**基于SPI的nor flash模块**

**固件详细设计说明**

**Ver 1.0**

**文件号：**

**深 圳 市 智 莱 科 技 股 份 有 限 公 司**

**文 档 配 置 说 明**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 文件名 | 基于SPI的nor flash模块固件详细设计说明V1.0 | | | | | | | |
| 适用 | 文档的发放范围：提供测试部做测试用例、硬件部做电路设计、固件或驱动工程师开发软件驱动、结构部门评估单板安装结构。作为设计输入或参考等。 | | | | | | | |
| 创 建 | 作 者 | 陶路生 | | 时间 | 2019.9.12 | | 版本号 | V1.0 |
| 说 明：  第一版 | | | | | | | |
| 修 改 | 作 者 |  | | 时间 |  | | 版本号 |  |
| 说 明： | | | | | | | |
| 修 改 | 作 者 |  | | 时间 |  | | 版本号 |  |
| 说 明： | | | | | | | |
| 审 核 | 许讯捷 | | 日 期 | | |  | | |
| 批 准 | 许讯捷 | | 日 期 | | |  | | |

**目 录**

[1 概述 4](#_Toc8480)

[2 工程环境简介 4](#_Toc21924)

[3 模块设计 4](#_Toc28083)

[3.1 nor flash 简介 4](#_Toc1122)

[3.2 W25Q64 工作原理 6](#_Toc25777)

[3.3 SPI 操作时序 6](#_Toc21468)

[3.4 Flash 操作编程思想 9](#_Toc7815)

# 概述

在以往的项目开发中，开发工程时经常需要操作nor flash,用来保存一些掉电不丢失得数据，例如设备站点编号灯，由于nor flash 的一些电气属性，促使在对flash 读写操作时，需要一些特定的方式，综合几款常用nor flash ,依托项目，对操作nor flash 做一些总结。 以华邦的W25Q64 为例。

