

STM32 软件编程基本操作

作者:向仔州

STM32 寄存器位绑定.....	2
Stm32 系统时钟初始化.....	5
STM32 寄存器操作 GPIO 驱动 led.....	8
STM32 固件库函数配置时钟.....	9
STM32 IO 口作为输入寄存器方式.....	11
STM32 固件库工程需要的基本头文件和 C 文件.....	12
STM32 IO 口作为输入库函数方式.....	13
STM32 中断系统.....	13
STM32 中断固件库操作.....	22
STM32 串口实现和 printf 实现.....	25
STM32 定时器.....	27
FSMC 驱动 LCD.....	35
LCD 字符解码方式.....	44
STM32 内部 CAN 总线使用.....	57
STM32 串口库函数操作.....	67
STM32+MCP2515 CAN 控制器测试.....	69
STM32 SPI 接口 读写 norflash 案例.....	82
SPI 操作 norflash(W25Q64)进行数据块读写.....	91
文件系统移植到 norflash.....	98
SDIO 接口驱动 SD 卡.....	116
SD 卡 FATFS 文件系统移植(这里涉及到两个存储设备).....	129
FSMC 驱动 SRAM 实现，方便移植 malloc 内存分配函数定义的变量存放 to SRAM.....	136
STM32 内存管理 malloc 底层实现.....	142
外部 SRAM 做动态内存.....	146
WAV 文件创建代码设计.....	148
I2S 总线获取音频数据.....	150
DMA 驱动实验.....	154

STM32 寄存器位绑定

以 stm32f103VC 为例：

该芯片可以未绑定的地址有 SRAM 区 **0x2000 0000 ~ 0x200f ffff 1M**

还有片上外设，也就是什么 SPI, I2C, DMA 类的从 **0x4000 0000 ~ 0x400f ffff 1M**

以 SRAM 为例来说明位绑定

2.3.1 嵌入式SRAM

STM32F10xxx内置64K字节的静态SRAM。它可以以字节、半字(16位)或全字(32位)访问。
SRAM的起始地址是0x2000 0000。

SRAM 其实地址是 0x2000 0000，那么我要操作 0x20000300 怎么办呢

bit_word_addr = bit_band_base + (byte_offset × 32) + (bit_number × 4)

bit_word_addr 绑定后产生的新地址。

bit_band_base 起始地址。

byte_offset 偏移地址

bit_number 这个地址里面的第几个引脚(0-31)

下面的例子说明如何映射别名区中SRAM地址为0x20000300的字节中的位2:

$0x22006008 = 0x22000000 + (0x300 \times 32) + (2 \times 4)$.

对0x22006008地址的写操作与对SRAM中地址0x20000300字节的位2执行读-改-写操作有着相同的效果。

如果是片上外设的话也一样，只是从 $0x42000000 + (A - 0x40000000) * 32 + (bit_number \times 4)$

比如我们要来操作一个 GPIO, 那么就可以用位操作，看下面

绑定 GPIOA 里面的 ODR 和 IDR 里面的第 4 位，然后操作该位置 1 或置 0

8.2.3 端口输入数据寄存器(GPIOx_IDR) (x=A..E)

地址偏移: 0x08 复位值: 0x0000 XXXX															
保留															
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0															
IDR15 IDR14 IDR13 IDR12 IDR11 IDR10 IDR9 IDR8 IDR7 IDR6 IDR5 IDR4 IDR3 IDR2 IDR1 IDR0															
r r r r r r r r r r r r r r r r															
位31:16 保留，始终读为0。															
位15:0 IDRy[15:0]: 端口输入数据(y = 0...15) (Port input data) 这些位只读并只能以字(16位)的形式读出。读出的值为对应I/O口的状态。															

IDR 第 4 位

8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E)

地址偏移: 0Ch 复位值: 0x0000 0000															
保留															
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0															
ODR15 ODR14 ODR13 ODR12 ODR11 ODR10 ODR9 ODR8 ODR7 ODR6 ODR5 ODR4 ODR3 ODR2 ODR1 ODR0															
r w r w r w r w r w r w r w r w r w r w r w															
位31:16 保留，始终读为0。															
位15:0 ODRy[15:0]: 端口输出数据(y = 0...15) (Port output data) 这些位可读可写并只能以字(16位)的形式操作。 注：对GPIOx_BSRR(x = A..E)，可以分别地对各个ODR位进行独立的设置/清除。															

ODR 第 4 位

先找到 GPIOA 的地址

```
1410 #define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
      #define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
```

GPIOA 基地址

```
1317 | #define GPIOA_BASE          (APB2PERIPH_BASE + 0x0800)
```

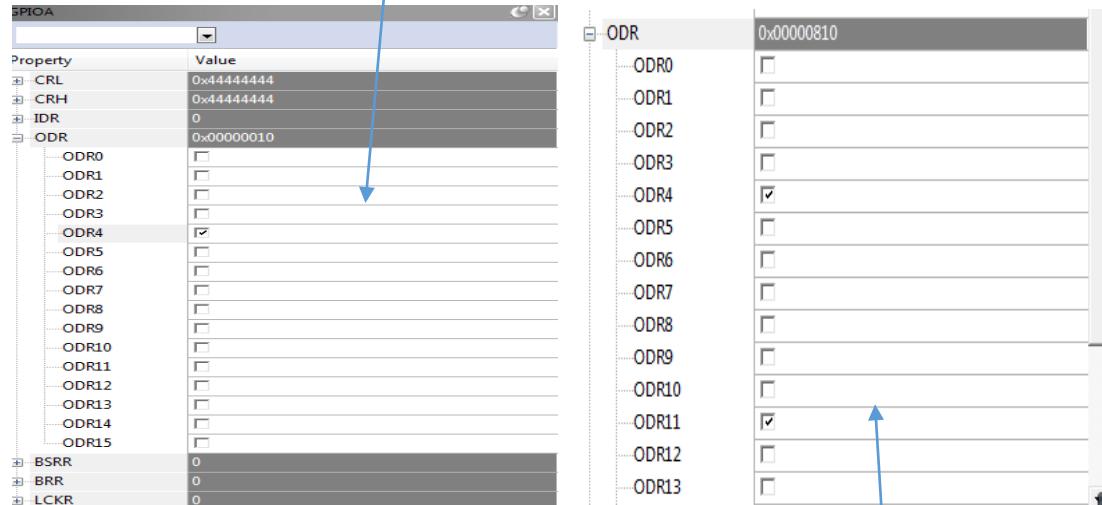
然后再找到偏移地址 IDR 是 0x08h, ODR 是 0x0Ch。

计算 ODR 把 GPIOA_BASE 起始地址带入公式 $0x42000000 + (GPIOA_BASE + 0x0Ch - 0x40000000) * 32 + (\text{bit_number} \times 4)$

然后确定用 IDR 的第 4 位，计算 IDR 把 GPIOA_BASE 起始地址带入公 $0x42000000 + (GPIOA_BASE + 0x0Ch - 0x40000000) * 32 + (4 \times 4)$

这样就可以操作 GPIOA – pin4 引脚输出电平了

```
1 #include<stdio.h>
2 #include "stm32f10x.h"
3
4 int main()
5 {
6
7     u32 *PAO4 = (u32 *) (0x42000000 + ((GPIOA_BASE+0x0C)-0x40000000)*32+4*4);
8
9     *PAO4=1;
10    *PAO4=0;
11    *PAO4=1;
12    return 0;
13 }
```



如果要控制第 ODR 第 11 位

```
int main()
{
    u32 *PAO4 = (u32 *) (0x42000000 + ((GPIOA_BASE+0x0C)-0x40000000)*32+4*4); //这是控制GPIO低8位CRL的4位
    u32 *PAO11 = (u32 *) (0x42000000 + ((GPIOA_BASE+0x0C)-0x40000000)*32+11*4); //这是控制GPIO高8位CRH的11位

    *PAO4=1;
    *PAO4=0;
    *PAO4=1;

    *PAO11=1;
    *PAO11=0;
    *PAO11=1;
```

但是上面这种位绑定方式很麻烦，所以需要弄一个简单的方法

```

1 #include<stdio.h>
2 #include "stm32f10x.h"
3
4 #define GPIO_ODR_A (GPIOA_BASE+0x0C)//定义GPIOA的基地址
5 #define GPIO_IDR_A (GPIOA_BASE+0x08)//定义GPIOA的基地址
6
7 #define GPIO_ODR_B (GPIOB_BASE+0x0C)//定义GPIOB的基地址
8 #define GPIO_IDR_B (GPIOB_BASE+0x08)//定义GPIOB的基地址
9
10#define BitBand(addr,bitnum) *((volatile unsigned long *)((addr & 0xF0000000)+0x2000000+((addr & 0xFFFF)<<5)+(bitnum<<2)))
11
12
13#define PAout(n) BitBand(GPIO_ODR_A,n)
14#define PAin(n) BitBand(GPIO_IDR_A,n)
15
16#define PBout(n) BitBand(GPIO_ODR_B,n)
17#define PBin(n) BitBand(GPIO_IDR_B,n)
18
19 int main()
20 {
21
22 PAout(14)=1;
23 PAout(14)=0;
24 PAout(14)=1;
25
26 PBout(14)=1;
27 PBout(14)=0;
28 PBout(14)=1;
29

```

ODR		0x00004000
ODR0	<input type="checkbox"/>	
ODR1	<input type="checkbox"/>	
ODR2	<input type="checkbox"/>	
ODR3	<input type="checkbox"/>	
ODR4	<input type="checkbox"/>	
ODR5	<input type="checkbox"/>	
ODR6	<input type="checkbox"/>	
ODR7	<input type="checkbox"/>	
ODR8	<input type="checkbox"/>	
ODR9	<input type="checkbox"/>	
ODR10	<input type="checkbox"/>	
ODR11	<input type="checkbox"/>	
ODR12	<input type="checkbox"/>	
ODR13	<input type="checkbox"/>	
ODR14	<input checked="" type="checkbox"/>	
ODR15	<input type="checkbox"/>	
BSRR	0	
BRR	0	
LCKR	0	

ODR
[Bits 31..0] RW (@ 0x4001080C) Port output data register (GPIOn_ODR)

GPIOB GPIOA

ODR		0x00004000
ODR0	<input type="checkbox"/>	
ODR1	<input type="checkbox"/>	
ODR2	<input type="checkbox"/>	
ODR3	<input type="checkbox"/>	
ODR4	<input type="checkbox"/>	
ODR5	<input type="checkbox"/>	
ODR6	<input type="checkbox"/>	
ODR7	<input type="checkbox"/>	
ODR8	<input type="checkbox"/>	
ODR9	<input type="checkbox"/>	
ODR10	<input type="checkbox"/>	
ODR11	<input type="checkbox"/>	
ODR12	<input type="checkbox"/>	
ODR13	<input type="checkbox"/>	
ODR14	<input checked="" type="checkbox"/>	
ODR15	<input type="checkbox"/>	
BSRR	0	
BRR	0	
LCKR	0	

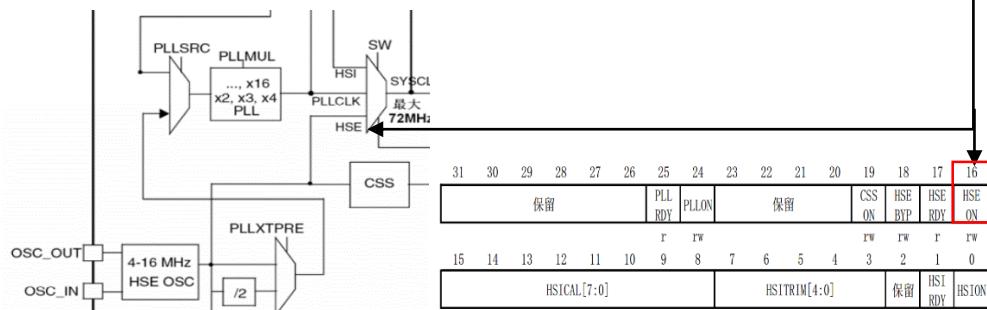
ODR
[Bits 31..0] RW (@ 0x40010C0C) Port output data register (GPIOn_ODR)

GPIOB GPIOA

其实这就是正点原子使用的方法，但是 GPIO 管脚配置还得用传统的方法。只要 GPIO 电平输入输出可以用位绑定。

Stm32 系统时钟初始化

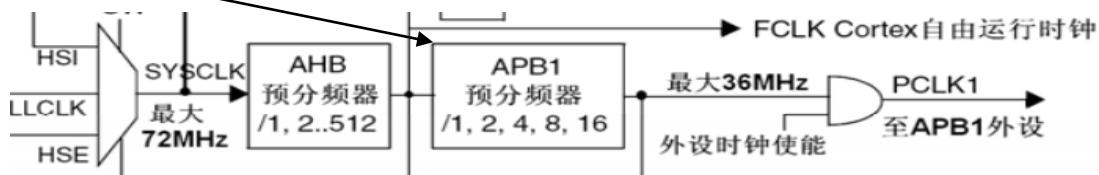
```
void stm32_clock_init(u8 PLL)
{
    Unsigned char temp=0;
    MYRCC_Deinit(); //复位并配置中断向量表
    RCC->CR|=0x00010000 //把 RC 时钟寄存器 16 置 1 来开启外部振荡器
```



while(!(RCC->CR>>17)); //查询 CR 寄存器第 17 位，确定上面的操作让外部振荡器就绪没得外部振荡器就绪的话就会给寄存器置 1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
				PLL RDY	PLLON			保留		CSS ON	HSE BYP	HSE RDY	HSE ON		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]				保留	HSI RDY	HSION	

RCC->CFGR=0x00000400; //设置时钟配置寄存器保证 APB1 时钟不超过 36M



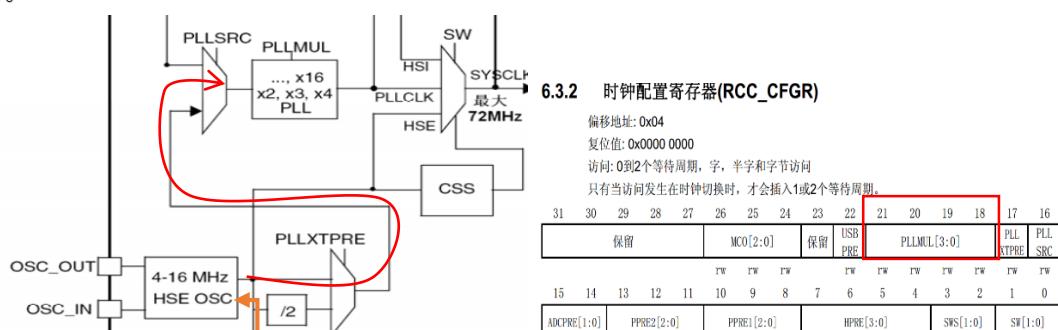
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
				MCO[2:0]	保留	USB PRE	PLLMUL[3:0]	PLL XTPRE	PLL SRC						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADCPRE[1:0]	PPRE2[2:0]	PPRE1[2:0]		HPRE[3:0]	SWS[1:0]	SW[1:0]									
rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	r	rw	rw	

位10:8

PPRE1[2:0]: 低速APB预分频(APB1) (APB low-speed prescaler (APB1))
由软件置1或清0来控制低速APB1时钟(PCLK1)的预分频系数。
警告：软件必须保证APB1时钟频率不超过36MHz。

0xx: HCLK不分频
100: HCLK 2分频
101: HCLK 4分频
110: HCLK 8分频
111: HCLK 16分频

PLL=-2; //抵消两位是因为 u8 PLL 传进来的如果是 72，那么 PLL 得到的是 9。

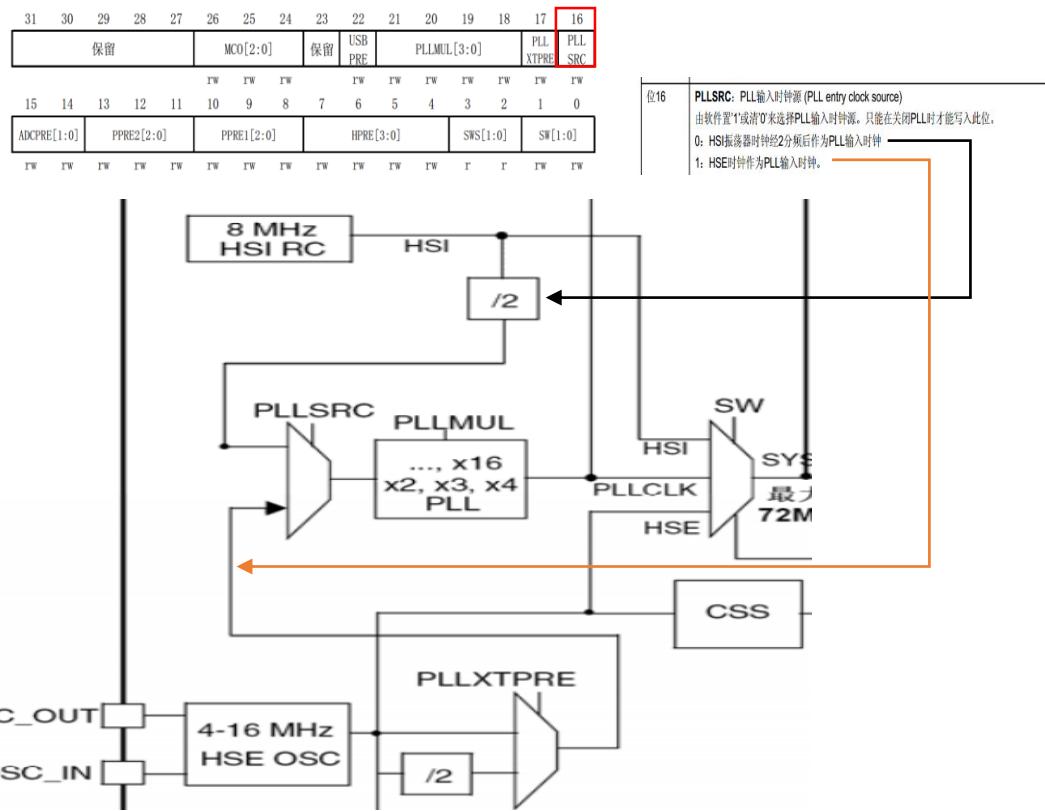


如果是 9 的话，如果是 9 的话就是 11 倍频，我们常用的晶振是 8M 那么 $8 \times 11 = 88M$ ，这样就不对了，所以要抵消两位就是 9 倍频， $8 \times 9 = 72M$ 刚好合适，如果你换成 4M 晶振当然可以不用抵消，但是只要 44M。

RCC->CFGR |= PLL<<18; //设置计算好的 PLL 值

位21:18	PLLMUL: PLL倍频系数 (PLL multiplication factor) 由软件设置来确定PLL倍频系数。只有在PLL关闭的情况下才可被写入。 注意: PLL的输出频率不能超过72MHz
	0000: PLL 2倍频输出
	0001: PLL 3倍频输出
	0010: PLL 4倍频输出
	0011: PLL 5倍频输出
	0100: PLL 6倍频输出
	0101: PLL 7倍频输出
	0110: PLL 8倍频输出
	0111: PLL 9倍频输出
	1000: PLL 10倍频输出
	1001: PLL 11倍频输出
	1010: PLL 12倍频输出
	1011: PLL 13倍频输出
	1100: PLL 14倍频输出
	1101: PLL 15倍频输出
	1110: PLL 16倍频输出
	1111: PLL 16倍频输出

RCC->CFGR |= 1<<16; //外部时钟作为 PLL 输入



FLASH->ACR |= 0x32; //配置完了让芯片休息 2 个周期。

RCC->CR |= 0x01000000; //设置 CR 寄存器的第 24 位, 让 PLL 启动起来, 上门两页都是设置, 现在才开始正式启动。

while(!!(RCC->CR>>25)); //使能锁相环后判断 PLL 是不是正常工作, 就要用 CR 第 25 位来判断

6.3.1 时钟控制寄存器(RCC_CR)

偏移地址: 0x00	保留	PLLE	PLLON	保留	CSS ON	HSE BYP	HSE RDY	HSE ON
复位值: 0x0000 XX83, X代表未定义		r	rw		rw	r	rw	
访问: 无等待状态, 字, 半字 和 字节访问	保留	保留	保留	保留	保留	保留	保留	保留
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	HSICAL[7:0]	HSITRIM[4:0]	保留	HS1RDY	HS1ON			
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0								

位31:26 保留, 始终读为0。

位25 **PLLRDY:** PLL时钟就绪标志 (PLL clock ready flag)

PLL锁定后由硬件置“1”。

0: PLL未锁定;

1: PLL锁定。

位24 **PLLON:** PLL使能 (PLL enable)

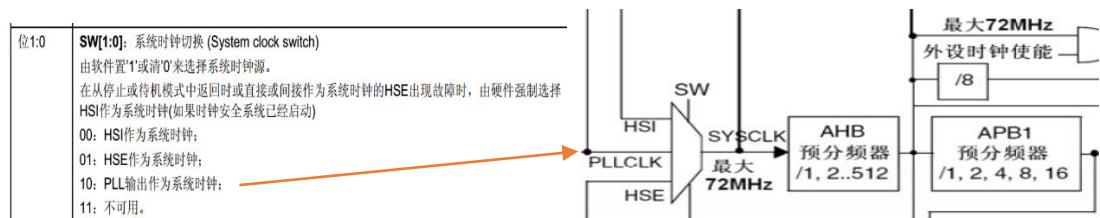
由软件置“1”或清零。

当进入待机和停止模式时, 该位由硬件清零。当PLL时钟被用作或选择将要作为系统时钟时, 该位不能被清零。

0: PLL关闭;

