IT

啥都不用改，沿用DMA的配置！！！！

代码修改-中断方式实现I2C通信

首先需要用中断的方式需要重新写回调函数。在程序前面添加函数声明：

```
/* USER CODE BEGIN PFP */
void HAL_I2C_MemTxCpltCallback(I2C_HandleTypeDef *hi2c);
void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hi2c);
/* USER CODE END PFP */
```

为了能够证明程序执行了回调函数中的内容，在回调函数中计数，因此全局变量中增添两个变量：

```
uint16_t check_TX;
uint16_t check_RX;
```

后面需要重新定义回调函数：

```
/* USER CODE BEGIN 4 */
void HAL_I2C_MemTxCpltCallback(I2C_HandleTypeDef *hi2c)
{
 check_TX++;
}

void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hi2c)
{
 check_RX++;
}
/* USER CODE END 4 */
```

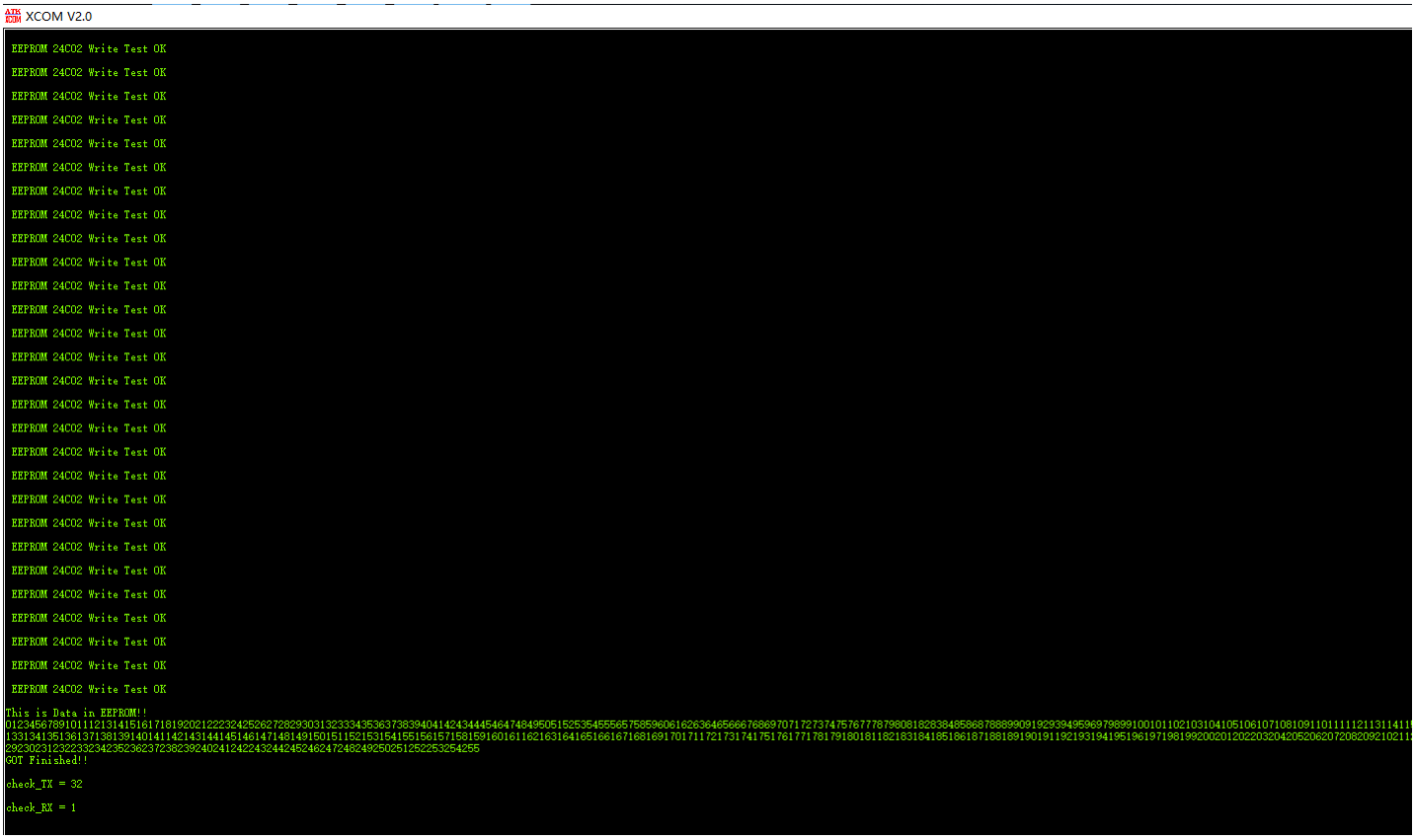
备注：本来打算是要在回调函数中使用printf函数打印出信息的，但是失败了，我认为可能是因为printf函数是一种很费时占资源的操作，在CPU高速调用回调函数时，这种费时操作并不能成功。但是对变量的运算时很简便的，轻松可以完成。所以我采用了计数，而非打印出信息。