# 工程环境简介

硬件：stm32f070单片机

开发环境：MDK5

MCU系统频率：48MHZ

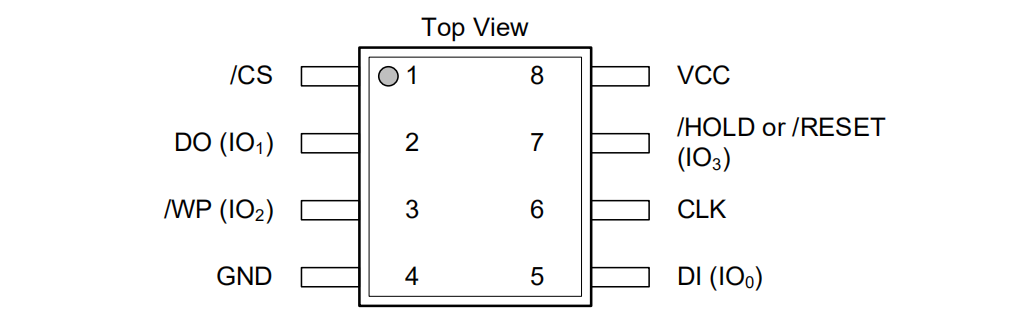
通信方式： spi

基本功能：通过spi总线，对nor flash 进行擦除、读、写 等操作。

# 模块设计

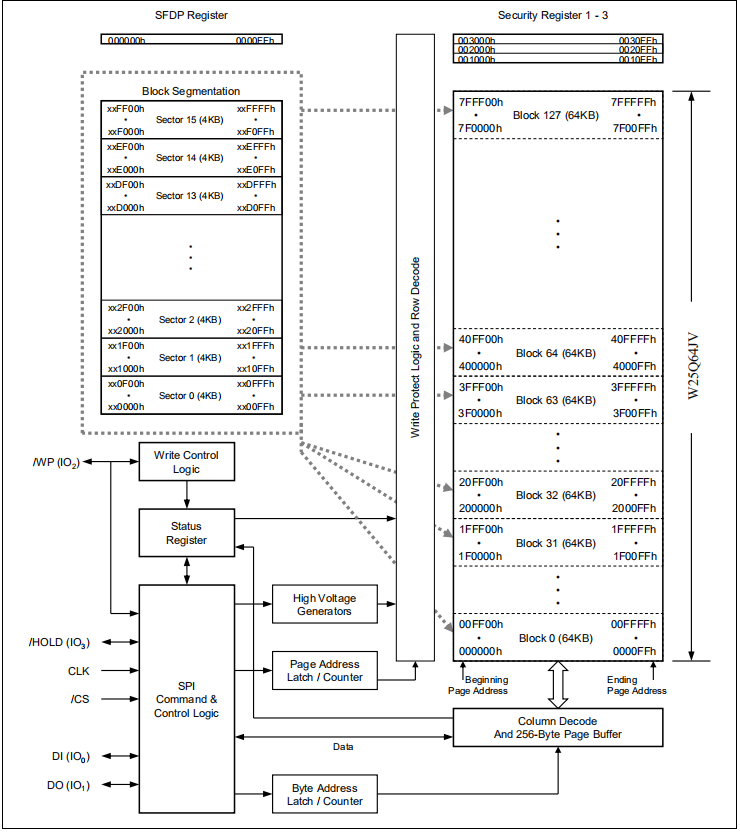
## nor flash 简介

1. 芯片引脚：



|  |  |
| --- | --- |
| 引脚 | 功能 |
| CS | CS为片选管脚，低电平有效。 |
| DO（MISO） | DO为串行数据输出引脚，在CLK（串行时钟）管脚的下降沿输出数据 |
| WP | WP为写保护管脚，有效电平为低电平。高电平可读可写，低电平仅仅可读 |
| DI（MOSI） | DI为串行数据输入引脚，数据、地址和命令从DI引脚输入到芯片内部，在CLK（串行时钟）管脚的上升沿捕获捕获数据。 |
| CLK（SLCK） | CLK为串行时钟引脚。SPI时钟引脚，为输入输出提供时钟脉冲。 |
| HOLD | HOLD为保持管脚，低电平有效。当CS为低电平，并且把HOLD拉低时，数据输出管脚将保持高阻态，并且会忽略数据输入管脚和时钟管脚上的信号。把HOLD管脚拉高，器件恢复正常工作。 |

（2）芯片内部框架



（3）工作方式

W25Q64支持SPI数据传输时序模式0（CPOL = 0、CPHA = 0）和模式3（CPOL = 1、CPHA = 1），模式0和模式3主要区别是当SPI主机硬件接口处于空闲状态时，SCLK的电平状态是高电平或者是低电平。对于模式0来说，SCLK处于低电平；对于模式3来说，SCLK处于高电平。不过，在这两种模式下，芯片都是在SCLK的上升沿采集输入数据，下降沿输出数据。