1: PLL使能。

RCC->CFGR |=0x00000002; //设置 CFGR 寄存器 0-1 位把 PLL 作为输入



while(temp!=0x02) //等待 PLL 设置成功

{

位3:2	SWS[1:0]: 系统时钟切换状态 (System clock switch status) 由硬件置'1'或清'0'来指示哪一个时钟源被作为系统时钟。 00: HSI作为系统时钟； 01: HSE作为系统时钟； 10: PLL输出作为系统时钟； 11: 不可用。	多次一举
------	---	------

```

tmp=RCC->CFGR>>2;
temp=&0x30;
}
}

```

以上时钟设置完成了

代码清单：

```

void  stm32_clock_init(u8 PLL)
{
  Unsigned char temp=0;
  MYRCC_Deinit();          //复位并配置中断向量表
  RCC->CR|=0x00010000     //把 RC 时钟寄存器 16 置 1 来开启外部振荡器
  while(!(RCC->CR>>17)); //查询 CR 寄存器第 17 位，确定上面的操作让外部振荡器就绪没得
  外部振荡器就绪的话就会给寄存器置 1
  RCC->CFGR=0x00000400;    //设置时钟配置寄存器保证 APB1 时钟不超过 36M
  PLL=-2;
  RCC->CFGR |=PLL<<18;//设置计算好的 PLL 值
  RCC->CFGR |=1<<16;     //外部时钟作为 PLL 输入
  FLASH->ACR |=0x32;      //配置完了让芯片休息 2 个周期。
  RCC->CR |=0x01000000;    //设置 CR 寄存器的第 24 位，让 PLL 启动起来，上门两页都是
  设置，现在才开始正式启动。
  while(!(RCC->CR>>25)); //使能锁相环后判断 PLL 是不是正常工作，就要用 CR 第 25 位来
  判断
  RCC->CFGR |=0x00000002; //设置 CFGR 寄存器 0-1 位把 PLL 作为输入
  while(temp!=0x02){ //等待 PLL 设置成功
    tmp=RCC->CFGR>>2;//多次一举
    temp=&0x30;
  }
}

```

STM32 寄存器操作 GPIO 驱动 led

最前面我们用位定义来控制 GPIO 的输出和输入，但是 GPIO 在 STM32 有很多种功能

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	3[1:0]	MODE3[1:0]	CNF2[1:0]	MODE2[1:0]	CNF1[1:0]	MODE1[1:0]	CNF0[1:0]	MODE0[1:0]								
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
位31:30		CNFy[1:0]: 端口x配置位(y = 0...7) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式														
位29:28		MODEy[1:0]: 端口x的模式位(y = 0...7) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式, 最大速度10MHz 10: 输出模式, 最大速度2MHz 11: 输出模式, 最大速度50MHz														

这些功能用位绑定来设置是不现实的，所以这里还是要用移位运算。

```

1 #include<stm32f10x.h>
2
3 int main()
4 {
5     GPIOB->CRL=0xffffffff;//在设置GPIOB引脚时,为了不影响其他的GPIO的配置功能先用&运算讲GPIOB清0
6     GPIOB->CRL|=0x00000003;//GPIOB_CRL寄存器CNF0位00, MODE0为11, 所以是通用推挽输出模式, 输出速率设置为50M
7     GPIOB->ODR=1<<0;//也就是给GPIOB第0个引脚输出1
8
9 //当然也可以使用我们未绑定来设置输出电平
10 PBout(0)=1;//这个和 GPIOB->ODR=1<<0;是一样的, 只是位绑定便于好看
11
12 return 0;
13 }

```

8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

偏移地址: 0x00

复位值: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]	MODE7[1:0]	CNF6[1:0]	MODE6[1:0]	CNF5[1:0]	MODE5[1:0]	CNF4[1:0]	MODE4[1:0]	CNF3[1:0]	MODE3[1:0]	CNF2[1:0]	MODE2[1:0]	CNF1[1:0]	MODE1[1:0]	CNF0[1:0]	MODE0[1:0]
rw	rw														

8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E)

地址偏移: 0Ch

复位值: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

这样才能真正点亮一个 led，有引脚配置，还有引脚输出才是对的。

```

1 #include<stm32f10x.h>
2
3 int main()
4 {
5     STM32_clock_init();
6     GPIOB->CRL=0xffffffff;//在设置GPIOB引脚时,为了不影响其他的GPIO的配置功能先用&运算讲GPIOB清0
7     GPIOB->CRL|=0x00000003;//GPIOB_CRL寄存器CNF0位00, MODE0为11, 所以是通用推挽输出模式, 输出速率设置为50M
8     GPIOB->ODR=1<<0;//也就是给GPIOB第0个引脚输出1
9
10 PBout(0)=1;//这个和 GPIOB->ODR=1<<0;是一样的, 只是位绑定便于好看
11
12 return 0;
13 }

```

这样才正确，正点原子版本也有时钟初始化只是名字可能不一样，但是这个过程千万不能忘

有个地方我们忘了就要先初始化时钟

初始化时钟之后该 IO 的 led 没有像软件仿真那样点亮，是为什么呢？

```

1 #include<stm32f10x.h>
2
3 int main()
4 {
5     stm32_clock_init();
6     RCC->APB2RSTR=0;
7
8     GPIOB->CR1=0xffffffff; //在设置GPIOB引脚时，为了不影响其他的GPIO的配置功能先用4连起来GPIOB0引脚
9     GPIOB->CR1|=0x00000003; //GPIOB CR1寄存器EN0位0, MODE0为1, 所以是通用推挽输出模式，输出速率设置为50M
10    GPIOB->DR=1<<0; //也就是给GPIOB第0个引脚输出1
11
12    //当然也可以使用我们来分别来设置输出电平
13    POut(0)=1; //这个和 GPIOB->DR=1<<0;是一样的，只是位绑定便于好看
14
15    return 0;
16 }

```

打开 APB2 GPIOB

- 6.3.4 APB2外设复位寄存器 (RCC_APB2RSTR)
- 6.3.5 APB1外设复位寄存器 (RCC_APB1RSTR)
- 6.3.6 AHB外设时钟使能寄存器 (RCC_AHBENR)
- 6.3.7 APB2外设时钟使能寄存器 (RCC_APB2ENR)
- 6.3.8 APB1外设时钟使能寄存器 (RCC_APB1ENR)

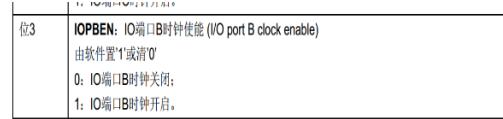
这样才能正常点亮 LED

6.3.7 APB2 外设时钟使能寄存器 (RCC_APB2ENR)

偏移地址: 0x18
复位值: 0x0000 0000
访问: 字, 半字和字节访问
通常无访问等待周期。但在 APB2 总线上的外设被访问时, 将插入等待状态直到 APB2 的外设访问结束。

注: 当外设时钟没有启用时, 程序不能读出外设寄存器的数值, 返回的数值始终是 0x0。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC3 EN	USART1 EN	TIM8 EN	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	IOPG EN	IOPF EN	IOPB EN	IOPC EN	IOPD EN	IOPA EN	IOPB EN	保留	AFIO EN



因为 GPIOB 是挂接在 APB2 总线上的 IOPB 分支线上的, 所以要打开 IOPBEN 这个分支, 这样做的好处是你要打开哪组 GPIO 就打开哪组分支, 不用的不打开这样可以低功耗

这样完成了一个真正点亮 LED 的方法是不是很麻烦呢。对寄存器操作 LED 就是这么麻烦

STM32 固件库函数配置时钟

固件库函数代码

```

02
03
04 void RCC_Configuration()
05 {
06     ErrorStatus HSEstartUpStatus; // 定义外部晶振是否正常启动的状态标志
07     RCC_DeInit(); // 复位RCC外部晶振
08     RCC_HSEConfig(RCC_HSE_ON); // 打开外部HSE高速晶振
09     HSEstartUpStatus = RCC_WaitForHSEStartUp(); // 等待外部高速时钟就绪
10     if(HSEstartUpStatus == SUCCESS) // 如果外部晶振就绪就可以操作flash和分频器
11     {
12         FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable); // 开启flash
13         FLASH_SetLatency(FLASH_Latency_2); // flash延时
14
15         RCC_HCLKConfig(RCC_SYSCLK_Div1); // 配置AHB总线分频
16         RCC_PCLK2Config(RCC_HCLK_Div1); // 配置APB2总线分频
17         RCC_PCLK1Config(RCC_HCLK_Div2); // 配置APB1总线分频
18         RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9); // 第一个参数RCC_PLLSource_HSE_Div1选择那一路锁相环作为输入,
19         // 第二个参数RCC_PLLMul_9设置倍频系数
20         RCC_PLLCmd(ENABLE); // 使能PLL时钟
21         while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY)==Status); // 等待PLL时钟启动稳定就绪
22         RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK); // 选择PLL作为系统输入
23         if(RCC_GetSYSCLKSource() != 0x08); // 0x08: PLL 作为系统时钟
24
25     }
26 }

```

我们选择黄色的
线, HSE 不经过
/2, 而是直接到
PLLSRC

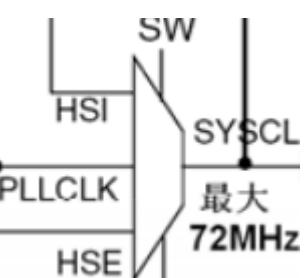
寄存器操作代码

MYRCC_DeInit();
RCC->CR1=0x00010000
while(! (RCC->CR>>17));

FLASH->ACR|=0x32;

RCC->CFGR=0x00000400;
RCC->CFGR1=1<<16;

RCC->CR1=0x01000000;
while(! (RCC->CR>>25));
RCC->CFGR1=0x00000002;



不管是寄存器还是固件库其实都是一样的就是初始化系统时钟, 只是固件库是人看的。

如何用固件库操作 GPIO

最先的章节我们讲了位绑定操作 GPIOA14 和 GPIOB14 引脚，现在我们用固件库来操作。首先 GPIOA 和 B 都是挂在 APB2 时钟总线上面的，所以我们要使能该总线。

GPIO_InitTypeDef structure

GPIO_InitTypeDef 定义于文件“*stm32f10x_gpio.h*”：

```
typedef struct
{
    u16 GPIO_Pin;
    GPIO_Speed_TypeDef GPIO_Speed;
    GPIO_Mode_TypeDef GPIO_Mode;
} GPIO_InitTypeDef;
```

Table 373. RCC_AHB2Periph 值

RCC_AHB2Periph	描述
RCC_APB2Periph_AFIO	功能复用 IO 时钟
RCC_APB2Periph_GPIOA	GPIOA 时钟
RCC_APB2Periph_GPIOB	GPIOB 时钟
RCC_APB2Periph_GPIOC	GPIOC 时钟
RCC_APB2Periph_GPIOD	GPIOD 时钟
RCC_APB2Periph_GPIOE	GPIOE 时钟
RCC_APB2Periph_ADC1	ADC1 时钟
RCC_APB2Periph_ADC2	ADC2 时钟
RCC_APB2Periph_TIM1	TIM1 时钟
RCC_APB2Periph_SPI1	SPI1 时钟
RCC_APB2Periph_USART1	USART1 时钟
RCC_APB2Periph_ALL	全部 APB2 外设时钟

例：

```
/* Enable GPIOA, GPIOB and SPI1 clocks */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
RCC_APB2Periph_SPI1, ENABLE);
```

GPIO 方向	索引	模式	设置	模式代码
GPIO Input	0x00	GPIO_Mode_AIN	0x00	0x00
		GPIO_Mode_IN_FLOATING	0x04	0x04
		GPIO_Mode_IPD	0x08	0x28
		GPIO_Mode_IPU	0x08	0x48
GPIO Output	0x01	GPIO_Mode_Out_OD	0x04	0x14
		GPIO_Mode_Out_PP	0x00	0x10
		GPIO_Mode_AF_OD	0x0C	0x1C
		GPIO_Mode_AF_PP	0x08	0x18

例：

```
/* Configure all the GPIOA in Input Floating mode */
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

```
void main()
{
    RCC_Configuration(); // 启动系统时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB | RCC_APB2Periph_AFIO, ENABLE);
    // 启动 GPIOA 和 GPIOB 时钟，和功能复用时钟，这样才能操作引脚电平，如果你要在使能一个 GPIOC，在后面 + 一下就是了。
    GPIO_InitTypeDef GPIO_B;
    GPIO_B.GPIO_Pin = GPIO_Pin_14; // GPIOB14 引脚
    GPIO_B.GPIO_Speed = GPIO_Speed_50MHz; // IO 口输出 50M 速率
    GPIO_B.GPIO_Mode = GPIO_Mode_Out_PP; // 推挽输出
    GPIO_Init(GPIOB, &GPIO_B); // 这里是确定初始化哪组 GPIO，我这里是 B

    GPIO_InitTypeDef GPIO_A;
    GPIO_A.GPIO_Pin = GPIO_Pin_14; // GPIOA14 引脚
    GPIO_A.GPIO_Speed = GPIO_Speed_50MHz; // IO 口输出 50M 速率
    GPIO_A.GPIO_Mode = GPIO_Mode_Out_PP; // 推挽输出
    GPIO_Init(GPIOA, &GPIO_A); // 这里是确定初始化哪组 GPIO，我这里是 A

    while(1){
        GPIO_SetBits(GPIOA, GPIO_Pin_14); // GPIOA14 输出 1
        GPIO_SetBits(GPIOB, GPIO_Pin_14); // GPIOB14 输出 1

        GPIO_ResetBits(GPIOA, GPIO_Pin_14); // GPIOA14 输出 0
        GPIO_ResetBits(GPIOB, GPIO_Pin_14); // GPIOB14 输出 0
    }
}
```

以上就是库函数来设置 GPIO 点亮 LED 的方法，是不是适合人类阅读。

我觉得库函数和寄存器是可以混合使用的，各有各的好。

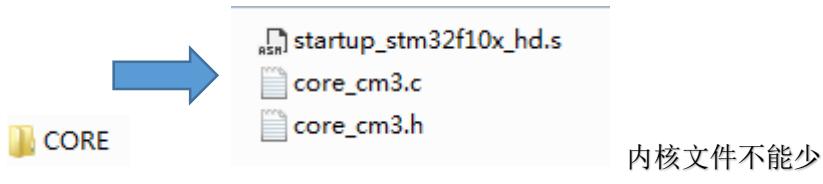
STM32 IO 口作为输入寄存器方式

现在用按键为例子

CNF3[1:0]	MODE3[1:0]	CNF2[1:0]	MODE2[1:0]	CNF1[1:0]	MODE1[1:0]	CNF0[1:0]	MODE0[1:0]
rw	rw	rw	rw	rw	rw	rw	rw
位31:30 CNFy[1:0]: 端口x配置位(y = 0...7) (Port x configuration bits) 软件通过这些位配置相应的I/O端口, 请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式							
位29:28 MODEy[1:0]: 端口x的模式位(y = 0...7) (Port x mode bits) 软件通过这些位配置相应的I/O端口, 请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式, 最大速度10MHz 10: 输出模式, 最大速度2MHz 11: 输出模式, 最大速度50MHz							
3	#define GPIO_IDR_A (GPIOA_BASE+0x08) //定义GPIOA的基址	4	#define BitBand(addr,bitnum) *((volatile unsigned long *) ((addr	5		6	
7	#define PAout(n) BitBand(GPIO_ODR_A,n)	8	#define PAin(n) BitBand(GPIO_IDR_A,n)	9			
0	int main()	1	{	2	RCC->APB2ENR = 1<<0; //GPIO设置成输入也要打开该端口的时钟总线	3	GPIOA->CRL&=0xfffffffff0; //先将GPIOA的第5个引脚模式清0
4	GPIOA->CRL = 0x00000008; //设置GPIOA5口为输入模式, 上下拉电阻	5	GPIOA->ODR = 1<<0; //默认输入1	6		7	
7	if(PAin(0)==0)	8	{	9	//GPIOA5收到低电平信号	0	}
1	return 0;	2	}	3			

这样就可以得到 IO 口输入信号了。

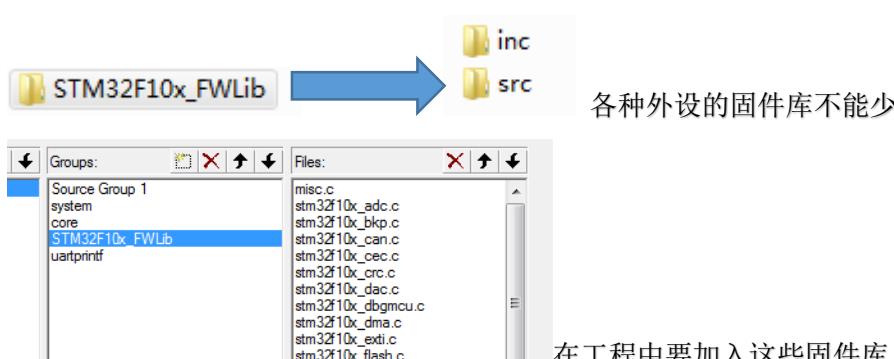
STM32 固件库工程需要的基本头文件和 C 文件



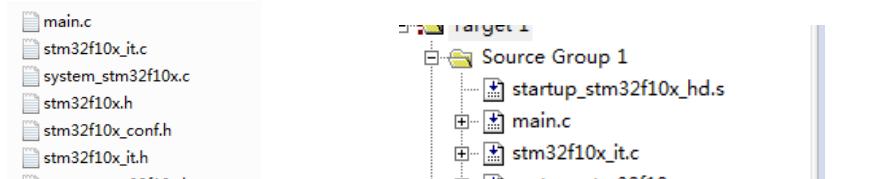
在项目中要加入内核 C 文件 core_cm3.c



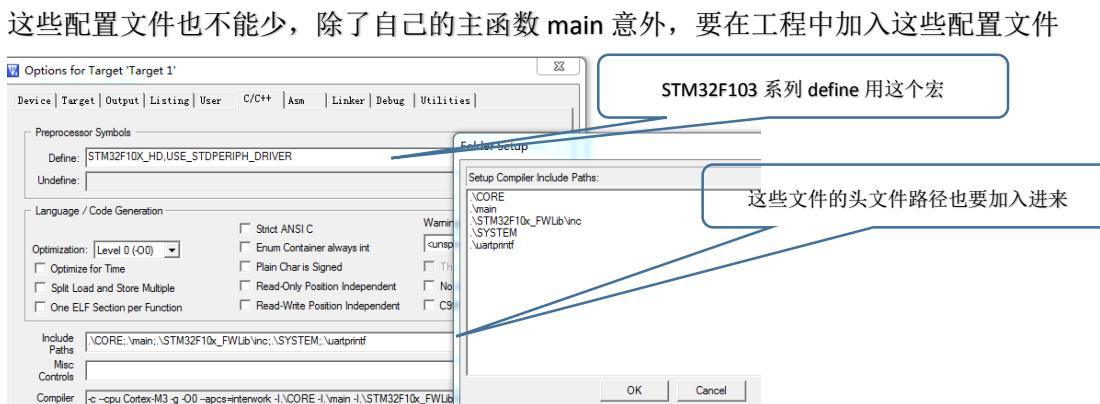
在工程中还要加入 s 文件



在工程中要加入这些固件库 c 文件



这些配置文件也不能少, 除了自己的主函数 main 意外, 要在工程中加入这些配置文件



Stm32f10x.h 这个头文件最好是你写的 C 文件里面都要包含，因为里面有很多官方寄存器代码，这样你在写程序的时候觉得有些地方固件库不爽，想在某个代码段操作寄存器，就可以成功操作。串口实验章节我就用了这种方法，固件库和寄存器配合使用。

STM32 IO 口作为输入库函数方式

```
//库函数版本按键输入
int main()
{
    GPIO_InitTypeDef GPIO_InitStructure;
    u8 ReadValue;
    RCC_Configuration(); //打开系统时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //打开GPIOA时钟

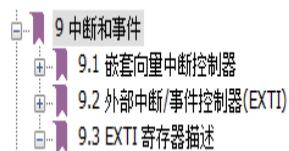
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //上拉输入模式
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    ReadValue = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0); //读GPIOA0端口的电平

    return 0;
}
```

STM32 中断系统

STM32 的 cortex-m3 内核开放了 60 个中断可以操作



0	7	可设置	WWDG	窗口定时器中断
1	8	可设置	PVD	连到EXTI的电源电压检测(PVD)中断
2	9	可设置	TAMPER	侵入检测中断
3	10	可设置	RTC	实时时钟(RTC)全局中断
4	11	可设置	FLASH	闪存全局中断
5	12	可设置	RCC	复位和时钟控制(RCC)中断
6	13	可设置	EXTI0	EXTI线0中断

这就是可以操作的中断的一部分，至于更多中断号，请看数据手册。

MDK(keil)软件与 NVIC 相关的寄存器定义了一个结构体