最后读写完毕后，在主程序内打印出check_TX和check_RX：
至于读写函数，可以依旧沿用DMA方式，也可以改成中断形式HAL_I2C_Mem_Write/Read_IT，并不影响结果，因为用户手册已经表示，两种方式都能触发HAL_I2C_MemTxCpltCallback和HAL_I2C_MemRxCpltCallback。

```
if ((recv = HAL_I2C_Mem_Read_IT(&hi2c2, ADDR_24LCxx_Read, 0, I2C_MEMADD_SIZE_8BIT, ReadBuffer, BufferSize)) == HAL_OK)
       {
        printf("This is Data in EEPROM!!\r\n");
        for (i = 0; i < 256; i++)
         printf("%d", ReadBuffer[i]);
        printf("\r\nGOT Finished!!\r\n");
        printf("\r\ncheck_TX = %d\r\n",check_TX);
        printf("\r\ncheck_RX = %d\r\n",check_RX);
       }
       else
       {
        printf("Failed to get data\r\n");
        printf("recv =  %d !\r\n", recv);
       }
```

实验结果-中断方式实现I2C通信，打印进入回调函数中的次数

实验结果如下：



可以看出写入了32次，读取了1次。

硬件I2C的极限测试

在网上看到很多人都在说意法半导体为了避免飞利浦的专利问题，STM32系列芯片的硬件I2C通信存在BUG，但这么多年过去了，STM32的HAL库出好几代了，CubeMX更是方便至极，我觉得这么牛皮的公司，应该能解决这些问题吧。所以最后，为了验证I2C的可靠性，做一个极限测试。继续沿用之前的代码，稍作修改，采用DMA的方式进行测试。I2C配置上，直接把速度拉满400KHZ（24C02最大支持到400KHZ）。在程序中，先写入数据。然后疯狂无限读取。看看会不会宕机。

代码修改-每ms读取一次I2C

因为需要1ms读取一次，现在stm32f7xx_it.c文件中声明一个变量Flag_1msChanged;

```
/* USER CODE BEGIN PV */
extern unsigned char Flag_1msChanged;
/* USER CODE END PV */
```

在void SysTick_Handler(void)函数中加一句：

```
void SysTick_Handler(void)
{
  /* USER CODE BEGIN SysTick_IRQn 0 */

  /* USER CODE END SysTick_IRQn 0 */
  HAL_IncTick();
  /* USER CODE BEGIN SysTick_IRQn 1 */
  Flag_1msChanged = 1;//每1ms实现一次标志位改变
  /* USER CODE END SysTick_IRQn 1 */
}
```

回到main.c文件，加入如下变量：

```
/* USER CODE BEGIN PV */
unsigned char Flag_1msChanged = 0;
unsigned char Flag_10msChanged = 0;
unsigned char Flag_100msChanged = 0;
unsigned char Flag_500msChanged = 0;
unsigned char Flag_1000msChanged = 0;
unsigned char Counter_1ms = 0;
unsigned char Counter_10ms = 0;
unsigned char Counter_100ms = 0;
unsigned char Counter_1000ms = 0;
unsigned char Counter_200ms = 0;
/* USER CODE END PV */
```

while（1）函数加入以下逻辑，就能够实现精确地定时操作。

```
 while (1)
  {
   if (Flag_1msChanged == 1)
   {
    Flag_1msChanged = 0;
    Counter_1ms++;
    if (Counter_1ms >= 10)
    {
     Counter_1ms = 0;
     Flag_10msChanged = 1;
     Counter_10ms++;
     if (Counter_10ms >= 10)
     {
      Counter_10ms = 0;
      Flag_100msChanged = 1;
      Counter_100ms++;
      if (Counter_100ms >= (10 * 1))//1s
      {
       Counter_100ms = 0;
      }
```