## W25Q64 指令介绍

1. 操作原理

通过spi 总线接口，用标准的spi 协议发送相关指令给flash , flash 芯片回根据不同的指令完成相关的操作。

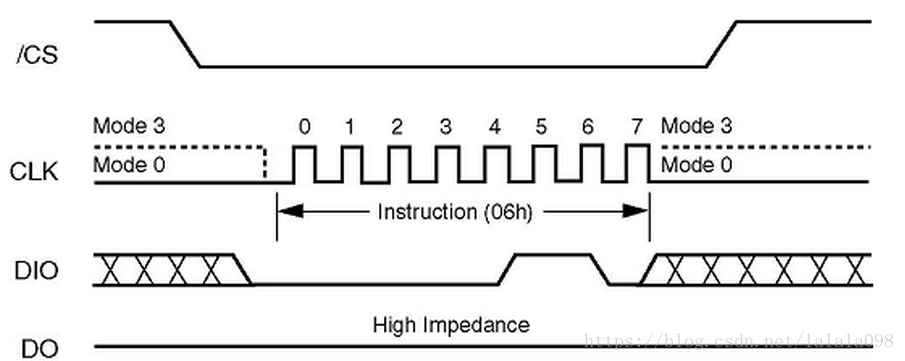
1. W25Q64 常用操作指令

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 指令名称 | 字节1 (地址) | 字节2 | 字节3 | 字节4 | 字节5 | 字节6 | 字节7 |
| 写使能 | 06h |  | | | | | |
| 写禁能 | 04h |  | | | | | |
| 读状态寄存器1 | 05h | (S7-S0) |  |  |  |  |  |
| 读状态寄存器2 | 35h | (S15-S8) |  |  |  |  |  |
| 读数据 | 03h | A23-A16 | A15-A8 | A7-A0 | DATA1 | DATA2 | ... |
| 写状态寄存器 | 01h | S7-S0 | S15-S8 |  |  |  |  |
| 页编程 | 02h | A23-A16 | A15-A8 | A7-A0 | DATA1 | DATA2 |  |
| 块擦除（64K） | D8h | A23-A16 | A15-A8 | A7-A0 |  |  |  |
| 半块擦除（32K） | 52h | A23-A16 | A15-A8 | A7-A0 |  |  |  |
| 扇区擦除（4K） | 20h | A23-A16 | A15-A8 | A7-A0 |  |  |  |
| 芯片擦除 | C7/60h |  |  |  |  |  |  |
| 器件ID | 90h |  |  |  |  |  |  |

## SPI 操作时序

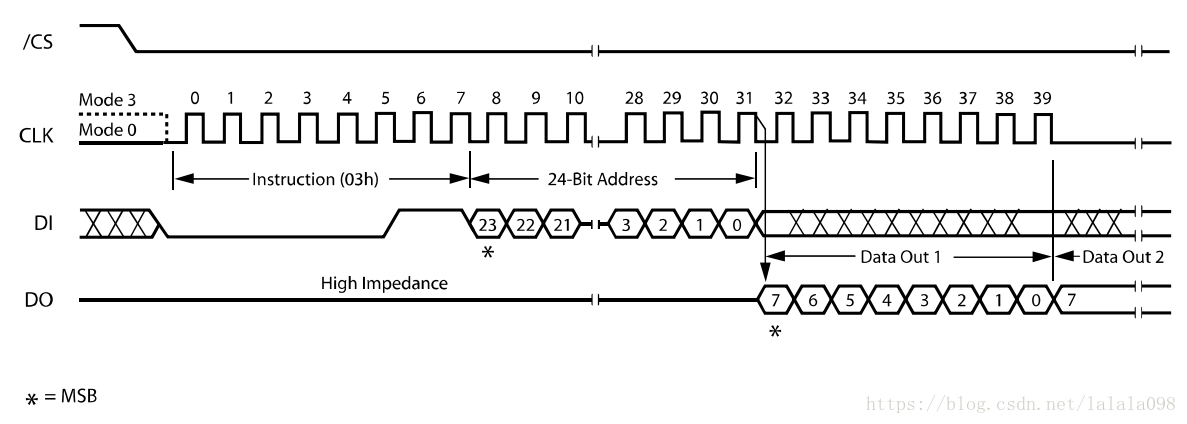
1. 写使能（06h）

“写使能”指令将“状态寄存器”WEL未置位。在每次执行“页编程”、“擦除”、“写状态”等操作时之前，需要只能WEL，且事/CS脚拉低。



（2）读数据（03h）

“读数据”指令允许读出一个或多个字节，,先把/CS引脚拉低，然后把03h 通过DIO 引脚送到新派你，之后发送24位地址，这些数据在clk 的上升沿被芯片采集。芯片接收到到地址后，就会把相应的地址数据在clk引脚的下降沿从DO引脚发送出来，高位在前。当读完这个地址的数据之后，地址自动增加，然后通过DO引脚把下一个地址的数据发送出来，形成一个数据流。也就是说只要时钟在工作，通过一条读指令，就可以把整个芯片的数据读出来，直到把/CS引脚拉高，结束读操作。（但不建议一次读取太多的数据，尽量不要超过4K）。