```
typedef struct
{
    vu32 ISER[2];
    u32 RESERVED0[30];
    vu32 ICER[2];
    u32 RESERVED1[30];
    vu32 ISPR[2];
    u32 RESERVED2[30];
    vu32 ICPR[2];
    u32 RESERVED3[30];
    vu32 IABR[2];
    u32 RESERVED4[62];
    vu32 IPR[15];
} NVIC_TypeDef;
```

我们知道 STM32 有 60 个外设中断，我们 ISER 寄存器用了 64 个位来设置中断。但是 STM32 只用了 ISER 前 60 位中断

因为 ISER 中断无法写 0 清除中断位，只有使用 ICER 来清除位

挂起现在执行的中断，转而执行更高优先级别的中断

接触挂起的中断，继续执行

如果软件想知道那个中断在执行，就查看这个寄存器位

中断优先级控制寄存器，很重要，下面细看

ISER[0]对应 0 ~ 31 号中断

ISER[1]的 0 ~ 27 位对应 32 ~ 59 号中断，如果要使用某一位中断，就要将 ISER 里面对应位置 1

比如我们要使用 5 号中断,也就是 RCC 中断.



这样就可以使用 5 号中断了.

0	7	可设置	WWDG	窗口定时器中断	0x0000_0040
1	8	可设置	PVD	连到EXTI的电源电压检测(PVD)中断	0x0000_0044
2	9	可设置	TAMPER	侵入检测中断	0x0000_0048
3	10	可设置	RTC	实时时钟(RTC)全局中断	0x0000_004C
4	11	可设置	FLASH	闪存全局中断	0x0000_0050
5	12	可设置	RCC	复位和时钟控制(RCC)中断	0x0000_0054
6	13	可设置	EXTI0	EXTI线0中断	0x0000_0058
7	14	可设置	EXTI1	EXTI线1中断	0x0000_005C
8	15	可设置	EXTI2	EXTI线2中断	0x0000_0060
9	16	可设置	EXTI3	EXTI线3中断	0x0000_0064
10	17	可设置	EXTI4	EXTI线4中断	0x0000_0068
11	18	可设置	DMA1通道1	DMA1通道1全局中断	0x0000_006C

如果要使用 EXTI3 中断那么

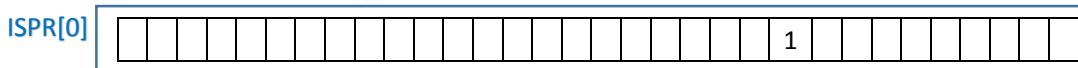


这样就可以使用 EXTI3 中断了

但是有个问题,如果我要将设置的中断位清除掉,不用这位中断了.如果向该位写 0 是无效的.
所以要解决改问题就必须使用 ICER 寄存器



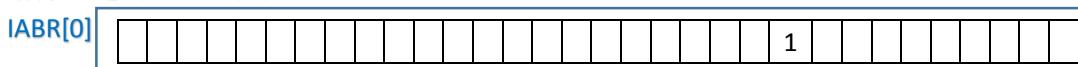
向该位写 1 来清除 EXTI3 中断,这样 EXTI3 中断就没法用了.



向第 9 位置 1,挂起了 EXTI3 中断.



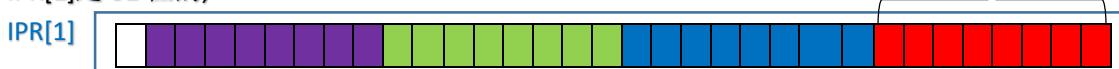
解除挂起的 EXTI3 中断



EXTI3 中断还在执行

IPR 寄存器是控制中断优先级的,每一个中断占用 8 位,IPR[15]=15*4=60,可以控制 60 个中断.
为什么呢?

IPR[1]是 32 位的,



你看 IPR[1]就包含了 4 个中断,所以 IPR[15]就等于 60 个中断.

所以 IPR[0] 的 31~24 对应中断 3

IPR[0]的 23~16 对应中断 2 IPR[0]的 15~8 对应中断 1

IPR[0]的 7~0 对应中断 0 依次类推

但是每个中断优先级的 8 位并没有全部用上,所以啊!!!! 要看 SCB->AIRCR 寄存器.

优先级	优先级	优先级	优先级	无效	无效	无效	无效
-----	-----	-----	-----	----	----	----	----

每个中断优先级的 8 位寄存器只用了前面 4 位来做抢占优先级和响应优先级.

中断源的中断优先级分配到那一组,SCB->AIRCR 来定义的

组	AIRCR[10: 8]	bit[7: 4]分配情况	分配结果
0	111	0: 4	0 位抢占优先级, 4 位响应优先级
1	110	1: 3	1 位抢占优先级, 3 位响应优先级
2	101	2: 2	2 位抢占优先级, 2 位响应优先级
3	100	3: 1	3 位抢占优先级, 1 位响应优先级
4	011	4: 0	4 位抢占优先级, 0 位响应优先级

如果分配到 0 组,那么抢占优先级不占,只要 4 位响应优先级可以操作.

比如,我现在有个外设中断 EXTIO,把该中断分配到第 4 组,那么这个中断的抢占优先级就可以设置 0~15,0 代表优先级最高,15 代表优先级最低.就是数字越小代表优先级越高.

优先级组 抢占优先级 响应优先级

EXTIO	2	3	1
RCC	2	4	0
DMA1	2	3	0

优先级顺序 DMA1 > EXTIO > RCC

DMA 中断到了可以打断 RCC 正在执行的中断程序,然后执行 DMA 中断程序.

也就是高抢占优先级可以打破低抢占优先级的中断.

优先级组 抢占优先级 响应优先级

EXTIO	2	3	0
DMA1	2	3	0

如果外设的抢占优先级一样,然后响应优先级一样,那么谁的中断先到,就先执行谁.

优先级组 抢占优先级 响应优先级

EXTIO	2	3	1
DMA1	2	3	0

就算 DMA1 响应优先级高于 EXTIO,也不能打断 EXTIO 的中断程序,必须等到 EXTIO 中断程序执行完了,才能再执行 DMA1 中断程序.只有抢占优先级的高低之分才能打断中断程序.

STM32 中断寄存器实现

```
1 #include "stm32f10x.h"
2
3
4 void MY_NVIC_PriorityGroupConfig(u8 NVIC_group) //只能分配0 ~ 4 组
5 {
6     u32 temp,temp1;
7     temp1=(~NVIC_group)&0x07;//取后三位
8     temp1<=8;
9     temp=SCB->AIRCR; //读取芯片上电后默认的设置
10    temp&=0x0000F8FF;//清空芯片上电后默认的分组
11    temp|=0x05fa0000;//写入钥匙
12    temp|=temp1;
13    SCB->AIRCR=temp;//设置分组
14 }
```

这里只能填入
0~4

为什么呢?

因为前面写到了
中断优先级只占
用 8 位里面的四

注意:一个工程里面只能有一个中断组,比如你一个工程项目里面有三个中断,那么这三个中断要么全部放在组 1,要么放在组 2,不能一个中断一个分组,这样是不行的.所以如果中断多建议分组 4.

比如我一个工程所以中断分到了 4 组

```

1 #include "stm32f10x.h"
2
3
4 void MY_NVIC_PriorityGroupConfig(u8 NVIC_group) //只能分配0 ~ 4 组
5 {
6     u32 temp,temp1;
7     temp1=(~NVIC_group)&0x07;//取后三位
8     temp1<<=8;
9     temp=SCB->AIRCR; //读取芯片上电后默认的设置
10    temp&=0x0000F8FF;//清空芯片上电后默认的分组
11    temp|=0x05fa0000;//写入钥匙
12    temp|=temp1;
13    SCB->AIRCR=temp;//设置分组
14 }

```

这个芯片有钥匙所以优先级值要和钥匙一起写入 AIRCR 寄存器

4 取反 011
temp=011&111
temp=011

将 011 写入 AIRCR
寄存器 10:8 位

位段	名称	类型	复位值	描述
31:16	VECTKEY	RW	-	访问钥匙：任何对该寄存器的写操作，都必须同时把 0x05FA 写入此段，否则写操作被忽略。若读取此半字，则 0xFA05
15	ENDIANESS	R	-	指示端设置。1 = 大端(BE8)，0 = 小端。此值是在复位时确定的，不能更改。
10:8	PRIGROUP	R/W	0	优先级分组
2	SYSRESETREQ	W	-	请求芯片控制逻辑产生一次复位
1	VECTCLRACTIVE	W	-	清零所有异常的活动状态信息。通常只在调试时用，或者在 OS 从错误中恢复时用。
0	VECTRESET	W	-	复位 CM3 处理器内核（调试逻辑除外），但是此复位不影响芯片上在内核以外的电路

所以为什么一个工程所有中断只能用一个组,就是这个原因.你一次性写入 AIRCR 寄存器就只有一组.

```

5 void MY_NVIC_INIT(u8 NVIC_PreemptionPriority,u8 NVIC_subpriority,u8 NVIC_channel,u8 NVIC_Group)
6 {
7     u32 temp;
8     u8 IPRADDR=NVIC_channel/4; //每组只能存4个，得到组地址
9     u8 IPROFFEST=NVIC_channel%4;//在组内的偏移
10    IPROFFEST=IPROFFEST*8+4;//得到偏移的确切位置
11    MY_NVIC_PriorityGroupConfig(NVIC_Group); //设置分组
12    temp=NVIC_PreemptionPriority<<(4-NVIC_Group);
13    temp|=NVIC_subpriority&(0x0f>>NVIC_Group);
14    temp&=0x0f;//取低4位
15    if(NVIC_channel<32)
16    {
17        NVIC->ISER[0] |=1<<NVIC_channel;//使能中断位，要清除中断执行相反操作
18    }
19    else
20    {
21        NVIC->ISER[1] |=1<<(NVIC_channel-32);
22    }
23    NVIC->IP[IPRADDR] |=temp<<IPROFFEST;//设置响应优先级和抢占优先级
24
25
26 }

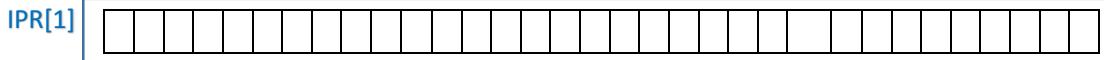
```

传入一个参数 MY_NVIC_INIT(2,1,5,2); 抢占优先级 2,响应优先级 1,RCC 中断号 5,分到 2 组

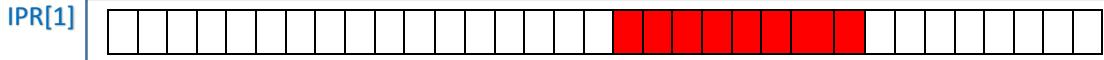
以这个为例我们来计算函数实现

u8 IPRADDR=NVIC_channel/4; //IPR 寄存器每组只有 4 个中断号.RCC 中断号是 5 所以在 IPR 寄存器第 0 个中断号,1=5/4,所以 IPRADDR=1 也就是数组 IPR[1];

数组 1 也有 4 个 8 位放在第几个 8 位? u8 IPREOFFEST=NVIC_channel%4;这个得到的就是第几个 8 位.



IPREOFFEST=IPREOFFEST*8;//得到偏移确切位置



IPREOFFEST=IPREOFFEST*8+4;//因为低 4 位不能用所以只能用高 4 位



MY_NVIC_PriorityGroupConfig(NVIC_Group);//设置分组

temp=NVIC_PreemptionPriority<<(4-NVIC_Group); //设置抢占优先级.

计算 2=4-2

temp=2<<2 //2<<10 = 1000

temp=1000

temp|=NVIC_subpriority&(0x0f>>NVIC_Group); //写入响应优先级

0000 1111>> NVIC_Group

temp|=NVIC_subpriority&(00000011)

NVIC_subpriority&(00000011) = 0001&(0011)

temp|=1001

temp=1000|1001

temp=1001

temp&=0x0f; //temp=1001 & 0000 1111

temp=1001;

if(NVIC_channel<32)

{

NVIC->ISER[0]|=1<<NVIC_channel;//使能中断

}

else

{

NVIC->ISER[1]|=1<<(NVIC_channel-32);

}

NVIC->IP[IPRADDR]|=temp<<IPREOFFEST; //设置响应优先级和抢占优先级

NVIC->IP[IPRADDR]|=1001<<IPREOFFEST; 将 1001 放入 IPR 寄存器 10 01

组	AIRCR[10: 8]	bit[7: 4]分配情况	分配结果
0	111	0: 4	0 位抢占优先级, 4 位响应优先级
1	110	1: 3	1 位抢占优先级, 3 位响应优先级
2	101	2: 2	2 位抢占优先级, 2 位响应优先级
3	100	3: 1	3 位抢占优先级, 1 位响应优先级
4	011	4: 0	4 位抢占优先级, 0 位响应优先级

这样中断优先级和中断号都打开了

前面是设置的中断优先级现在我们来打开外设的中断控制器

在 STM32F10x.h 文件中定义了中断寄存器

typedef struct

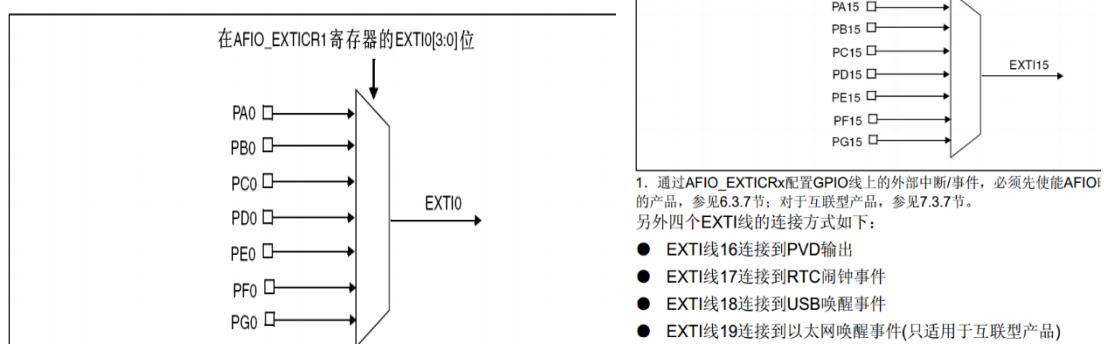
```
{  
    __IO uint32_t IMR;  
    __IO uint32_t EMR;  
    __IO uint32_t RTSR;  
    __IO uint32_t FTSR;  
    __IO uint32_t SWIER;  
    __IO uint32_t PR;  
} EXTI_TypeDef;
```

IMR 寄存器是控制线上的中断

外部中断/事件线路映像

112通用I/O端口以下图的方式连接到16个外部中断/事件线上:

图20 外部中断通用I/O映像



STM32F103 的 IMR 寄存器一共有 19 个中断线 EXIT0 ~ EXIT19

9.3.1 中断屏蔽寄存器(EXTI_IMR)

偏移地址: 0x00

复位值: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16		
保留														MR19	MR18	MR17	MR16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR4	MR4	MR3	MR2	MR1	MRO		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		
位31:20 保留，必须始终保持为复位状态(0)。																	
位19:0 MRx: 线x上的中断屏蔽 (Interrupt Mask on line x) 0: 屏蔽来自线x上的中断请求； 1: 开放来自线x上的中断请求。 注：位19只适用于互联型产品，对于其它产品为保留位。																	

STM32 很麻烦，如果你想取消某一线上的中断，不能直接写 0，必须操作另外一个寄存器来屏蔽线上中断，这个寄存器就是 EMR 寄存器

9.3.2 事件屏蔽寄存器(EXTI_EMR)

偏移地址: 0x04

复位值: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16		
保留														MR19	MR18	MR17	MR16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR4	MR4	MR3	MR2	MR1	MRO		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		
位31:20 保留，必须始终保持为复位状态(0)。																	
位19:0 MRx: 线x上的事件屏蔽 (Event Mask on line x) 0: 屏蔽来自线x上的事件请求； 1: 开放来自线x上的事件请求。 注：位19只适用于互联型产品，对于其它产品为保留位。																	

RTSR;寄存器是上升沿触发寄存器。

9.3.3 上升沿触发选择寄存器(EXTI_RTSR)

偏移地址: 0x08

复位值: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留														TR19	TR18
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
位31:19 保留, 必须始终保持为复位状态(0)。															
位18:0 TRx: 线x上的上升沿触发事件配置位 (Rising trigger event configuration bit of line x) 0: 禁止输入线x上的上升沿触发(中断和事件) 1: 允许输入线x上的上升沿触发(中断和事件) 注: 位19只适用于互联型产品, 对于其它产品为保留位。															

FTSR;下降沿触发寄存器。

9.3.4 下降沿触发选择寄存器(EXTI_FTSR)

偏移地址: 0x0C

复位值: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留														TR19	TR18
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
位31:19 保留, 必须始终保持为复位状态(0)。															
位18:0 TRx: 线x上的下降沿触发事件配置位 (Falling trigger event configuration bit of line x) 0: 禁止输入线x上的下降沿触发(中断和事件) 1: 允许输入线x上的下降沿触发(中断和事件) 注: 位19只适用于互联型产品, 对于其它产品为保留位。															

SWIER;该寄存器不说

PR: 查询中断事件寄存器

9.3.6 挂起寄存器(EXTI_PR)

偏移地址: 0x14

复位值: 0xFFFF FFFF

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留														PR19	PR18
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc	wl	rc	wl	rc	wl	rc	wl	rc	wl	rc	wl	rc	wl	rc	wl
位31:19 保留, 必须始终保持为复位状态(0)。															
位18:0 PRx: 挂起位 (Pending bit) 0: 没有发生触发请求 1: 发生了选择的触发请求 当在外部中断线上发生了选择的边沿事件, 该位被置'1'。在该位中写入'1'可以清除它, 也可以通过改变边沿检测的极性消除。 注: 位19只适用于互联型产品, 对于其它产品为保留位。															

STM32 IO 口中断有个问题就是, 并不是每个 GPIO 口都可以分配一个中断函数, 而是几个 IO 口共用一个中断函数, 也可以说说几个 IO 口共用一个中断线, 比如 GPIOA0~GPIOG0 就是用的 EXTI0, 那么 GPIOA15~GPIOG15 用的是 EXTI15, 所以我们要映射中断线和 IO 口的关系。

8.4.4 外部中断配置寄存器 2(AFIO_EXTICR2)

地址偏移: 0x0C

复位值: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI17[3:0]	EXTI16[3:0]	EXTI15[3:0]	EXTI14[3:0]	EXTI13[3:0]	EXTI12[3:0]	EXTI11[3:0]	EXTI10[3:0]	EXTI9[3:0]	EXTI8[3:0]	EXTI7[3:0]	EXTI6[3:0]	EXTI5[3:0]	EXTI4[3:0]	EXTI3[3:0]	EXTI2[3:0]
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
位31:16 保留 位15:0 EXTIx[4:0] = 4 - EXTIx[3:0] (EXTIx[3:0] = 4 - EXTIx[3:0] configuration) 这些位可由软件清零, 用于选择EXTIx分组的中断输入源。 0000: PA0[4]引脚 0100: PE1[4]引脚 0001: PB0[4]引脚 0101: PE1[4]引脚 0010: PC0[4]引脚 0110: PG1[4]引脚 0011: PD0[4]引脚															

- 8 通用和复用功能I/O(GPIO和AFIO)
 - 8.1 GPIO功能描述
 - 8.2 GPIO寄存器描述
 - 8.3 复用功能I/O和调试配置(AFIO)
 - 8.4 AFIO寄存器描述
 - 8.4.1 事件控制寄存器(AFIO_EVCR)
 - 8.4.2 复用重映射和刷写I/O配置寄存器(AFIO_MAPR)
 - 8.4.3 外部中断配置寄存器1(AFIO_EXTICR1)
 - 8.4.4 外部中断配置寄存器2(AFIO_EXTICR2)
 - 8.4.5 外部中断配置寄存器3(AFIO_EXTICR3)
 - 8.4.6 外部中断配置寄存器4(AFIO_EXTICR4)

```

typedef struct
{
    __IO uint32_t EVCR;
    __IO uint32_t MAPR;
    __IO uint32_t EXTICR[4];
    uint32_t RESERVED0;
    __IO uint32_t MAPR2;
} AFIO_TypeDef;

```

EXTICR[4];寄存器就是映射 IO 和中断线关系的。

8.4.3 外部中断配置寄存器 1(AFIO_EXTICR1)

地址偏移: 0x08

复位值: 0x0000



位15:0	EXTIx[3:0]: EXTIx配置(x = 0 ... 3) (EXTIx configuration)	
	这些位可由软件读写，用于选择EXTIx外部中断的输入源。参看9.2.5节。	
	0000: PA[x]引脚	0100: PE[x]引脚
	0001: PB[x]引脚	0101: PF[x]引脚
	0010: PC[x]引脚	0110: PG[x]引脚
	0011: PD[x]引脚	

比如我要使用 GPIOA0 作为中断，那么我们就写入 0000 这样数字到 EXTIO 寄存器中

比如我要使用 GPIOA1 作为中断，那么我们就写入 0000 这样数字到 EXTI1 寄存器中

比如我要使用 GPIOB0 作为中断，那么我们就写入 0001 这样数字到 EXTIO 寄存器中