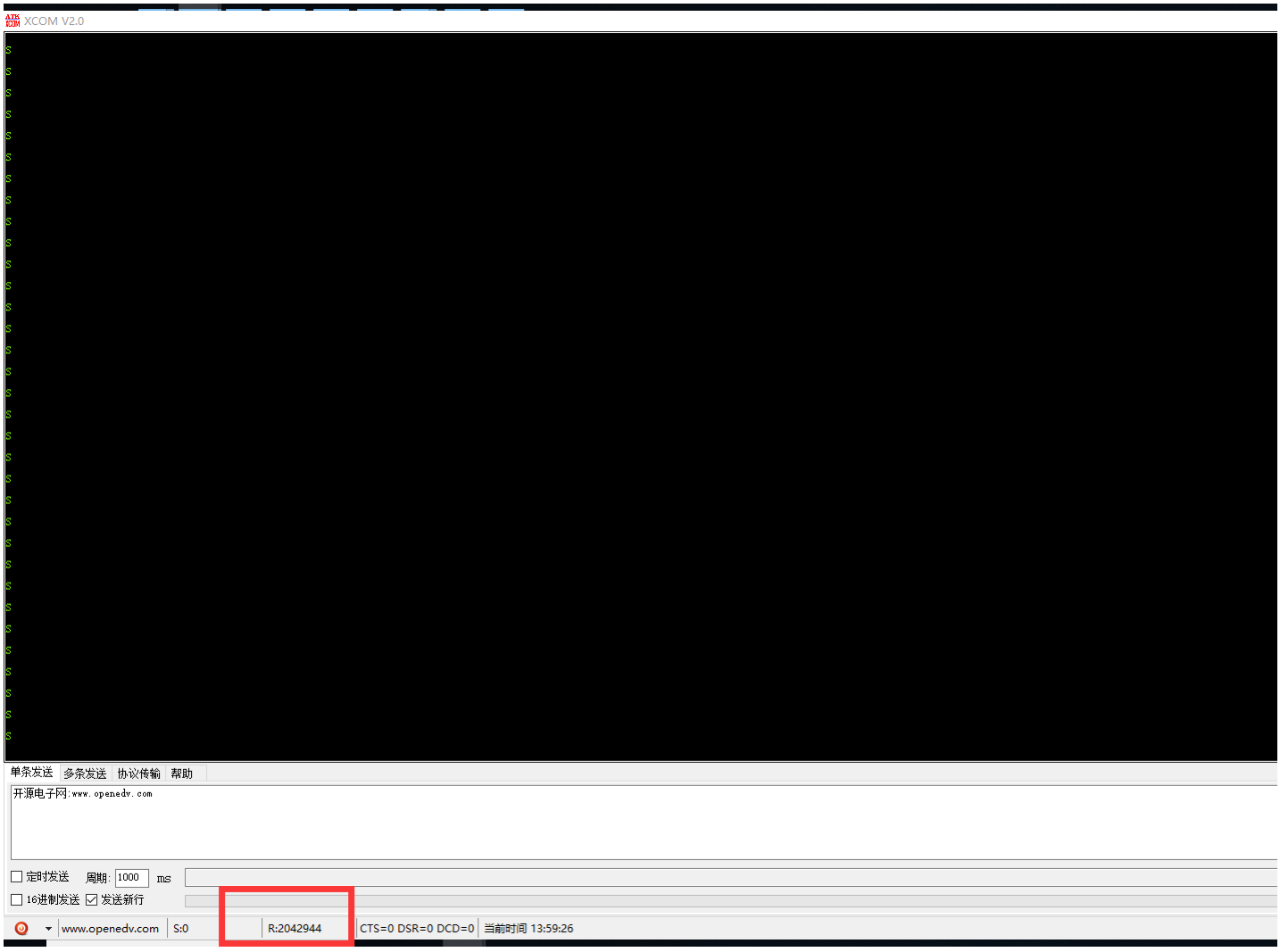
在1ms的区块内增加如下代码：

```
 while (1)
  {
   if (Flag_1msChanged == 1)
```

```
    {
     Flag_1msChanged = 0;
     Counter_1ms++;
//每1ms执行一次的代码
 if ((recv = HAL_I2C_Mem_Read_IT(&hi2c2, ADDR_24LCxx_Read, 0, I2C_MEMADD_SIZE_8BIT, ReadBuffer, BufferSize))
                                                                                    == HAL_OK)
      {
       printf("\r\nS\r\n");
      }
      else
      {
       printf("F\r\n");
       printf("recv =  %d !\r\n", recv);
      }
//成功读取打印S   失败打印F并返回错误代码
     if (Counter_1ms >= 10)
     {
      Counter_1ms = 0;
      Flag_10msChanged = 1;
      Counter_10ms++;
      if (Counter_10ms >= 10)
      {
       Counter_10ms = 0;
       Flag_100msChanged = 1;
       Counter_100ms++;
       if (Counter_100ms >= (10 * 1))//1s
       {
        Counter_100ms = 0;
       }
      }
}
```

实验结果

连续跑了一个多小时没有任何问题，没出现前人发现的宕机之类的问题，也可能是测试代码太过于简单。但我个人认为，稳定性没什么问题的。可以放心使用。下图为测试结果，可以看到已经接收到了两百万余次成功。（确实跑了一个多小时，但是次数没够 这里没有去深究 只是怀疑printf是一种耗时操作，每次循环printf可能都大于1ms）



总结

这些内容也只是STM32的I2C通信的皮毛，在整理资料的过程中仍然发现还有很多未知。但是个人觉得学习东西浅尝辄止就足够了，基本能够应付简单的I2C通信问题。在实际的项目中，若发现当前的知识并不足以解决问题，那么以问题为导向去学习，效率会更高。