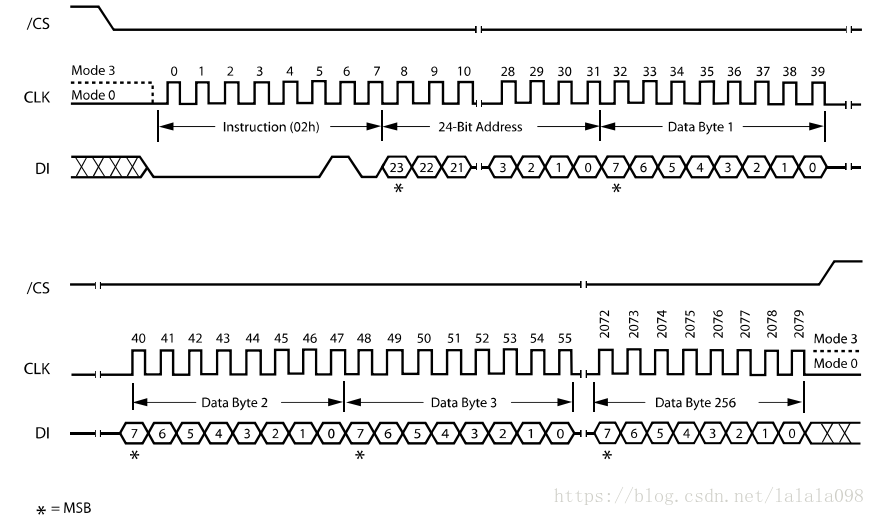


（3）页编程（02h）

执行“页编程”指令之前，需要先执行“写使能”指令，而且要求待写入的区域位都位1，也就是需要先把待写入区先擦除。

在写完一页（256字节）之后，必须把地址给为0 , 不然的话，如果时钟还在继续，地址自动变为页的开始地址。也就是说，在使用页编程指令的时候，操作的地址不能跨页写，否则回写失败。