```

19
20 void Ex_NVIC_config(u8 GPIOx,u8 BITx,u8 TRIM)//GPIOx代表第几个GPIO组 A~G , BITx代表这个GPIO组第几位TRIM触发边沿选择
21 {
22     u8 EXTADDR;
23     u8 EXTOFFSET;
24     EXTADDR=BITx/4;//得到中断寄存器组的编号
25     EXTOFFSET=(BITx%4)*4;
26     RCC->APB2ENR|=0x01;//使能io复用时钟
27     AFIO->EXTICR[EXTADDR]|=GPIOx<<EXTOFFSET;//EXTI第几个中断线映射到GPIO第几个引脚
28
29     EXTI->IMR|=1<<BITx;//开启某个EXTI线上的中断
30     EXTI->EMR|=1<<BITx;//不屏蔽某个线上的中断
31
32     if(TRIM&0X01)EXTI->FTSR|=1<<BITx;//下降沿触发
33     if(TRIM&0X02)EXTI->RTSR|=1<<BITx;//上升沿触发
34
35 }

```

这个就是设置 IO 口外部中断的函数(寄存器版)

比如执行 Ex_NVIC_config(GPIOA,5,0X01);

```

Ex_NVIC_config(GPIOA,5,0X01); //该函数的计算方式
EXTADDR=BITx/4;//得到中断寄存器组
EXTADDR=5/4=1
EXTOFFSET=(BITx%4)*4;
EXTOFFSET=(5%4)*4=4
AFIO->EXTICR[EXTADDR]|=GPIOx<<EXTOFFSET; //GPIOx=GPIOA=0 0<<4 从 EXTI4 左移到 EXTI5

```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXT17[3:0]				EXTI6[3:0]				EXTI5[3:0]				EXTI4[3:0]			
RW	RW	RW	RW												

EXTICR[1]

这样外部 IO 口的中断就设置完成了。这个函数调用一次只能配置一个 IO 口，如果有多个 IO 口就要调用多次。

```

void EXTI15_10_IRQHandler(void)
{
    EXTI->PR=1<<10; //清除中断线10上的标准位,如果不清楚标志位下一次中断就没有反应了
}

int main()
{
    RCC->APB2ENR=1<<5;
    GPIOD->CRH&=0xfffff0ff;//设置GPIOD10
    GPIOD->CRH|=0x00000800;//设置GPIOD10为输入
    GPIOD->ODR|=1<<10;//设置GPIOD10为输入上拉
    Ex_NVIC_config(GPIO_D,10,FTIR);//申请GPIOD10为EXTI10中断线, 设置下降沿触发
    MY_NVIC_INIT(2,1,EXTI15_10_IRQHandler,2);//抢占优先级2, 响应优先级1, , 第2组 EXTI15_10_IRQHandler断号在STM32f10x.h头文件里面

    //外部输入电平变化就会进入EXTI15_10_IRQHandler中断函数
    return 0;
}

```

这就是中断处理函数，这里要提醒一下 EXTI15_10_IRQHandler 中断号要去 STM32F10x.h 里面找，中断服务函数 EXTI15_10_IRQHandler 要去 startup_stm32f10x_hd.s 汇编文件里面去找。

还有就是 STM32F1 的外部中断 EXTI 0~4 都有单独的中断服务函数，但是从 5 开始，他们就没有单独的服务函数了，而是多个中断共用一个服务函数，比如外部中断 5~9 的中断服务函数为： void EXTI9_5_IRQHandler(void)，类似的， void EXTI15_10_IRQHandler(void) 就是外部中断 10~15 的中断服务函数，所以 5~9 和 10~15 中断线是共享中断，在服务函数里面要去判断哪个 IO 口按下了。

其实寄存器使用中断很麻烦除了 GPIO 操作，中断，SPI 这些外设还是建议使用固件库函数

STM32 中断固件库操作

```
1 ///////////////STM32外部中断固件库操作///////////////
2 int main()
3 {
4     GPIO_InitTypeDef GPIO_InitStructure; //GPIO结构体
5     EXTI_InitTypeDef EXTI_InitStructure; //中断结构体
6     NVIC_InitTypeDef NVIC_InitStructure; //中断优先级结构体
7
8     //记得调用RCC函数启动RCC时钟//
9     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE); //PD10引脚做为按键输入，所以打开GPIOD时钟
10
11     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10; //PD10作为按键输入
12     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //PD10作为上拉输入
13     GPIO_Init(GPIOD, &GPIO_InitStructure);
14
15     //////////////设置引脚中断功能/////////////
16     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); //因为PD10是中断引脚，不是普通的GPIO所以要开启IO口复用功能时钟
17
18     EXTI_ClearITPendingBit(EXTI_Line2); //清除中断挂起位，打开中断之前一定要清楚该线上中断的挂起位
19     EXTI_InitStructure.EXTI_Line = EXTI_Line10; //申请中断线 我们是PD10，所以是中断线10
20     EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //设置为中断触发，因为是检测外部的中断，如果是内部事件，就用事件触发EXTI_Mode_Event
21     EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //设置为下降沿触发
22     EXTI_InitStructure.EXTI_LineCmd = ENABLE;
23     EXTI_Init(&EXTI_InitStructure);
24
25     //////////////设置引脚中断功能/////////////
26     GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource10); /*因为PD10的中断线是10~15 所以这里要声明下10~15中断线的，声明那个IO口作为中断输入
27 Line10作为中断源*/
28
29     //////////////设置中断优先级/////////////
30
31     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); //将中断分配到0组
32     NVIC_InitStructure.NVIC_IRQChannel = EXTI15_10_IRQn; //因为PD10是挂机到line10的，正好line10线EXTI15~10下
33     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1; //设置响应优先级
34     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //使能中断优先级分组
35     NVIC_Init(&NVIC_InitStructure);
36
37 }
```

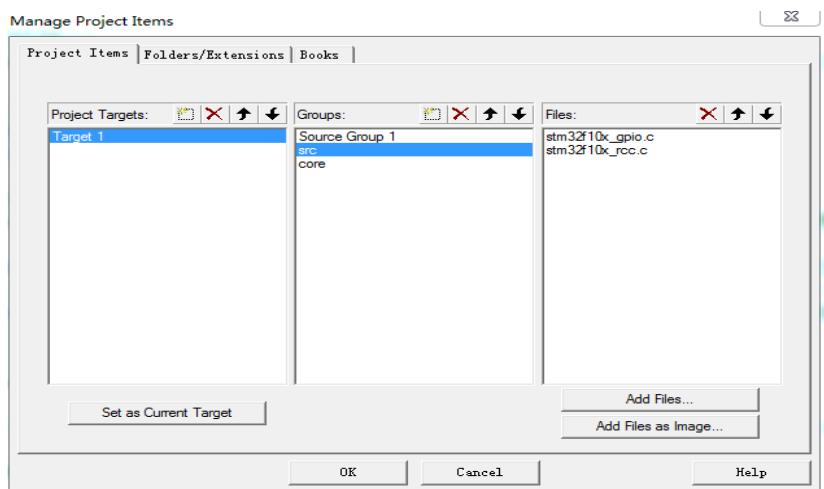
```
3 //////////////设置引脚中断功能/////////////
4 RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); //因为PD10是中断引脚，不是普通的GPIO所以要开启IO口复用功能时钟
5
6 EXTI_ClearITPendingBit(EXTI_Line2); //清除中断挂起位，打开中断之前一定要清楚该线上中断的挂起位
7 EXTI_InitStructure.EXTI_Line = EXTI_Line10; //申请中断线 我们是PD10，所以是中断线10
8 EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //设置为中断触发，因为是检测外部的中断，如果是内部事件，就用事件触发EXTI_Mode_Event
9 EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //设置为下降沿触发
10 EXTI_InitStructure.EXTI_LineCmd = ENABLE;
11 EXTI_Init(&EXTI_InitStructure);
12
13 GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource10); /*因为PD10的中断线是10~15 所以这里要声明下10~15中断线的，声明那个IO口作为中断输入
14 Line10作为中断源*/
```

编译报错

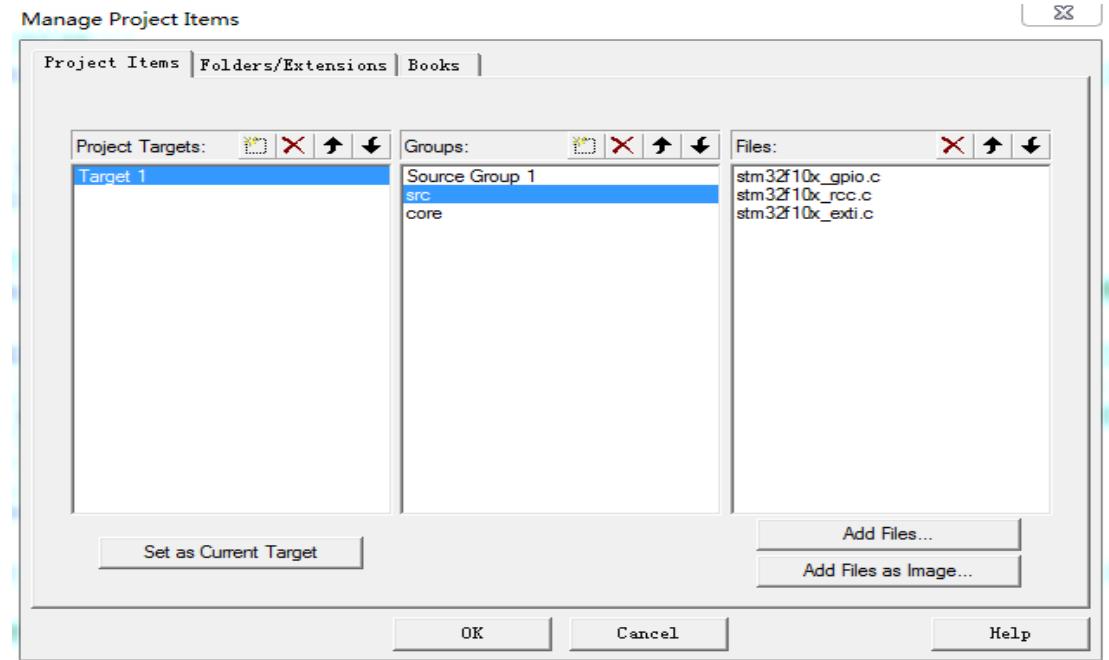
```
Build Output
```

```
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 4 error messages.
".\out\interrupt.axf" - 4 Error(s), 0 Warning(s).
Target not created
```

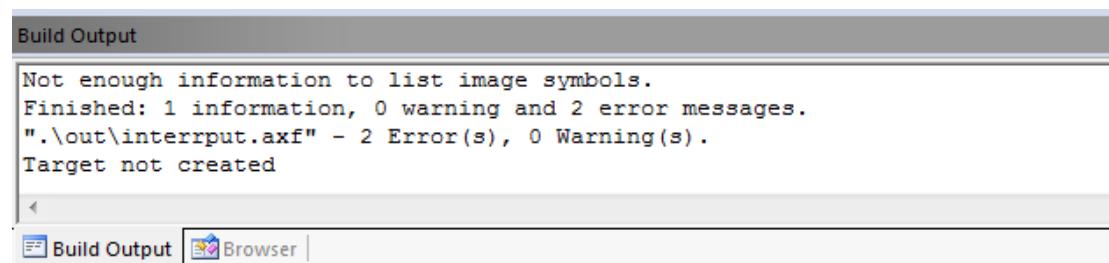
这是因为你用了固件库的 EXIT 中断，所以要加 EXIT 源文件



加上 EXTI 文件之后

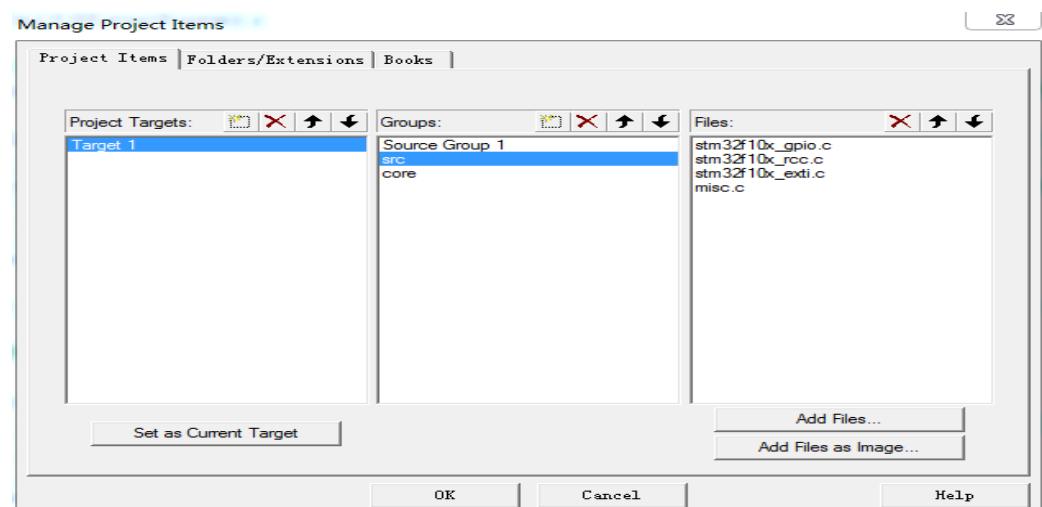


编译还是报错



少了两个错，是因为 exti 文件解决了，但是中断优先级设置 misc 没有加上

```
5 ///////////////////////////////////////////////////////////////////设置中断优先级/////////////////////////////////////////////////////////////////
6
7 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); //将中断分配到0组
8 NVIC_InitStructure.NVIC IRQChannel = EXTI15_10_IRQn; //因为PD10是挂机到line10的，正好line10线EXTI15~10下
9 NVIC_InitStructure.NVIC IRQChannelSubPriority = 1; //设置响应优先级
0 NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE; //使能中断优先级分组
1 NVIC_Init(&NVIC_InitStructure);
2
```



加了 misc 之后，就可以编译通过了。

Build Output

```
linking...
Program Size: Code=3608 RO-data=336 RW-data=40 ZI-data=1632
FromELF: creating hex file...
".\out\interrupt.axf" - 0 Error(s), 0 Warning(s).
```

编译通过

这里有一点要注意，就是固件库手册在使用 **EXTI_ClearITPendingBit** 清除标志位的时候会报错。

是因为固件库数据手册大小写写错了

Table 141. 函数 EXTI_ClearITPendingBit

函数名	EXTI_ClearITPendingBit
函数原形	void EXTI_ClearITPendingBit(

例：

```
/* Clears the EXTI line 2 interrupt pending bit */
EXTI_ClearITPendingBit(EXTI_Line2);
```

EXTI_ClearITPendingBit 这个 P 要大写

中断线和 GPIO 映射报错

固件库用的是带下划线的参数，这是固件库手册描述错误

```
GPIO_EXTIConfig(GPIO_PortSource_GPIOB, GPIO_PinSource8); //映射中断线
GPIO_EXTIConfig(GPIO_PortSourceGPIOE, GPIO_PinSource2);
```

应该用不带下划线的参数，按照正点原子手册来

```
)
void EXTI15_10_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line10)!=RESET) //判断是不是这个中断线发生中断
    {
        EXTI_ClearFlag(EXTI_Line10); //清楚中断标志位
        EXTI_ClearITPendingBit(EXTI_Line10);
    }
}
```

这是中断服务函数

STM32 串口实现和 printf 实现

PA9 和 PA10 引脚是 USART1 串口

先在 C 文件实现串口中断优先级

```
void NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure; // 定义中断优先级结构体

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); // 中断组选择2组
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn; // 配置串口 USART1 位中断源
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // 抢占优先级为1
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1; // 响应优先级(子优先级)为1
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; // 中断优先级开启
    NVIC_Init(&NVIC_InitStructure); // 初始化中断优先级
}
```

再在 C 文件实现串口初始化

```
void USART_config(uint32_t baudrate)
{
    GPIO_InitTypeDef GPIO_InitStructure; // 设置 GPIO 为串口功能
    USART_InitTypeDef USART_InitStructure; // 设置串口参数

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // 打开 APB2 总线 GPIOA 的时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE); // 打开 APB2 总线 USART1 时钟

    // 将 USART1 TX 发送引脚对应的 GPIOA_9 管脚设置成推挽输出模式
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // 将 USART1 RX 接受引脚 GPIOA_10 设置成浮空输入模式
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    NVIC_Configuration(); // 串口中断优先级配置, 自己定义

    USART_InitStructureUSART_BaudRate = baudrate; // 设置串口波特率
    USART_InitStructureUSART_WordLength = USART_WordLength_8b; // 配置串口数据位为8位
    USART_InitStructureUSART_StopBits = USART_StopBits_1; // 配置串口停止位为1位
    USART_InitStructureUSART_Parity = USART_Parity_No; // 无校验位
    USART_InitStructureUSART_HardwareFlowControl = USART_HardwareFlowControl_None; // 无硬件流控
    USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx; // 发送接受都启动
    USART_Init(USART1, &USART_InitStructure); // 串口初始化配置

    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); // 开启串口接受中断
    USART_Cmd(USART1, ENABLE); // 使能串口1
}
```

```
void USART1_Send(uint8_t ch)
{
    USART_SendData(USART1, ch); // USART1 串口发送字节
    while (USART_GetFlagStatus(USART1, USART_FLAG_TC) == RESET); // 必须等待发送完成
}
```

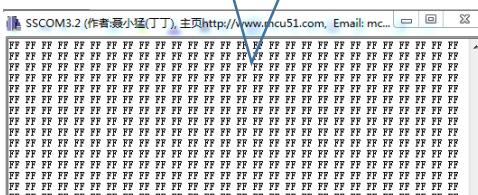
```
int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_config(115200); // 初始化串口

    while (1)
    {
        USART1_Send(0xff);
    }
    return 0;
}
```

在主函数里面测试下

编写串口发送函数试试串口是否设置成功

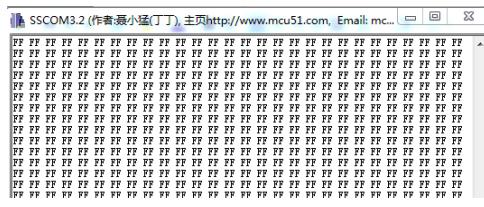
串口助手显示没有问题



```

void USART1_Send(uint8_t ch)
{
    // USART_SendData(USART1, ch); // USART1串口发送字节
    // while(USART_GetFlagStatus(USART1, USART_FLAG_TC) == RESET); // 必须等待发送完成
    while((USART1->SR&0X40)==0); // 循环发送,直到发送完毕
    USART1->DR = (uint8_t) ch;
}

```



也没有问题，我比较喜欢用寄存器。

测试串口中断接受功能

```

void USART1_IRQHandler(void)
{
    uint8_t ucTemp;
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) // 检查USART1是不是发生了中断
    {
        ucTemp = USART_ReceiveData(USART1); // 接受串口
        USART1_Send(ucTemp); // 把接受的数据返回给电脑
    }
}

```

串口中断函数名是 MCU 固定的名字，可以定义在任何地方

函数名	USART_GetITStatus
函数原形	ITStatus USART_GetITStatus(USART_TypeDef * USARTx)
功能描述	检查指定的 USART 中断发生与否

其实中断已经发生了，用这个函数来确认下

```

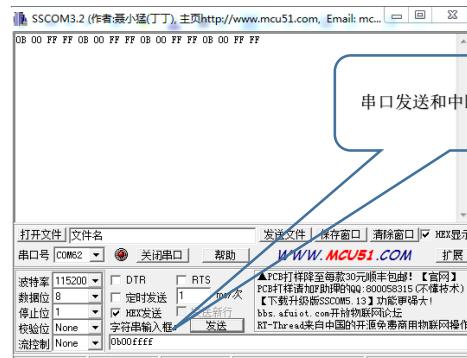
int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    while(1);

    return 0;
}

```

主函数实验下串口中断功能



串口 printf 实现

```

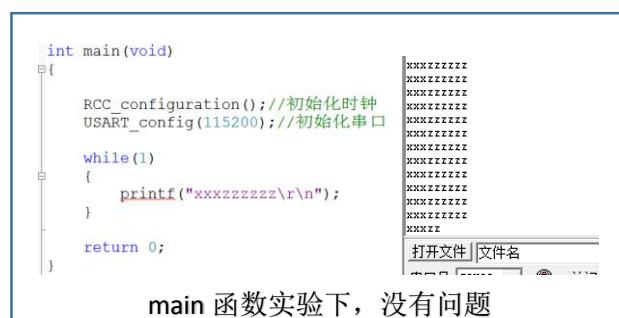
#include "stdio.h" //一定要加入stdio.h stm32里面自带的标准库
//加入以下代码,支持printf函数,而不需要选择use MicroLIB
#if 1
#pragma import(_use_no_semihosting)
//标准库需要的支持函数
struct __FILE
{
    int handle;
};

FILE __stdout;
//定义_sys_exit()以避免使用半主机模式
_sys_exit(int x)
{
    x = x;
}

//重定义fputc函数
int fputc(int ch, FILE *f)
{
    while((USART1->SR&0X40)==0); // 循环发送,直到发送完毕
    USART1->DR = (u8) ch;
    return ch;
}
#endif

```

在串口 C 文件里面加入 printf 的重定向代码，就可以使用 printf 了



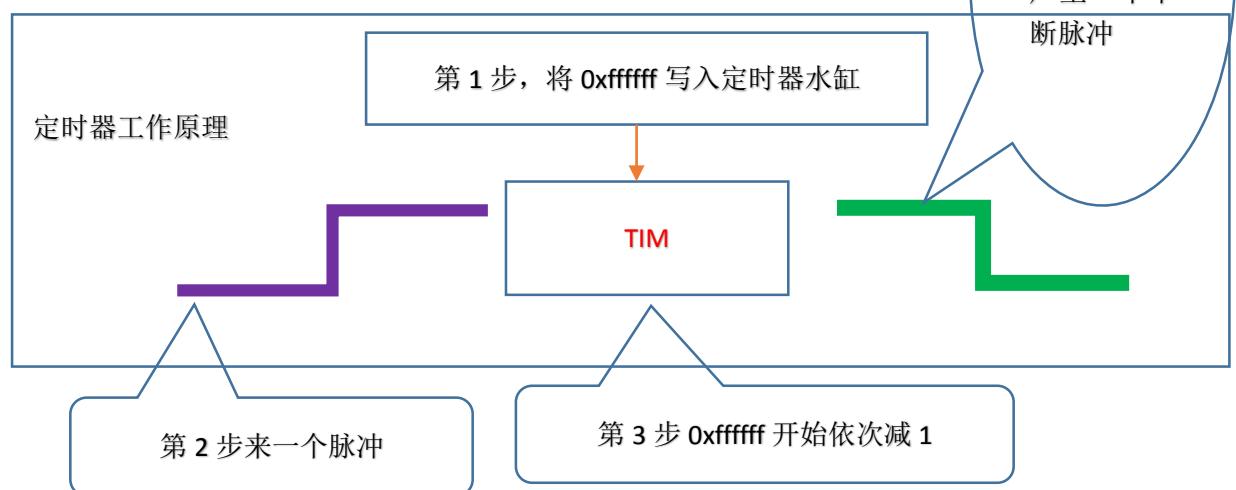
STM32 定时器

TIM1, TIM8 高级定时器

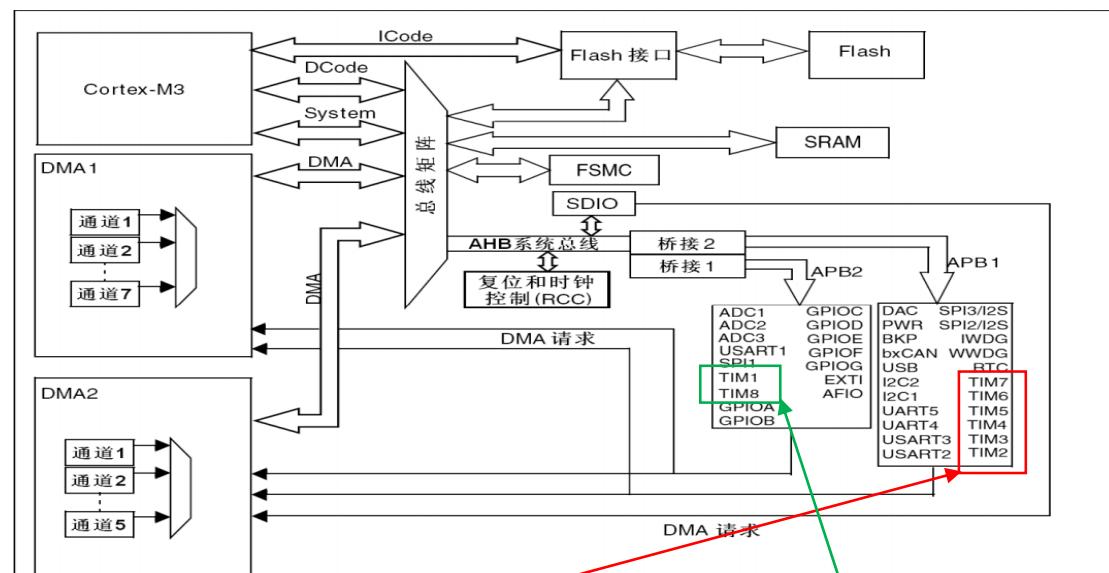
TIM2~TIM5 通用定时器

TIM6, TIM7 基本定时器

我们以定时器 TIM3 为例来做延时：

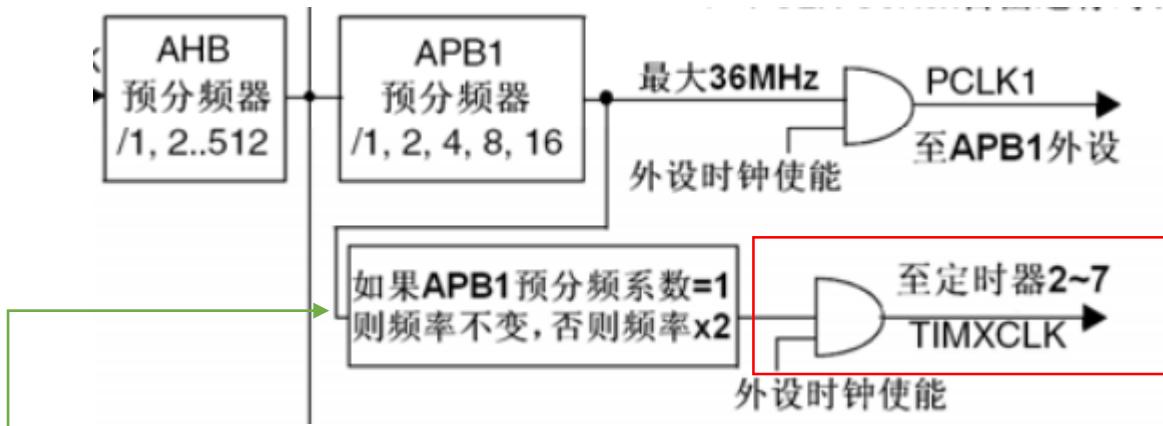


TIM 时钟来源：



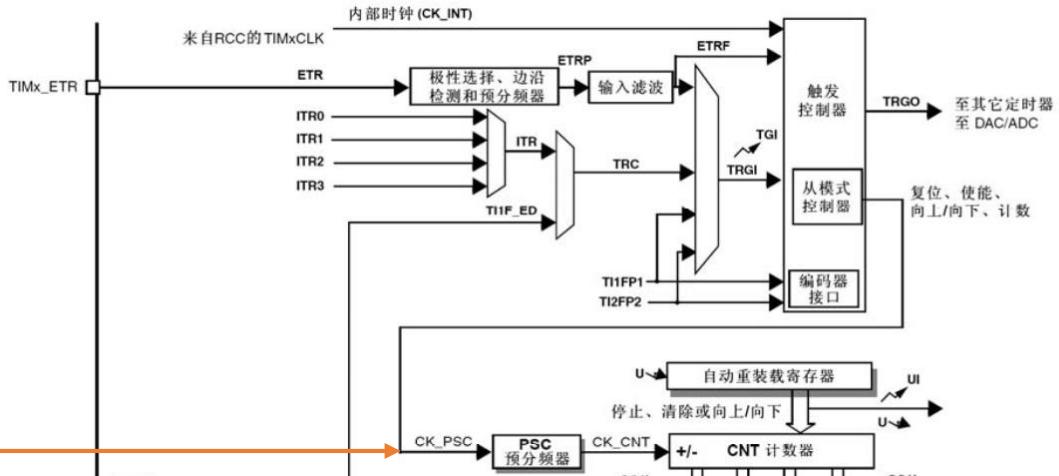
TIM 时钟来源于存储器总线架构的 APB1 总线。但是 TIM1, TIM8 来源于 APB2 总线
所以定时器时钟总线和前面的 GPIO 时钟总线来源是不一样的,GPIO 来源 APB2, TIM 来源 APB1

我们的 APB1 时钟最大不超过 36M，那么我们 TIM3 是 36M 的时钟吗？其实不是



看到了吗, APB1 其余外设是需要 36M 的时钟, 但是定时器的时钟是要 X2 倍频的, 所以定时器的时钟频率还是 72M。定时器在 APB1 总线上面玩特殊。APB1 X2 =72M

如果我们直接把 72M 加到 TIM3 定时器可以吗？那是不行的, 因为 TIM3 定时器只有 16 位, 72M 太快了, 所以达不到延时要求。



所以它要用个预分频器又来分频, 降低脉冲速度, 你说这是不是瞎折腾, 前面不倍频不就行了。

我们要设置 **PSC 预分频器** 来分频。

STM32 寄存器操作 TIM3 定时器

```

72 void Timer_init(u16 arr,u16 psc)
73 {
74     RCC->APB1ENR|=1<<1;
75     TIM3->ARR=arr;
76     TIM3->PSC=psc; //预分频器的值 计数器时钟频率=72M/(psc+1)
77     TIM3->DIER|=1<<0; //允许更新TIM3中断
78     TIM3->DIER|=1<<6; //允许触发TIM3中断
79     TIM3->CR1|=0x01; //使能定时器3
80
81     MY_NVIC_init(1,3,TIM3_IRQn,2); //抢占优先级1，响应优先级3，分到2组
82
83 }
84
85 void main()
86 {
87     //启动RCC时钟
88     Timer_init(10000,7199); //延时1秒产生定时器中断
89
90 }

```

划线这几个下面讲一下，其余的就不用讲了

RCC->APB1ENR|=1<<1; 打开 TIM3 定时器时钟

通常无访问等待周期。但在APB1总线上的外设被访问时，将插入等待状态直到APB1外设访问结束。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留	DACEN	PWR EN	BKP EN	CAN2 EN	CAN1 EN	保留	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART3 EN	USART2 EN	保留	保留	保留
		rw	rw	rw	rw				rw	rw	rw	rw			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	保留	WWDG EN		保留		TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN			
		rw	rw		rw				rw	rw	rw	rw			

位1

TIM3EN: 定时器3时钟使能 (Timer 3 clock enable)

由软件置'1'或清'0'

0: 定时器3时钟关闭;

1: 定时器3时钟开启。

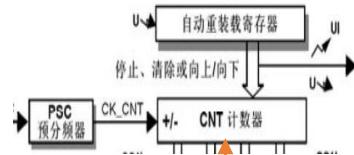
TIM3->ARR=arr;

14.4.12 自动重装载寄存器(TIMx_ARR)

偏移地址:0x2C

复位值:0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
位15:0 ARR[15:0]: 自动重装载的值 (Auto reload value)															



根据 CK_CNT 一个脉冲减掉 arr 里面一个位，减到 0 触发中断

至于 CK_CNT 多久触发一个脉冲呢？

TIM3->PSC=psc; //预分频器的值 计数器时钟频率=72M/(psc+1)

14.4.11 预分频器(TIMx_PSC)

偏移地址: 0x28

复位值: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
位15:0 PSC[15:0]: 预分频器的值 (Prescaler value)															



进入 psc 预分频器的时钟是 72M ,根据传入的参数:

```
void main()
{
    //启动RCC时钟
    Timer_init(10000, 7199); //延时1秒产生定时器中断
}
```

7199 是传入 psc 的参数，根据公式 $CK_CNT=CK_CNT=f_{CK_PSC}/(PSC[15:0]+1)$ 。
所以 $CK_CNT=72M/7200=10K$ 所以输入 CNT 计数器的时钟脉冲为 0.1ms 一次
根据延时要求 1 秒，那么 ARR 必须要 10000 个 0.1ms 的脉冲才能延时 1 秒。
如果你想 1000 个脉冲就能延时 1 秒，那么 CK_CNT 要 1ms 一个脉冲才行，可以修改 PSC。

```
void TIM3_IRQHandler(void)
{
    if (TIM3->SR & 0x0001) //溢出中断
    {
        //中断来了 执行想要执行的函数.....
    }

    TIM3->SR &= ~ (1<<0); //清除中断标志位
}
```

ARR 寄存器里面的值减完后，就进入 TIM3 中断函数程序。TIM3 定时器讲完了。

STM32 定时器 TIM3 固件库操作

```
void TIM_Configuration()
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //RCC->APB1ENR|=1<<1;//开启TIM3时钟

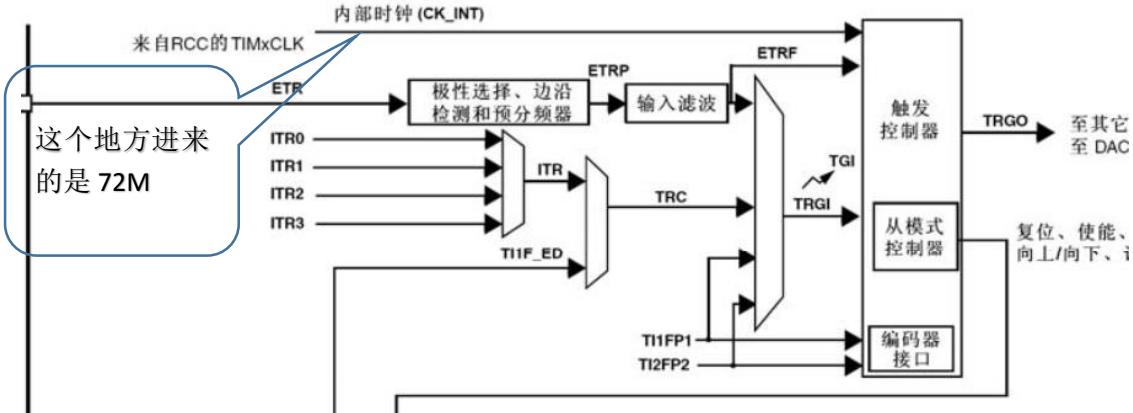
    TIM_DeInit(TIM3); //为了安全起见先调用这个配置一下TIM，也可以不调用，但是不安全
    TIM_InternalClockConfig(TIM3); //虽然TIM定时器默认是72M时钟，但是安全起见，还是配置一下。
    TIM_TimeBaseStructure.TIM_Period = 2000; //TIM3->ARR=arr;预装载值，当计数器达到这个值得时候产生中断
    TIM_TimeBaseStructure.TIM_Prescaler = 35999; // TIM3->PSC=psc;预分频器的值 计数器时钟频率=72M/(psc+1)
    TIM_TimeBaseStructure.TIM_ClockDivision = 0x0; //设置时钟分割
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计算CNT计数器来一个CK_INT就加一次
    TIM_TimeBaseInit(TIM3, & TIM_TimeBaseStructure);

    //中断优先级 NVIC 设置
    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn; //TIM3 中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //先占优先级 0 级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3; //从优先级 3 级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道被使能
    NVIC_Init(&NVIC_InitStructure); //④初始化 NVIC 寄存器

    TIM_Cmd(TIM3, ENABLE); //启动TIM3开始计时

    TIM_PrescalerConfig(TIM3, 35999, TIM_PSCReloadMode_Immediate); //立即装入预分频值前面都操作了我不知道这里为何还要操作
    TIM_ClearFlag(TIM3, TIM_FLAG_Update); //更新中断标志位
    TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE); //开启TIM3中断源
}
```

TIM_InternalClockConfig(TIM3); //虽然 TIM 定时器默认是 72M 时钟，但是安全起见，还是配置一下。
也就是选择 CK_INT 为 TIM 的时钟源

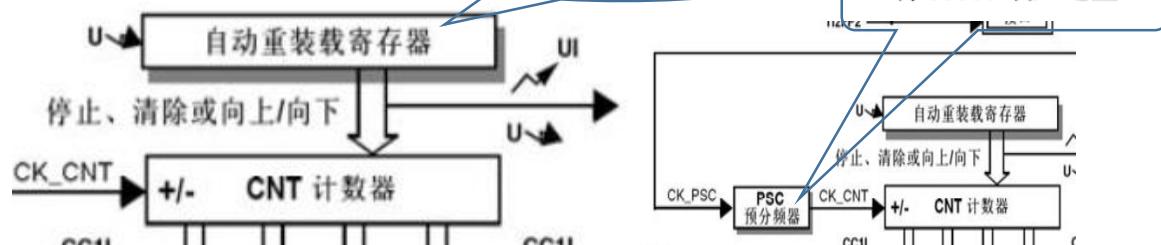


`TIM_TimeBaseStructure.TIM_Period = 2000; //TIM3->ARR=arr;预装载值, 当计数器达到这个值得时候产生中断`

`TIM_TimeBaseStructure.TIM_Period = 2000; //TIM3->ARR=arr;预装载值, 当计数器达到这个值得时候产生中断`

将 2000 装入这里

将 35999 装入这里



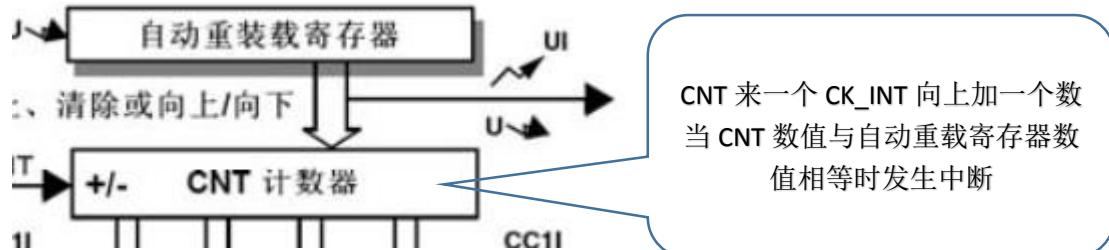
`TIM_TimeBaseStructure.TIM_Period = 2000; //TIM3->ARR=arr;预装载值, 当计数器达到这个值得时候产生中断`

`TIM_TimeBaseStructure.TIM_Period = 2000; //TIM3->ARR=arr;预装载值, 当计数器达到这个值得时候产生中断`

`TIM_TimeBaseStructure.TIM_Prescaler = 35999; // TIM3->PSC=psc;预分频器的值 计数器时钟频率=72M/(psc+1)`

`TIM_TimeBaseStructure.TIM_ClockDivision = 0x0; //设置时钟分割 我也没明白`

`TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计算 CNT 计数器来一个 CK_INT 就加一次`



其余的代码就很好理解了这里不多讲了