在写操作完成时候，检测flash 状态，使用“读状态寄存器”查看BUSY 是否被置位，直到该状态为变为0 ，才认为flash 操作完成。

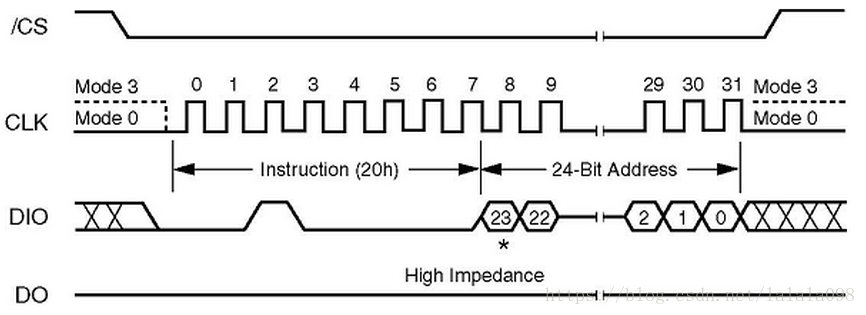


（4）扇区擦除（20h）

“扇区擦除”指令将一个扇区（4k字节）擦除，擦除后，扇区所有数据都为0xff ，在执行扇区擦除指令之前，需要先执行 “写使能”指令，保证WEL位位1。

操作时，先将 /CS引脚拉低，然后发送20h 指令，接着将24位地址发送到芯片，然后拉高/CS 引脚。

操作完成后，需要检测flash 状态寄存器，直到BUSY位为0。



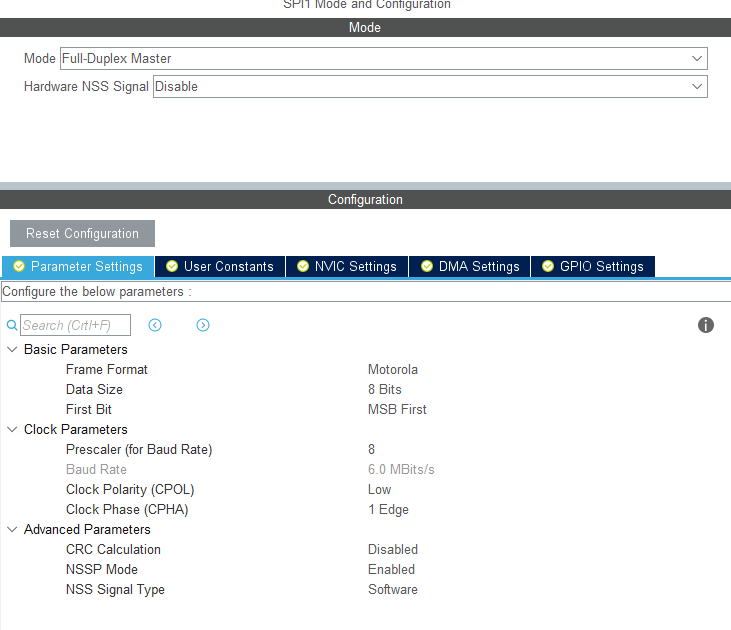
## Flash 操作编程思想

3.4.1 初始化

1. SPI初始化

初始化主要完成对SPI模块与相关应交功能的初始化，不同类型的单片机初始化方式不同，本设计以stm32070 为基础模板，（nrf52840与nxp 相关初始化类似）。

在stm32开发中，我们使用stm32cube 开发工具，配置初始化spi，主要配置参数如图所示



默认配置spi 工作模式为 主从模式, 工作频率为6M（nor flash 最大支持80M的操作频率）

1. 封装spi 读写api

由于不同芯片使用底层库又差异，因此将spi 读写操作二次封装是必要的

SPI 读操作

|  |
| --- |
| static uint8\_t spi\_read\_data(uint8\_t \*tx\_buf, uint16\_t tx\_len, uint8\_t \*rx\_buf, uint16\_t rx\_len)  {  HAL\_StatusTypeDef ret\_code;  FLASH\_CSEN();  HAL\_SPI\_Transmit(&hspi1, tx\_buf, tx\_len, W25Qx\_TIMEOUT\_VALUE);  ret\_code = HAL\_SPI\_Receive(&hspi1, rx\_buf, rx\_len, W25Qx\_TIMEOUT\_VALUE);  FLASH\_CSDIS();  if(ret\_code != HAL\_OK)  {  return SPI\_ERROR;  }  else  {  return SPI\_OK;  }  } |

SPI 写操作

|  |
| --- |
| static uint8\_t spi\_write\_data( uint8\_t \*write\_buf, uint16\_t length)  {  HAL\_StatusTypeDef ret\_code;  FLASH\_CSEN();  ret\_code = HAL\_SPI\_Transmit(&hspi1, write\_buf, length, W25Qx\_TIMEOUT\_VALUE);  FLASH\_CSDIS();  if(ret\_code != HAL\_OK )  {  return SPI\_ERROR;  }  else  {  return SPI\_OK;  }  } |

3.4.2 flash 读/ 写 / 擦除 等操作

（1）flash 擦除操作

Flash 操作包括，扇区擦除（4K）、半块擦除（32K）、块擦除（64K）, 由芯片手册可知，flash 擦除操作相对比较耗时，实际测试使用4M SPI 通信频率，擦除一个扇区大概需要