```

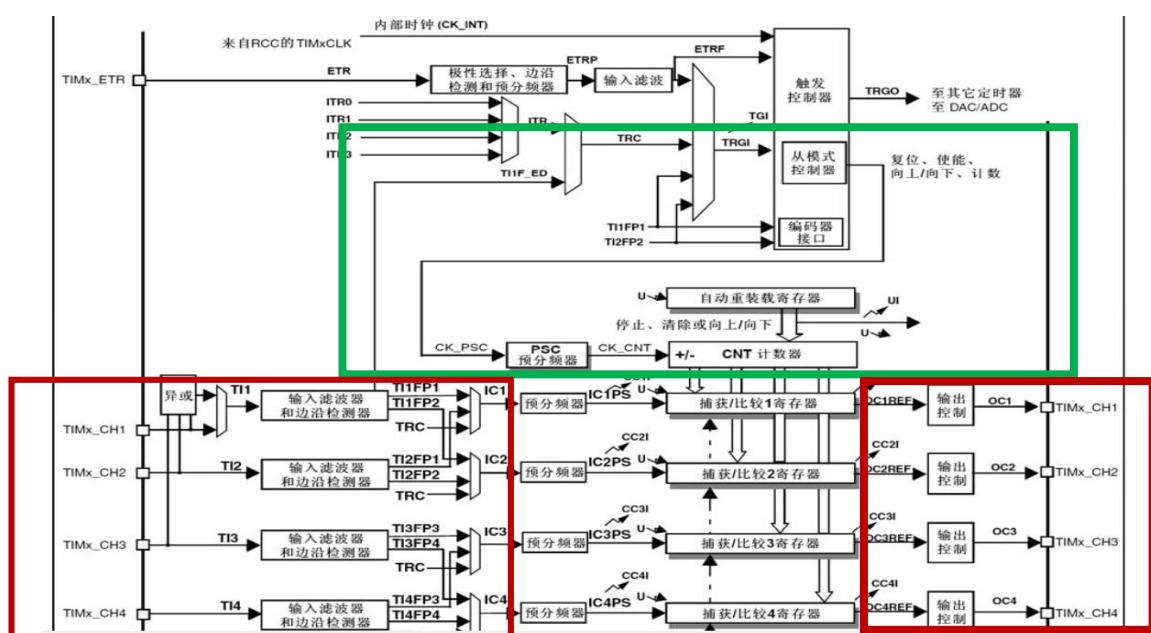
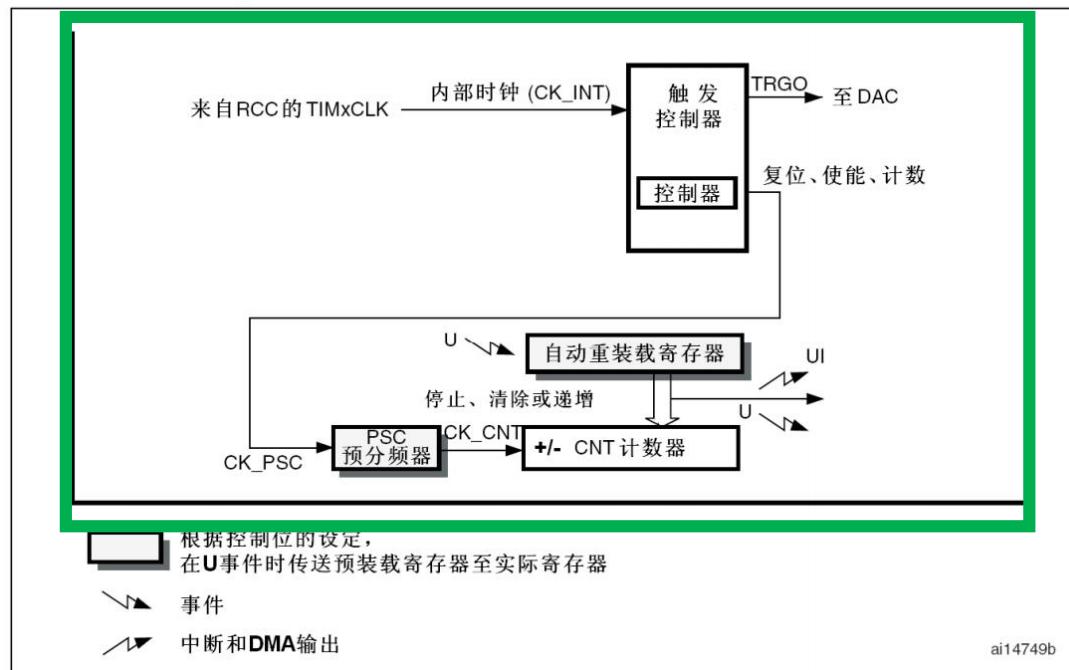
123 void TIM3_IRQHandler(void)
124 {
125     if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET) //检查 TIM3 更新中断发生与否
126     {
127         TIM_ClearITPendingBit(TIM3, TIM_IT_Update); //清除 TIM3 更新中断标志
128         //执行中断函数里面的代码.....
129     }
130 }
131
132
133
134
135 void main()
136 {
137     //启动RCC时钟
138     TIM_Configuration; //延时1秒产生定时器中断
139
140 }
```

这就是通用定时器中断服务程序。

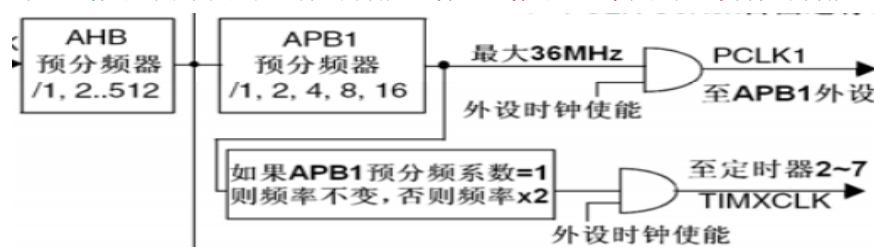
STM32 基本定时器 TIM6, TIM7 操作

基本定时器不像 TIM2~TIM5 那样有输入捕捉和输出比较功能，因为 STM32 芯片没有给 TIM6 和 TIM7 配置外部引脚，所以基本定时器只有**定时功能和 DAC 功能**。

图144 基本定时器框图

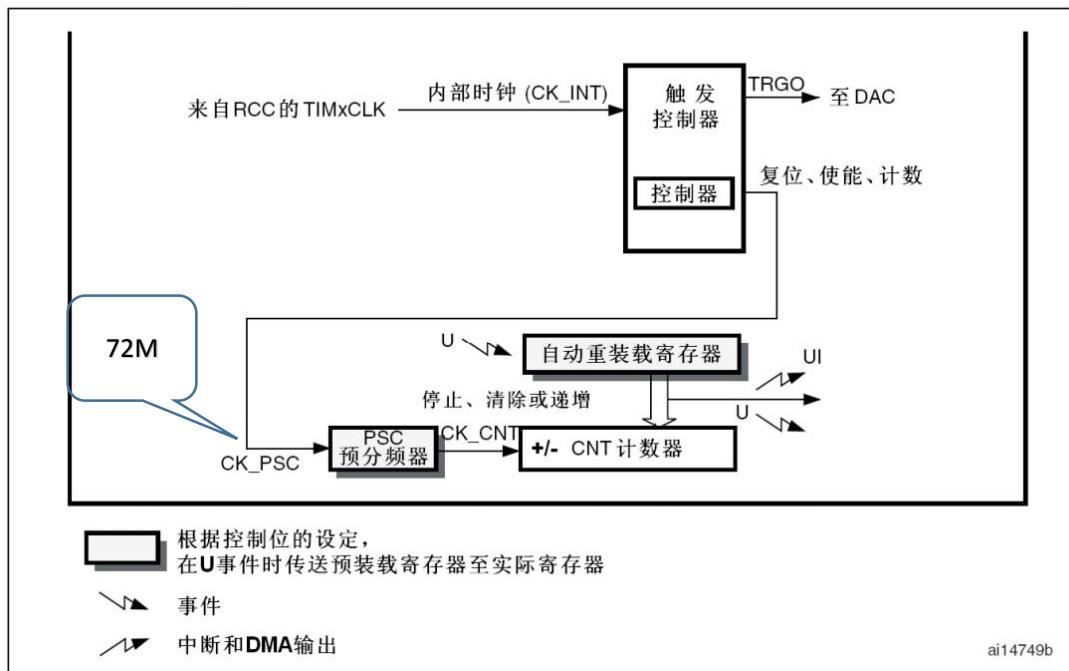


绿色线框是两个定时器有的功能，棕色线框是基本定时器没有的功能，只有通用定时器有。



和通用定时器一样时钟经过 APB1 预分频之后再 X2 倍频

图144 基本定时器框图



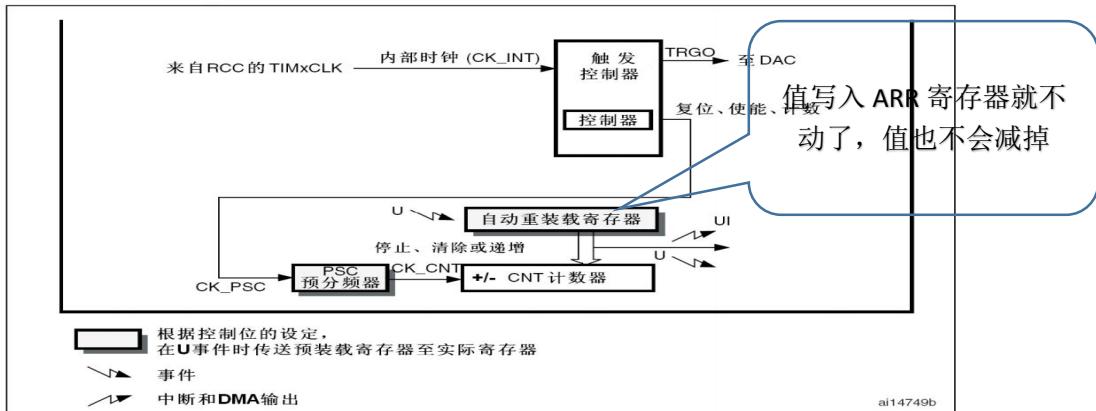
这个自动重装载寄存器不像 51 单片机那样，定时时间到产生中断。然后还要用软件去写两行代码让定时器值重新装进寄存器。

```
void T0_ISR() interrupt 1
{
    实现用户功能目标代码
    /*恢复计数器初值*/
    TH0=(65536-50000)/256; //50ms 定时, 计数器的初值高 8 位
    TL0=(65536-50000)%256; //50ms 定时, 计数器的初值低 8 位
}
```

这是 51 单片机的操作，很麻烦。

但是 STM32 就不一样

图144 基本定时器框图



CNT 计数器根据 CK_CNT 的脉冲一次一次向上加，当值加到与 ARR 值相等了后，比较强输出一个信号产生中断。所以我们不用每次在中断函数里面去写两行代码来加载重载寄存器。

我们以 TIM7 为例来操作基本定时器：

```
void TIM7_Configuration (void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure; //定时器7结构体
    NVIC_InitTypeDef NVIC_InitStructure; //中断优先级结构体

    TIM_TimeBaseStructure.TIM_Prescaler = 36000 - 1; //预分频系数为36000-1, 这样计数器时钟为72MHz/36000 = 2kHz
    TIM_TimeBaseStructure.TIM_Period = 2000; //TIM3->ARR=arr;预装载值, 当计数器达到这个值得时候产生中断
    TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割 TIM_CKD_DIV1=0x0000,不分割
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计算CNT计数器来一个CK_INT就加一次
    TIM_TimeBaseInit (TIM7, &TIM_TimeBaseStructure);
    TIM_UpdateRequestConfig(TIM7, TIM_UpdateSource-Regular); //生成单一的脉冲：计数器在下一个更新事件停止
    TIM_Cmd (TIM7, ENABLE);
    TIM_ITConfig (TIM7, TIM_IT_Update,ENABLE); //使能定时器7中断
    TIM_ClearFlag (TIM7, TIM_FLAG_Update); //清除TIM7中断标志位

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); //设置分组
    NVIC_InitStructure.NVIC_IRQChannel =TIM7_IRQn; //设置 TIM7线
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority =0; //设置抢占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority =0; //设置响应优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道被使能
    NVIC_Init(&NVIC_InitStructure);

}

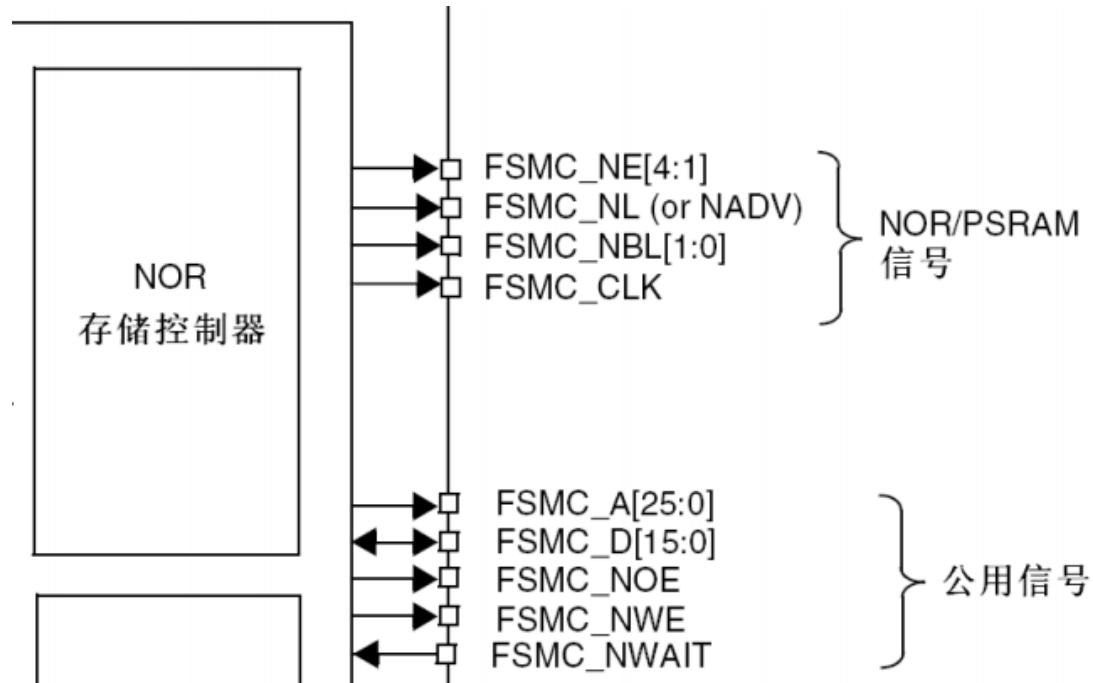
void TIM7_IRQHandler (void)
{
    if (TIM_GetITStatus (TIM7, TIM_IT_Update) != RESET) //检查 TIM3 更新中断发生与否
    {
        TIM_ClearITPendingBit (TIM7, TIM_IT_Update ); //清除 TIM3 更新中断标志
        //执行中断函数里面的代码.....
    }
}

void main()
{
    //启动RCC时钟
    TIM7_Configuration;//延时1秒产生定时器中断
}
```

基本定时器也和通用定时器差不多只是没有通用定时器那么多功能，基本定时器只能定时和 DAC。

STM32 通用定时器输入捕获功能

FSMC 驱动 LCD



我们只用框图的上面这部分。

FSMC存储块

地址	存储块	支持的存储器类型
6000 0000h 6FFF FFFFh	块 1 4 × 64 MB	NOR / PSRAM
7000 0000h 7FFF FFFFh	块 2 4 × 64 MB	NAND 闪存
8000 0000h 8FFF FFFFh	块 3 4 × 64 MB	
9000 0000h 9FFF FFFFh	块 4 4 × 64 MB	PC 卡

每个块 256M，每个块分 4 个区，每个区是 64M，每个区都有独立的寄存器对所连接的存储器进行配置。每个块起始地址不一样，但是大小都一样是 256M

HADDR[27:26]用于选择四个存储块之一：

表86 NOR/PSRAM存储块选择

HADDR[27:26] ⁽¹⁾	选择的存储块
00	存储块1 NOR/PSRAM 1
01	存储块1 NOR/PSRAM 2
10	存储块1 NOR/PSRAM 3
11	存储块1 NOR/PSRAM 4

(1) HADDR是需要转换到外部存储器的内部AHB地址线。

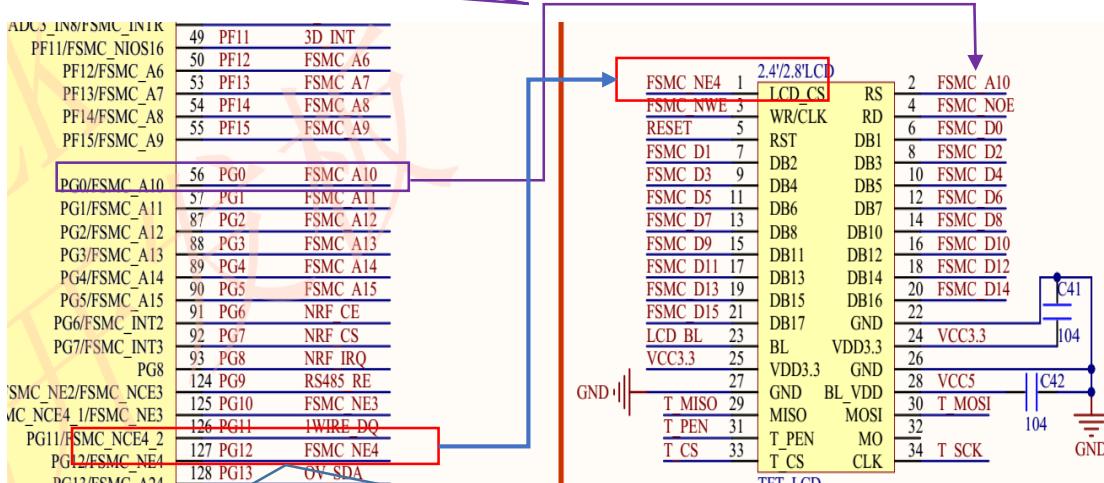
HADDR[25:0]包含外部存储器地址。HADDR是字节地址，而存储器访问不都是按字节访问，因此接到存储器的地址线依存储器的数据宽度有所不同，如下表：

表87 外部存储器地址

数据宽度 ⁽¹⁾	连到存储器的地址线	最大访问存储器空间(位)
8位	HADDR[25:0]与FSMC_A[25:0]对应相连	64M字节 × 8 = 512 M位
16位	HADDR[25:1]与FSMC_A[24:0]对应相连，HADDR[0]未接	64M字节/2 × 16 = 512 M位

(1) 对于16位宽度的外部存储器，FSMC将在内部使用HADDR[25:1]产生外部存储器的地址FSMC_A[24:0]。不论外部存储器的宽度是多少(16位或8位)，FSMC_A[0]始终应该连到外部存储器的地址线A[0]。

A10 信号0发送命令信号1发送数据

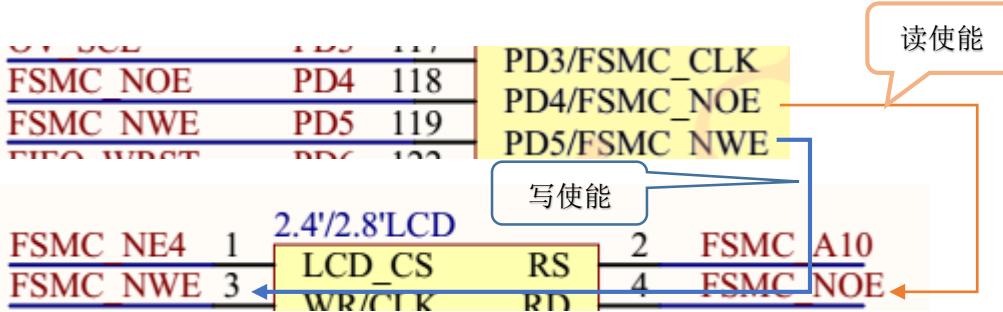


前面说了1个bank有4个片选，我这里的NE4接的就是第一个bank的第4段内存64M

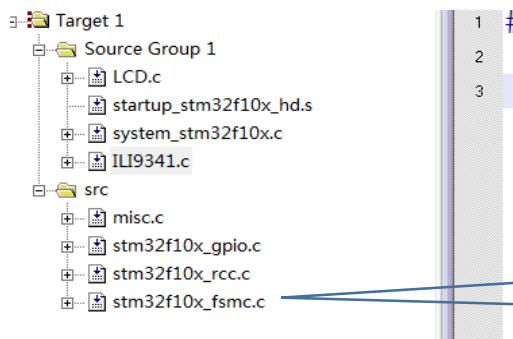
为什么接在第一个bank上面呢？



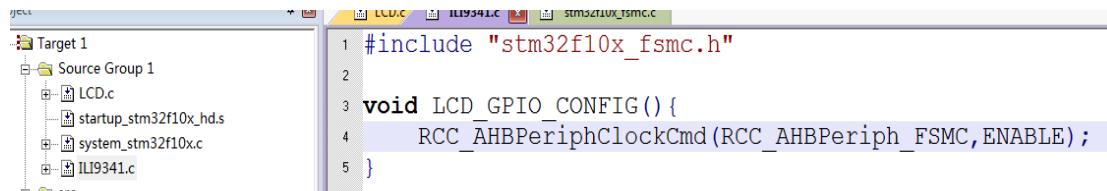
因为液晶的GRAM就同等于NOR Flash或者PSRAM，其余的NAND flash或者PC卡都不属于NOR Flash协议



现在我们来写 FSMC 驱动 LCD 的代码



配置 FSMC 基本参数



在配置 FSMC 时钟的时候发现老版本的固件库没有 FSMC 配置说明



此份固件库用户手册的整体架构如下：

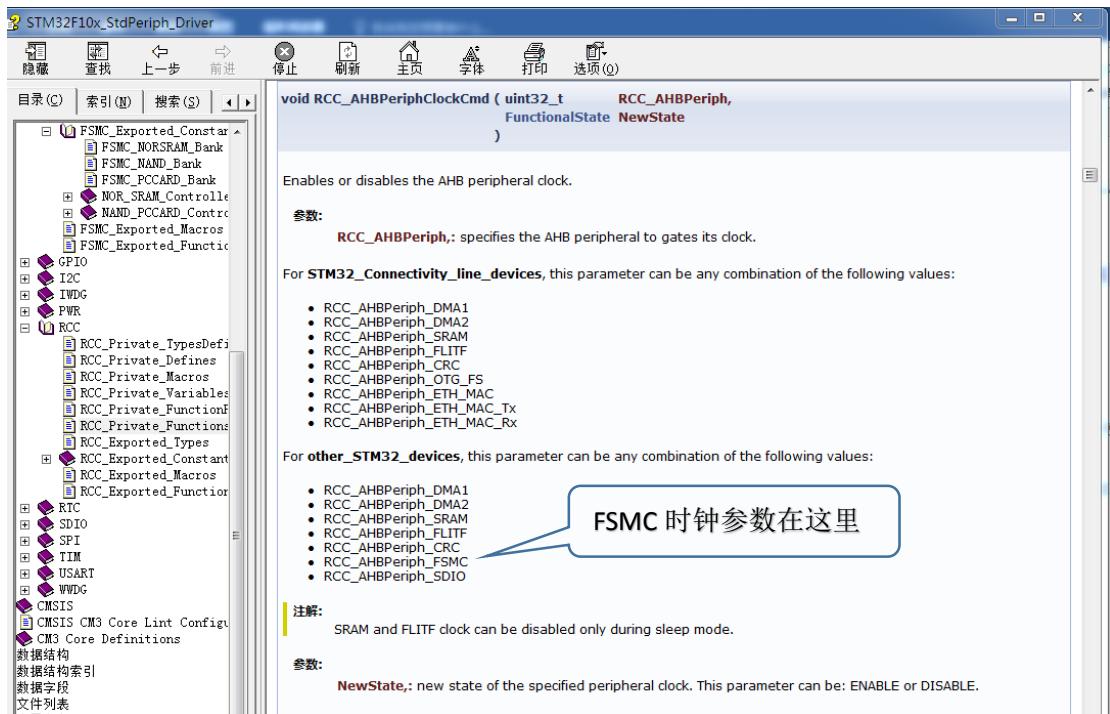
- 定义，文档约定和固态函数规则。
- 固态函数库概述（包的内容，库的架构），安装指南，库使用实
- 固件库具体描述：设置架构和每个外设的函数。

STM32F101xx 和 STM32F103xx 在整个文档中被写作 STM32F101x



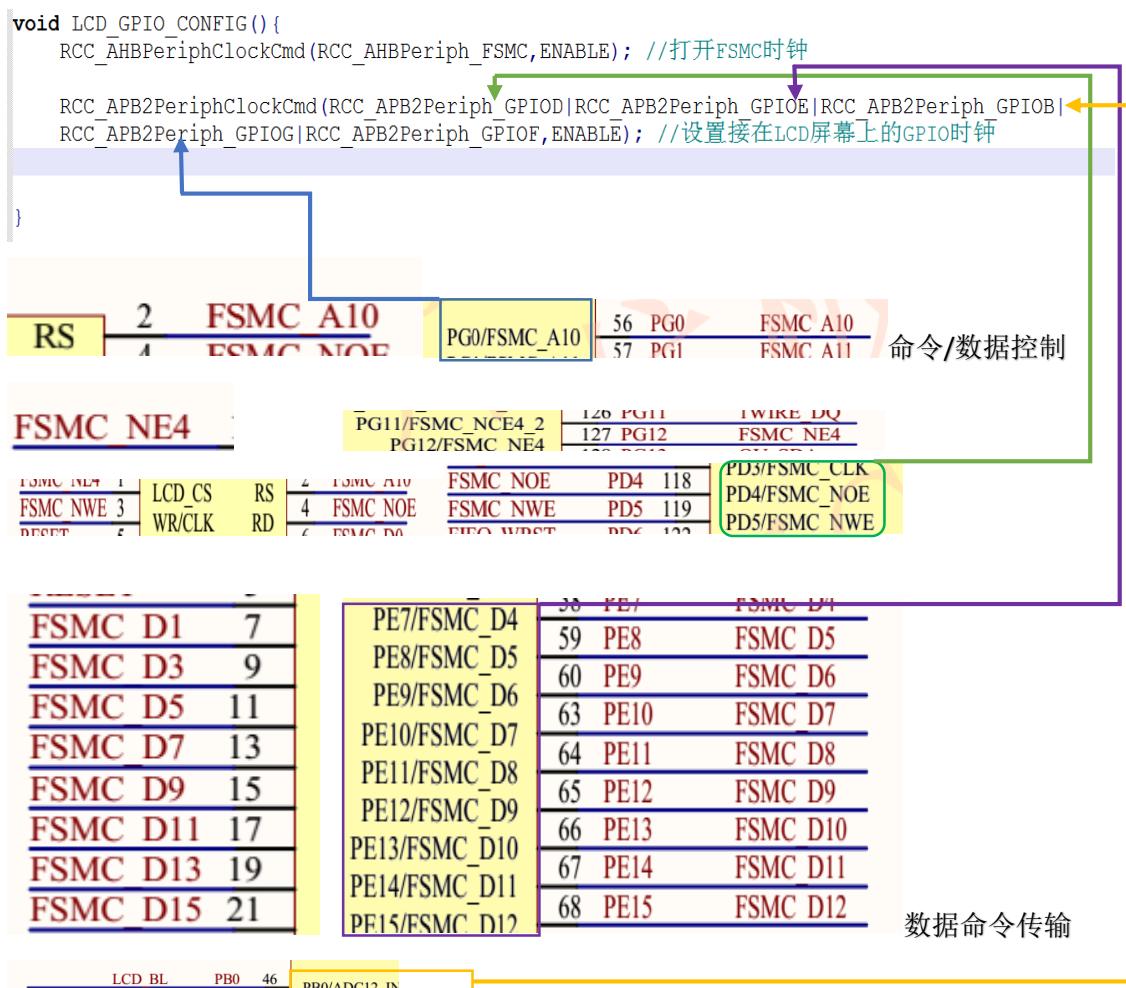
译文英文原版为 UM0427 Oct. 2007 Rev 2，译文仅供参考，与英文版冲突的，

所以我换了一个查找库函数的方式



在这个文档里面发现了 FSMC 配置参数

设置接在 LCD 屏的 GPIO



所以这些 IO 口的时钟都要打开

设置接入 LCD 的 GPIO

```
RCC_APB2Periph_GPIOG|RCC_APB2Periph_GPIOF,ENABLE); //设置接在LCD屏幕上的GPIO时钟

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; //LCD背光
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOB, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11; //LCD复位
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOG, &GPIO_InitStructure);

//////////////////////////////设置LCD数据线/////////////////////
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|
                               GPIO_Pin_14|GPIO_Pin_15;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //GPIO和FSMC复用输出
GPIO_Init(GPIOD, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_11|
                               GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //GPIO和FSMC复用输出
GPIO_Init(GPIOE, &GPIO_InitStructure);

//////////////////////////////设置LCD控制线////////////////////
/*FSMC_NOE <-> LCD_RD
 *FSMC_NWE <-> LCD_WR
 *FSMC_NE4 <-> LCD_CS
 *FSMC_A10 <-> LCD_DC
 */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4; //设置NOE
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //GPIO和FSMC复用输出
GPIO_Init(GPIOD, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //设置NWE
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //GPIO和FSMC复用输出
GPIO_Init(GPIOD, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12; //设置NE4
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //GPIO和FSMC复用输出
GPIO_Init(GPIOG, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; //设置A10
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //GPIO和FSMC复用输出
GPIO_Init(GPIOG, &GPIO_InitStructure);
```

打开 LCD 背光

—

```
GPIO_SetBits(GPIOB, GPIO_Pin_10); //打开背光
```

现在我们来设置 FSMC 参数

HADDR[27:26]用于选择四个存储块之一：

表86 NOR/PSRAM存储块选择

HADDR[27:26] ⁽¹⁾	选择的存储块
00	存储块1 NOR/PSRAM 1
01	存储块1 NOR/PSRAM 2
10	存储块1 NOR/PSRAM 3
11	存储块1 NOR/PSRAM 4

(1) HADDR是需要转换到外部存储器的内部AHB地址线。

HADDR[25:0]包含外部存储器地址。HADDR是字节地址，而存储器访问不都是按字节访问，因此接到存储器的地址线依存储器的数据宽度有所不同，如下表：

表87 外部存储器地址

数据宽度 ⁽¹⁾	连到存储器的地址线	最大访问存储器空间(位)
8位	HADDR[25:0]与FSMC_A[25:0]对应相连	64M字节 × 8 = 512 M位
16位	HADDR[25:1]与FSMC_A[24:0]对应相连，HADDR[0]未接	64M字节/2 × 16 = 512 M位

(1) 对于16位宽度的外部存储器，FSMC将在内部使用HADDR[25:1]产生外部存储器的地址FSMC_A[24:0]。不论外部存储器的宽度是多少(16位或8位)，FSMC_A[0]始终应该连到外部存储器的地址线A[0]。

地址 存储块 支持的存储器类型



我们来看看 HADDR 怎么用，我们假如用 `FSMC_A0` 来控制 LCD 的写命令/数据

PF0/FSMC_A0	10 PF0	FSMC A0
PF1/FSMC_A1	11 PF1	FSMC A1
PF2/FSMC_A2	12 PF2	FSMC A2
PF3/FSMC_A3	13 PF3	FSMC A3
PF4/FSMC_A4	14 PF4	FSMC A4
PF5/FSMC_A5	15 PF5	FSMC A5

如果是 `FSMC_A0=HADDR_A0`

我们写 0001 = 0001

但是我们知道 STM32 规定访问 16 外部存储器的时候，`FSMC_A0=HADDR_A1`

所以要控制 `FSMC_A0` 就必须是 `FSMC_A0=HADDR_A1`

0001 = 0010

这里 HADDR 是内部 AHB 地址总线，其中 HADDR[25:0]来自外部存储器地址 `FSMC_A[25:0]`，而 HADDR[26:27]对 4 个区进行寻址。如表 18.1.2.1 所示：

Bank1 所选区	片选信号	地址范围	HADDR	
			[27:26]	[25:0]
第 1 区	FSMC_NE1	0X6000, 0000~63FF, FFFF	00	FSMC_A[25:0]
第 2 区	FSMC_NE2	0X6400, 0000~67FF, FFFF	01	
第 3 区	FSMC_NE3	0X6800, 0000~6BFF, FFFF	10	
第 4 区	FSMC_NE4	0X6C00, 0000~6FFF, FFFF	11	

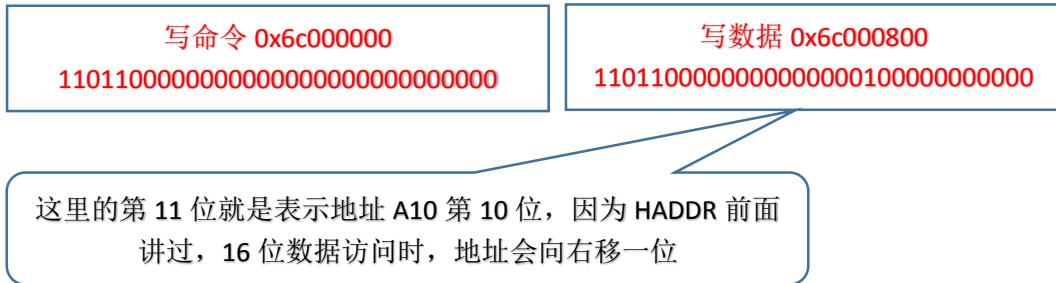
```

03
04 #define LCD_addr_reg ((u32)(0x6c000000)) //因为是A10, 如果是A15 ->0X6C000000
05 #define LCD_Data_reg ((u32)(0x6c000800)) //因为是A10, 如果是A15 ->0X6C010000
06
07 #define LCD_WR_REG(regs) (* (u16 *) (LCD_addr_reg)) = ((u16)regs)
08 #define LCD_WR_DATA(datas) (* (u16 *) (LCD_Data_reg)) = ((u16)datas)
09
10 void LCD_Write_command(unsigned int reg)
11 {
12     LCD_WR_REG(reg);
13 }
14
15 void LCD_Write_data(unsigned int data)
16 {
17     LCD_WR_DATA(data);
18 }

```

因为我们用的是 bank1 的 NE4 分区，所以引脚 A0-A16 起始地址是 0X6C00 0000

因为我们要单独控制 A10，所以给 LCD 写命令是访问 0X6C000000，写数据 0x6C000800



LCD_WR_REG (reg) 把要写的命令 reg 放入这个地址

LCD_WR_DATA (data) 把要写的数据 data 放入这个地址

也可以像正点原子这样

本实验，我们用到 FSMC 驱动 LCD，通过前面的介绍，我们知道 TFTLCD 的 RS 接在 FSMC 的 A10 上面，CS 接在 FSMC_NE4 上，并且是 16 位数据总线。即我们使用的是 FSMC 存储器 1 的第 4 区，我们定义如下 LCD 操作结构体（在 lcd.h 里面定义）：

```

//LCD 操作结构体
typedef struct
{
    u16 LCD_REG;
    u16 LCD_RAM;
} LCD_TypeDef;

//使用 NOR/SRAM 的 Bank1.sector4,地址位 HADDR[27,26]=11 A10 作为数据命令区分线
//注意 16 位数据总线时，STM32 内部地址会右移一位对齐！
#define LCD_BASE ((u32)(0x6C000000 | 0x0000007FE))
#define LCD ((LCD_TypeDef *) LCD_BASE)

```

其中 LCD_BASE，必须根据我们外部电路的连接来确定，我们使用 Bank1.sector4 就是从地址 0X6C000000 开始，而 0X0000007FE，则是 A10 的偏移量。我们将这个地址强制转换为

LCD_TypeDef 结构体地址，那么可以得到 LCD->LCD_REG 的地址就是 0X6C00,07FE，对应 A10 的状态为 0(即 RS=0)，而 LCD->LCD_RAM 的地址就是 0X6C00,0800（结构体地址自增），对应 A10 的状态为 1(即 RS=1)。

所以，有了这个定义，当我们要往 LCD 写命令/数据的时候，可以这样写：

```

LCD->LCD_REG=CMD; //写命令
LCD->LCD_RAM=DATA; //写数据

```

而读的时候反过来操作就可以了，如下所示：

```

CMD=LCD->LCD_REG; //读 LCD 寄存器
DATA = LCD->LCD_RAM; //读 LCD 数据

```

用上面设置的函数来操作液晶屏幕

1. 初始化液晶屏

```
void LCD_REG_CONFIG() //LCD寄存器初始化函数
{
    LCD_Write_command(0xcf);
    LCD_Write_data(0x00);
    LCD_Write_data(0x81);
    LCD_Write_data(0x30);
    /*
    * .....写很多命令...
    * .....写很多数据...
    */
}
```

2. 给液晶屏进行清屏，显示自己喜欢的颜色

```
void LCD_clear(u16 x,u16 y,u16 width,u16 height,u16 color)//液晶清屏
{
    u32 i=0;
    /*column address set*/
    LCD_Write_command(0x2A);
    LCD_Write_data(x>>8); /*先传入高8位*/
    LCD_Write_data(x&0xff); /*然后传入低8位 column start*/
    LCD_Write_data((x+width-1)>>8); /*column end*/
    LCD_Write_data((x+width-1)&0xff);/*和上面是一样的*/

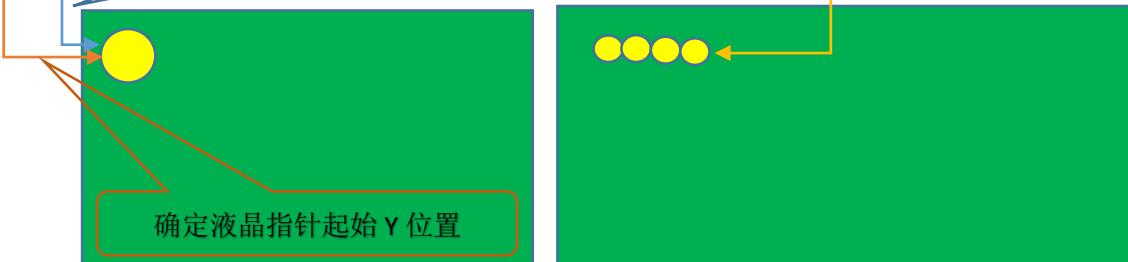
    /*page address set*/
    LCD_Write_command(0x2B);
    LCD_Write_data(y>>8); /*先传入高8位*/
    LCD_Write_data(y&0xff); /*然后传入低8位 column start*/
    LCD_Write_data((y+height-1)>>8); /*column end*/
    LCD_Write_data((y+height-1)&0xff);/*和上面是一样的*/
    /*屏幕上的起始坐标已经确定了*/

    LCD_Write_command(0x2C);/*写清除内存的命令，也就是按照上面设置的坐标开始。
    for(i=0;i<width+height;i++)//从上面设置的坐标开始，指针自动增加换行，接着屏幕点清除，清除到width+height
    {
        //位置为止
        LCD_Write_data(color);/*确定清除后的颜色*/
    }
}
```

确定液晶指针起始 X 位置

开始清屏

确定液晶指针起始 Y 位置



4.先学会坐标定位

```
void LCD_SetCursor(u16 x, u16 y)
{
    LCD_Write_command(0x2A); //设置X坐标
    LCD_Write_data(x>>8);
    LCD_Write_data(x&0xff);

    LCD_Write_command(0x2B); //设置y坐标
    LCD_Write_data(y>>8);
    LCD_Write_data(y&0xff);

}
```

5.再学会给该坐标点画颜色

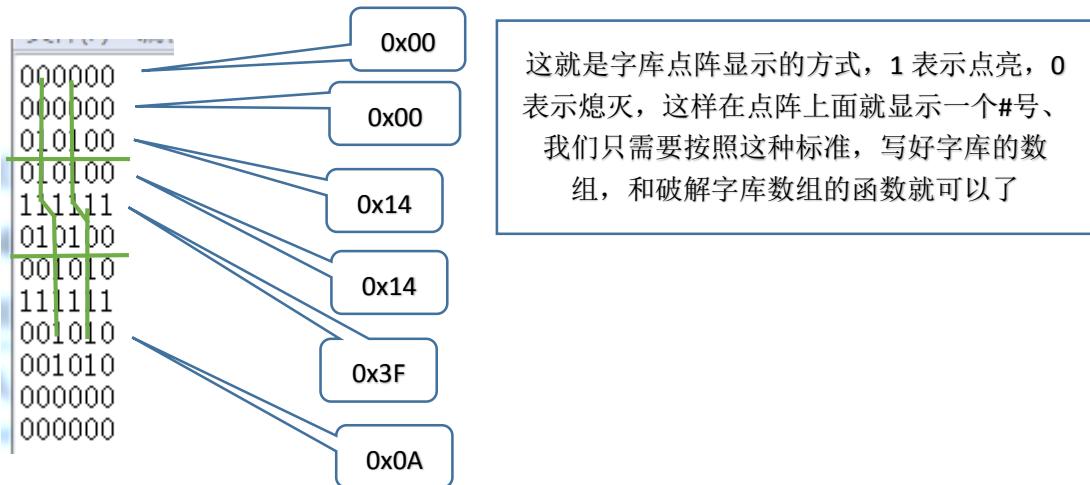
```
//画点
//x,y:坐标
//POINT_COLOR:此点的颜色
void LCD_DrawPoint(u16 x,u16 y,u16 POINT_COLOR)
{
    LCD_SetCursor(x,y); //设置光标位置
    LCD_Write_command(0x55); //开始写入 GRAM
    LCD_Write_data(POINT_COLOR);
}
```

现在我们来了解英文字符显示原理

比如我要写#号，这个#号正好是 12*6 大小。

```
#define STR_WIDTH          6      /* 字符宽度 */
#define STR_HEIGHT         12      /* 字符高度 */

/*
 * 常用 ASCII 表，偏移量 32，大小:12（高度）*6（宽度）
 */
const unsigned char asc2_1206[95][12]={
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*" ",0*/
{0x00,0x00,0x04,0x04,0x04,0x04,0x04,0x04,0x04,0x04,0x00,0x00},/*"!",1*/
{0x00,0x14,0x0A,0x0A,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*""",2*/
{0x00,0x00,0x14,0x14,0x3F,0x14,0x0A,0x3F,0x0A,0x0A,0x00,0x00},/*"#",3*/
}
```



下面展示一段 VC++6.0 实现的点阵代码。

```

char i = 90; /* } */

int main()
{
    char temp = 0, page = 0, column = 0;

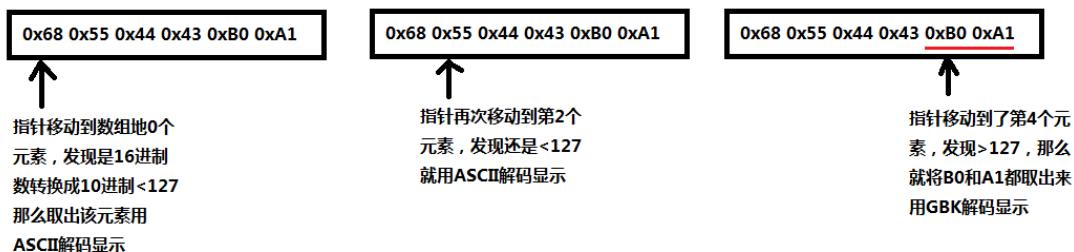
    for( page=0; page < STR_HEIGHT; page++ )
    {
        temp = asc2_1206[i][page];