50ms，所以在擦除操作的时候需要关注下时间。

需要注意 ，WriteAddr 可以是flash 空间内任意地址，实际擦除的位置未写入地址所在的块区，例如 当 WriteAddr = 100，实际擦除数据以为未第0块（0-4096地址）。

|  |
| --- |
| uint8\_t flash\_sector\_erase\_4KByte(uint32\_t WriteAddr)  {  uint8\_t ret ;  uint8\_t len = 4;  flash\_write\_en();  spi\_tx\_buf[0] = SECTOR\_ERACE\_4KB;  spi\_tx\_buf[1] = (uint8\_t)((WriteAddr & 0x00ff0000) >> 16);  spi\_tx\_buf[2] = (uint8\_t)((WriteAddr & 0x0000ff00) >> 8);  spi\_tx\_buf[3] = (uint8\_t)WriteAddr;  ret = spi\_write\_data(spi\_tx\_buf, len);  ret = flash\_wait\_busy();  return ret ;  } |

1. Flash 读操作

Flash 读操作没由过多注意，一般情况下，可以从任意地址，读取任意长度的数据，但由flash 发送读取指令后，地址位回自动累加，为防止出现错误，尽量不要一次读出过多数据，如需读取大量数据，需要尽量分扇区读取。

|  |
| --- |
| uint8\_t flash\_read\_data(uint32\_t ReadAddr, uint8\_t \*pBuffer, uint16\_t length)  {  uint8\_t ret ;  spi\_tx\_buf[0] = READ\_DATA;  spi\_tx\_buf[1] = (uint8\_t)((ReadAddr & 0x00ff0000) >> 16);  spi\_tx\_buf[2] = (uint8\_t)((ReadAddr & 0x0000ff00) >> 8);  spi\_tx\_buf[3] = (uint8\_t)ReadAddr;  ret = spi\_read\_data( spi\_tx\_buf, 4, pBuffer, length);  return ret ;  } |

1. flash 写操作

Flash 写操作写操作需要注意，我们使用的flash 指令位页编程页，所以单次写数据，不能操作256（一页的大小）。

也就是说，我们操作的时候 WriteAddr % SPI\_FLASH\_PAGE\_SIZE + WriteBytesNum 不允许大于一页的大小。

例如 当 WriteAddr = 100；如果我们需要写入的数据长度为158 （大于 SPI\_FLASH\_PAGE\_SIZE-WriteAddr ），也就是操作的时候跨越了页，因此跨页的数据会写失败。

|  |
| --- |
| uint8\_t flash\_write\_page(uint32\_t WriteAddr, uint8\_t \*pBuffer, uint16\_t WriteBytesNum )  {  uint8\_t ret ;  uint16\_t len;  uint8\_t flash\_status;  if(WriteAddr % SPI\_FLASH\_PAGE\_SIZE + WriteBytesNum > 256)  {  return SPI\_ERROR;  }  flash\_write\_en();  spi\_tx\_buf[0] = PAGE\_PROGRAM;  spi\_tx\_buf[1] = (uint8\_t)((WriteAddr & 0x00ff0000) >> 16);  spi\_tx\_buf[2] = (uint8\_t)((WriteAddr & 0x0000ff00) >> 8);  spi\_tx\_buf[3] = (uint8\_t)WriteAddr;  memcpy(&spi\_tx\_buf[4], pBuffer, WriteBytesNum);  len = WriteBytesNum + 4;  ret = spi\_write\_data( spi\_tx\_buf, len);  ret = flash\_wait\_busy();  return ret;  } |

1. 任意写操作

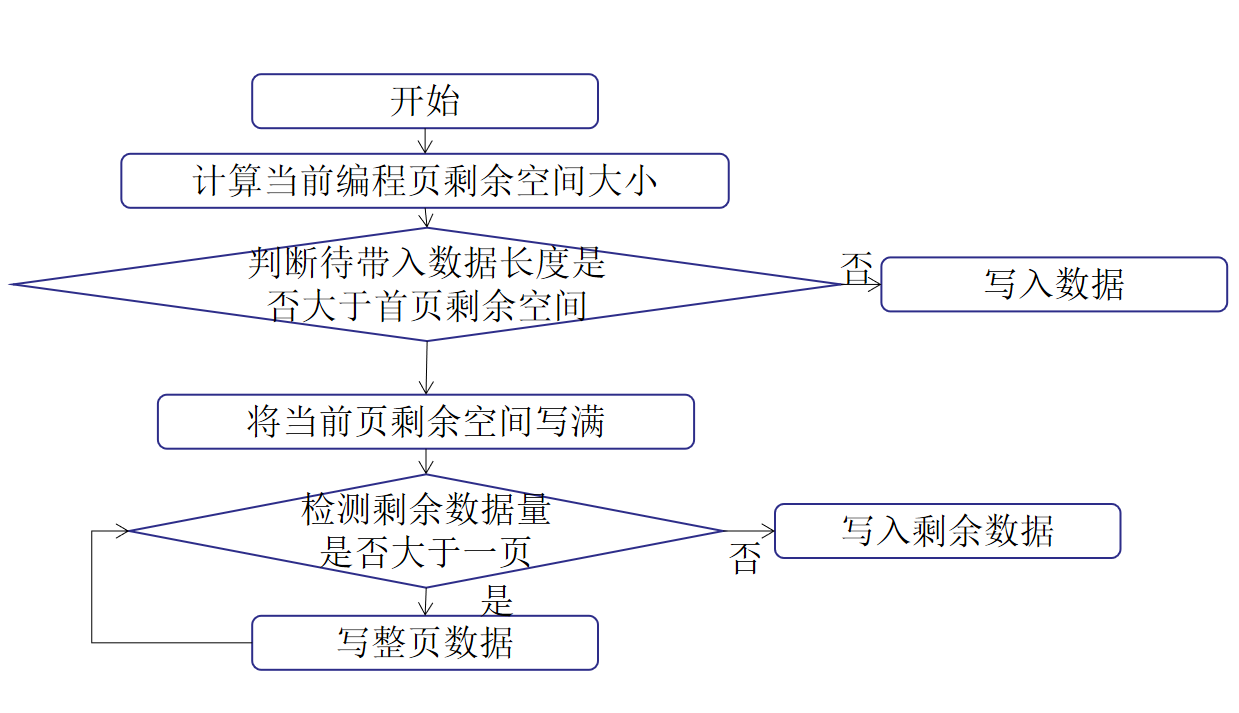
为简化操作，与防止写数据时需要跨页操作，因此，我们对 单页写数据的api 进行再次封装，封住原理为

A: 检测写入首地址所在页剩余空间，先将低一些数据写完

B: 将剩余数据分为页大小写数据。

|  |
| --- |
| uint8\_t spi\_flash\_write\_data(uint32\_t WriteAddr, uint8\_t\* pBuffer, uint16\_t NumByteToWrite)  {  uint8\_t ret ;  int32\_t residu\_len = NumByteToWrite; //未写入数据 长度  uint16\_t pageremain;  pageremain = SPI\_FLASH\_PAGE\_SIZE - WriteAddr % SPI\_FLASH\_PAGE\_SIZE; //判断当前页剩余空间  if(residu\_len <= pageremain) //如果当前页剩余空间 大于待写入数据长度  {  pageremain = residu\_len; //256ֽ  ret = flash\_write\_page( WriteAddr, (void \*)pBuffer, pageremain);  return ret;  }  else  {  while(1)  {  ret = flash\_write\_page( WriteAddr, (void \*)pBuffer, pageremain); //写入数据  if(ret != SPI\_OK)  {  return ret ; //如果写失败，直接return。  }  residu\_len = residu\_len - pageremain; //待写入数据剩余长度  if(residu\_len > 0)  {  pBuffer += pageremain; //数据指针偏移  WriteAddr += pageremain; //写入数据flash地址偏移  if(residu\_len > SPI\_FLASH\_PAGE\_SIZE) //如果剩余数据长度 大于一页大小  {  pageremain = SPI\_FLASH\_PAGE\_SIZE; //写一整页数据  }  else  {  pageremain = residu\_len; //剩余长度  }  }  else  {  break; //写入完成  }  }  }  return ret;  } |

流程图表示：



---完---