        for( column=0; column < STR_WIDTH; column++ )
        {
            if( temp & 0x01 )
            {
                printf( " * ");
            }
            else
            {
                printf( " . ");
            }
            temp >>= 1;
        }
        printf( "\n" ); /* 写完一行 */
    }
    printf( "\n" ); /* 全部写完 */
}

```

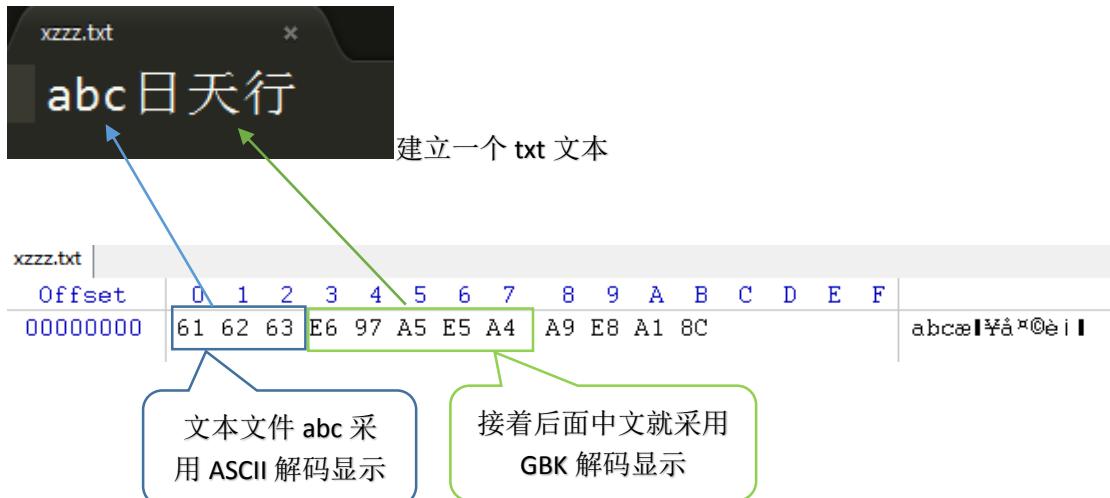
STM32 字库实现直接复制粘贴正点原子的代码就是了。

LCD 字符解码方式



0x68 0x55 0x44 0x43 0xB0 0xA1 0x22 0xFB 0x23 0x01

指针再次移动发现是 <127 , ASCII解码 指针移到这里发现第1个元素是>127但是第2个元素<127，这种情况下，也会将第1个元素和第二个元素合并起来用GBK解码



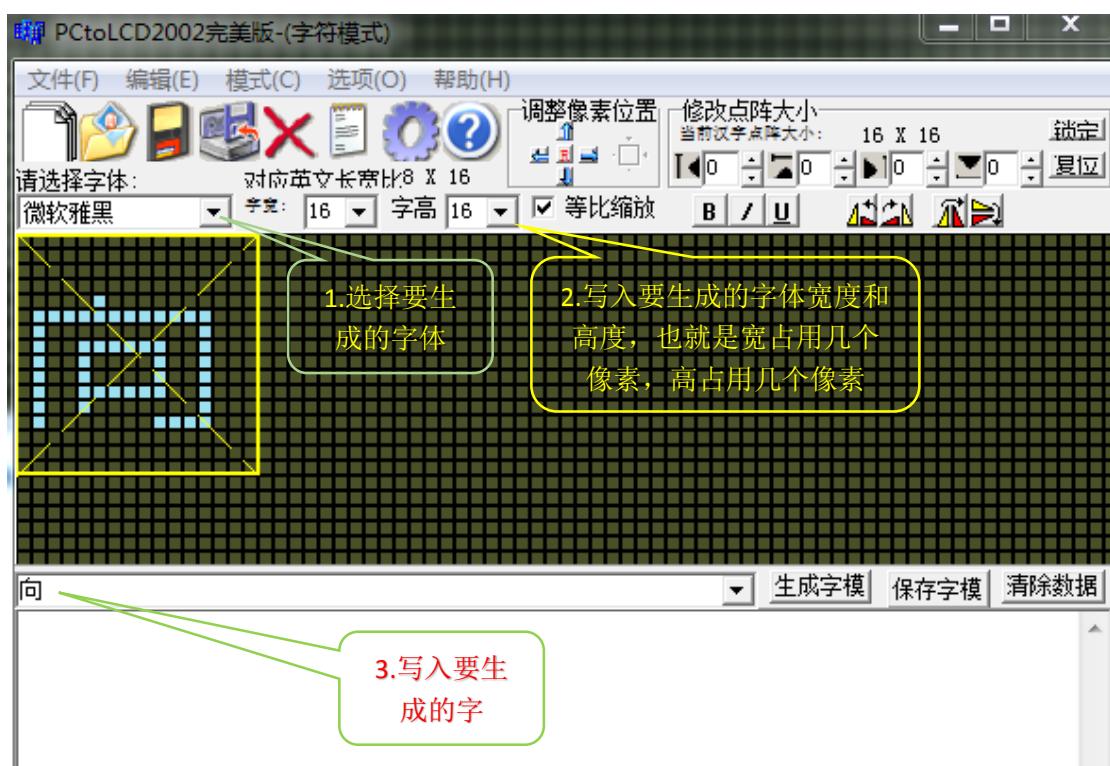
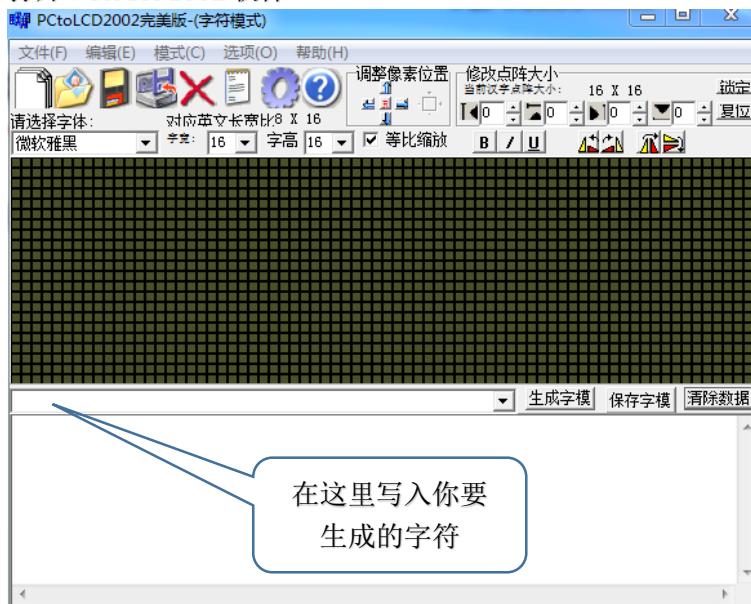
GBK 编码和 GB2312 都是要求浏览器必须支持这种编码，才能显示，国内浏览器没有这个问题。但是国外的一些浏览器可能不支持，这个时候就要用一种万国编码，叫做 utf-8，这种编码支持多个国家的语言，编码方式和上面介绍的原理差不多。

字符	GB18030 编码	Unicode 编号	UTF-16 编码
A	0x41	0x0000 0041	大端格式 0x0041
啊	0xB0A1	0x0000 554A	大端格式 0x554A
嶧	0x9735 F832	0x0002 75CC	大端格式 0xD85D DDCC

你看三种编码不同的数字表达方式，utf-8 编码就是 Unicode 编码的一种，但是这种编码就是要占用 4 个字节。嵌入式设备内存小要注意，PC 端内存大无所谓。

字模生成软件使用

打开 PCtoLCD2002 软件



```

uint_8 char_model[] = {{0x00,0x00,0x07,0xF8,0x04,0x00,0x04,0x00,0x05,0xF0,0x0D,0x20,0x05,0x20,0x05,0x20},
{0x05,0x20,0x05,0xE8,0x04,0x08,0x04,0x08,0x07,0xF8,0x00,0x00,0x00,0x00,0x00},/*"向",0*/
};

void Dispaly_char(void)
{
    uint_8 row,byte,bit;

    for(row = 0;row < 16;row++)
    {
        for(byte = 0;byte < 2;byte++)
        {
            for(bit = 0; bit < 8; bit++)
            {
                if (char_model[row*2+byte] & (0x80 & bit))
                {
                    //点亮该像素点
                }
                else
                {
                    //点黑该像素点
                }
            }
        }
    }
}

```

这是生成'向'字体的二维数组，但是我们 Dispaly_char 函数是按照一维数组来计算的

而且这个程序的计算方法必须和取模软件的扫描方式匹配，这里取模和数组都是逐列式取的

我们 PCtoLCD2002 生成字模选择的显示方式是阴码，逐列式，取模走向顺向

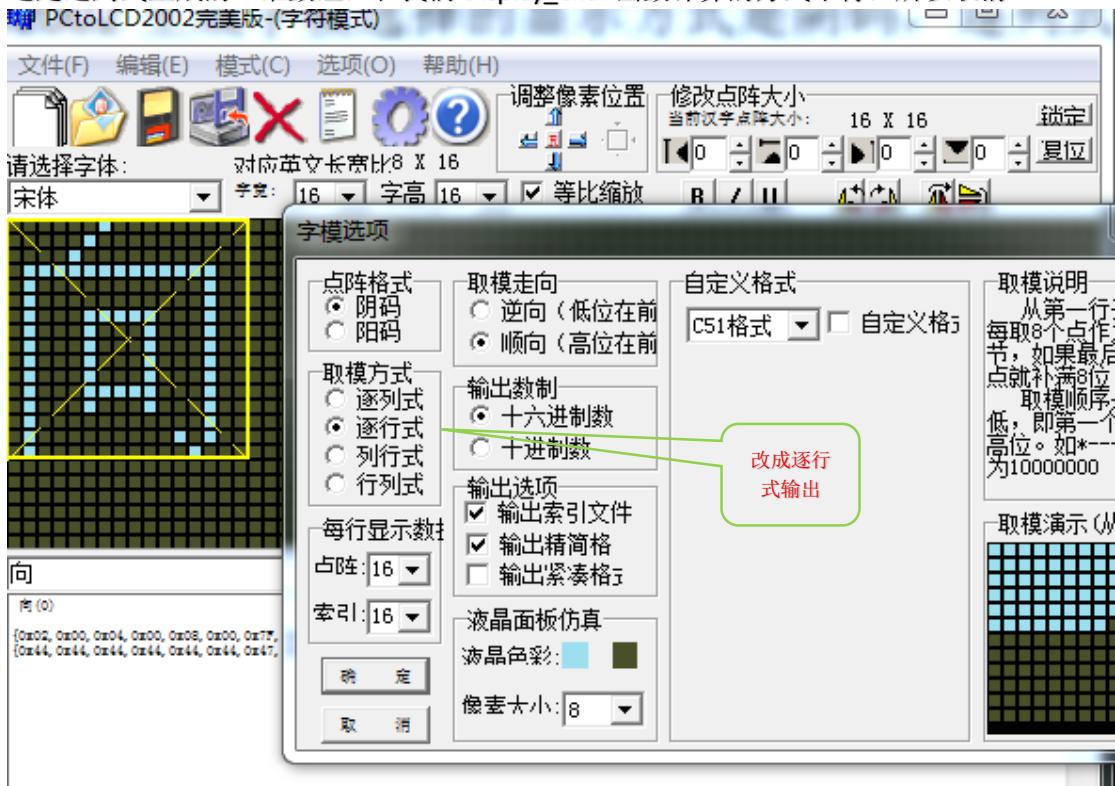
我们先将数组改成一维数组

```

uint_8 char_model[] = {0x00,0x00,0x07,0xF8,0x04,0x00,0x04,0x00,0x05,0xF0,0x0D,0x20,0x05,0x20,0x05,0x20,
0x05,0x20,0x05,0xE8,0x04,0x08,0x04,0x08,0x07,0xF8,0x00,0x00,0x00,0x00,0x00};/*"向",0*/

```

这是逐列式生成的一维数组，和我们 Dispaly_char 函数计算的方式不符，所以取消



```

/*逐列式*/
uint_8 char_model[] = {0x00,0x00,0x07,0xF8,0x04,0x00,0x04,0x00,0x05,0xF0,0x0D,0x20,0x05,0x20,0x05,0x20,
0x05,0x20,0x05,0xE8,0x04,0x08,0x04,0x08,0x07,0xF8,0x00,0x00,0x00,0x00,0x00};/*"向",0*/

/*逐行式*/
{0x02,0x00,0x04,0x00,0x08,0x00,0x7F,0xFC,0x40,0x04,0x40,0x04,0x47,0xC4,0x44,0x44},
{0x44,0x44,0x44,0x44,0x44,0x44,0x47,0xC4,0x44,0x44,0x40,0x04,0x40,0x14,0x40,0x08},/*"向",0*/

```

你看逐行式生成的数据和逐列式生成的数据不一样，我们下面取消逐列式，用逐行式数组，同时把逐行式二维数组做出一维数组。

把括号取消掉就是一维数组，然后用下面代码计算取字模

```
/*逐行式*/
uint_8 char_model[] = {0x02,0x00,0x04,0x00,0x08,0x00,0x7F,0xFC,0x40,0x04,0x40,0x04,0xC4,0x44,0x44,
0x44,0x44,0x44,0x44,0x44,0x44,0x47,0xC4,0x44,0x44,0x40,0x04,0x40,0x14,0x40,0x08};/*"向",0*/
void Dispaly_char(void)
{
    uint_8 row,byte,bit;

    for(row = 0;row < 16;row++)
    {
        for (byte = 0; byte < 2;byte++)
        {
            for(bit = 0; bit < 8; bit++)
            {
                if (char_model[row*2+byte] & (0x80 & bit))
                {
                    //点亮该像素点
                } else
                {
                    //点黑该像素点
                }
            }
        }
    }
}
```

这就是逐行式二维数组，和下面取模驱动代码匹配了

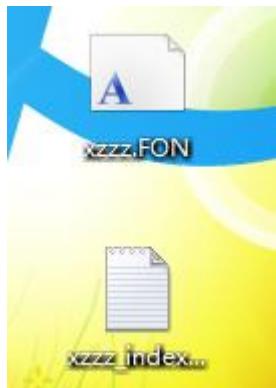
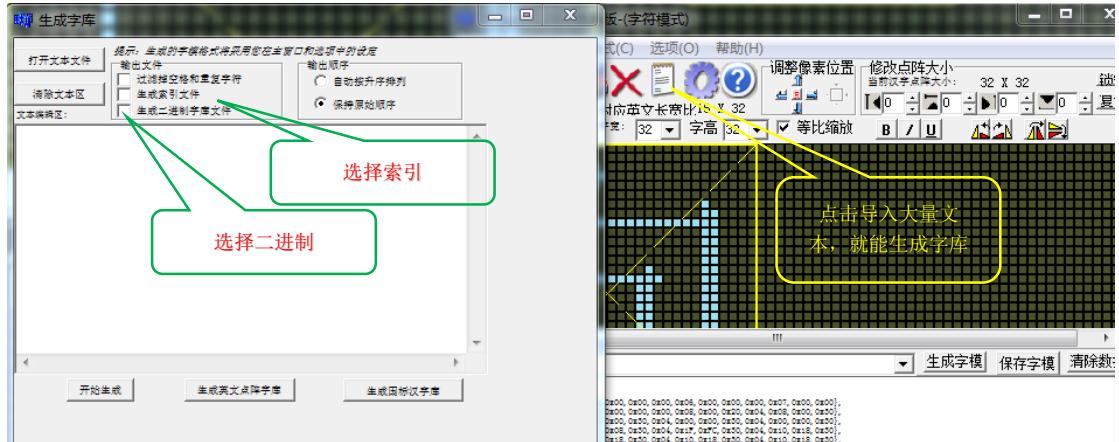
```
void Dispaly_char(void)
{
    uint_8 row,byte,bit;

    for(row = 0;row < 16;row++)
    {
        for (byte = 0;byte < 2;byte++)
        {
            for(bit = 0; bit < 8; bit++)
            {
                if (char_model[row*2+byte] & (0x80 & bit))
                {
                    //点亮该像素点
                } else
                {
                    //点黑该像素点
                }
            }
        }
    }
}
```

bit 循环 8 次把 0x02 显示完

```
/*逐行式*/
uint_8 char_model[] = [0x02,0x00,0x04,0x04,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x47,0xC4]
```


如果你要生成字库



这就是我生成的字库文件.FON 和索引文件.txt

```
1 0000 0000 0000 0000 0000 0000 0000 0000 0000  
2 0000 0000 0000 0000 0000 0000 0000 0000 0000  
3 0000 0000 0000 0000 0000 0000 0000 0000 0000  
4 0000 0000 0000 0000 0000 0000 0000 0000 0000  
5 0000 0000 0000 0000 0000 0000 0000 0000 0000  
6 0000 0000 0000 0000 0000 0000 0000 0000 0000  
7 0000 0000 0000 0000 0000 0000 0000 0000 0000  
8 0000 0000 0000 0000 0000 0000 0000 0000 0000  
9 0000 0000 0000 0000 0000 0000 0000 0000 0000  
0 0000 0000 0000 0000 0000 0000 0000 0000 0000  
1 ff00 0100 0100 0100 0100 0100 0100 0100 0100  
2 0100 0100 0100 0100 0100 0100 0100 0100 0100  
3 0000 0100 0100 0100 0100 0100 0100 0100 0100  
4 0100 0100 0100 0100 0100 0100 0100 0100 0100
```

这就是.fon 字库文件，但是我们要让它变成 16 进制，要加入 0x 和逗号，那么用 sublime 软件来修改。

A screenshot of a text editor showing a block of hex code. A yellow box highlights the first line: '0x00,0x00, 0x00,0x00 0x00,0x00 0x00,0x00'. A red callout box points to this line with the text: '就是全部要改成 0x 这样，然后拷贝进软件里面的数组，你可以用 sublime 快捷键来修改，自己寻找方法' (All of this needs to be changed to 0x like this, then copy it into the array in the software, you can use sublime's keyboard shortcut to modify it, find it yourself).

```
0x00,0x00, 0x00,0x00 0x00,0x00 0x00,0x00  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000
```

这种拷贝数组的做法，英文字符是可以的，但是中文字符就不能这样做了，因为汉字太多。

根据 xzzz_index.txt 索引表来编写 ASCII 码字模寻址程序

```
SOH|STX|ETX|EOT|ENQ|ACK|BEL|BS      VT|FF|SO|SI|DLE|DC1|DC2|DC3|DC4|NAK|SYN|NETB|CAN|EM|SUB|ESC|FS|GS|RS|US !#$%&(')*+,-./*\0123456789:;=>?@ABCDEFGHIJKLMNPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~|
```

前面控制符我们是不要的，我们就要先确定 ‘! ’感叹号在第几个字节，感叹号前面还有个空格

我们知道‘!’感叹号在第 32 个位置，32 之前的位置被控制字符占用了
那么在数组里面，‘!’就放在第 32 个元素下标的位置。

ASCII 码字模地址寻找计算公式

比如字模大小是 8x16

那么这个字模占用内存大小就是 $8 \times 16 / 8 = 16$ 字节

↑
这里除以8是每个字节
是8位

英文字库第 0 个字节存储的是 SOH

```
char temp ;  
temp = '!'; //如果IDE软件设置的编码方式是ASCII模式，那么这个！= 0x21  
  
offset = '!'(0x21) + 0X32(绕过控制符) - (0x20也就是空格位置)  
offset= '!' = 0x33，感叹号正好在数组0x33位置
```

然后用自定义的 Display_char 去获取数组的 0x33 下标的数据，用来刷屏，显示 ‘!’ 感叹号

```
/* 常用ASCII表，偏移量32，大小:16 * (高度) * 8 * (宽度)  
*/  
const uint8_t ASCII18x16_Table[ ] = { /* @conslons字体，阴码点格式，逐行顺向取摸 */  
0x00,  
0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x08, 0x08, 0x18, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x34, 0x24, 0x24, 0x24, 0x24, 0x24, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x16, 0x24, 0x7f, 0x24, 0x24, 0x24, 0x24, 0x24, 0x24, 0x00, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x68, 0x48, 0x68, 0x1c, 0x16, 0x12, 0x12, 0x7c, 0x10, 0x10, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x61, 0xd2, 0x96, 0x74, 0x08, 0x10, 0x16, 0x29, 0x49, 0xc6, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x3c, 0x64, 0x44, 0x38, 0x72, 0x4a, 0xce, 0x46, 0x7f, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x04, 0x08, 0x18, 0x10, 0x30, 0x30, 0x30, 0x10, 0x10, 0x18, 0xc, 0x04,  
0x00, 0x00, 0x00, 0x20, 0x10, 0x08, 0x08, 0x0c, 0x04, 0x04, 0x04, 0x0c, 0x08, 0x18, 0x10, 0x20,  
0x00, 0x00, 0x00, 0x08, 0x08, 0x34, 0x1c, 0x6a, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0x7f, 0x18, 0x18, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x18, 0x08, 0x30, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x02, 0x06, 0x04, 0xc, 0x08, 0x08, 0x08, 0x10, 0x10, 0x20, 0x20, 0x40, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x3c, 0x66, 0x42, 0x47, 0xb5, 0x73, 0x42, 0x66, 0x3c, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x18, 0x78, 0x48, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x7e, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x3c, 0x46, 0x06, 0x06, 0x04, 0x04, 0x08, 0x10, 0x20, 0x7e, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x7c, 0x06, 0x06, 0x04, 0x3c, 0x02, 0x02, 0x06, 0x7c, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x0c, 0x14, 0x24, 0x64, 0x44, 0xff, 0x04, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x7e, 0x60, 0x60, 0x60, 0x7e, 0x02, 0x02, 0x06, 0x7c, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x1e, 0x30, 0x60, 0x48, 0x76, 0x42, 0x42, 0x62, 0x3c, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x7e, 0x02, 0x06, 0x04, 0x0c, 0x08, 0x18, 0x10, 0x30, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x3c, 0x62, 0x42, 0x36, 0x1c, 0x66, 0x42, 0x42, 0x3c, 0x00, 0x00, 0x00, 0x00,
```

这就是字模软件生产的所有 ASCII 字体的数组

下面使用程序在这些数组里面找到指定的 ASCII 字体显示

```

typedef struct font
{
    const uint_8 *table;
    uint_16 width;
    uint_16 height;
}xfont;

xfont Font8x16 = {
    ASCII8x16_Table, //放入数组
    8, //设置字体宽8
    16, //设置字体高16
};

```

```

/*
* x,y字体显示位置
* xchar填入要显示的单个ASCII字符
*/
void Dispaly_char_xzz(uint_16 x,uint_16 y,const char xchar)
{
    uint_16 position;
    uint_8 font_length = 0;
    uint_8 *pfont;

    position = xchar + 0x32 - ' ';
    font_length = (Font8x16.width*Font8x16.height)/8; // 8x16/8 = 16字节，每个英文16字节大小

    pfont = (uint_8 *) &Font8x16.ASCII8x16_Table[position*font_length]; //计算该字模在数组里面的偏移位置

    LCDDisplay(x,y); //设置窗口位置

    for(row = 0;row < font_length; font_length++)
    {
        for(bit = 0; bit < 8; bit++)
        {
            if(pfont[row]&(0X80>>bit))
                //显示颜色
            else
                //不显示
        }
    }
}

```

感叹号在 33 位置

ASCII8x16_Table[] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0xff,0x01,0x03,0x06,0x08,0x00,0x00,
.....,
0x00,0x00,0x10,0x00,0x10,0x00,0x10,0x00,
0x10,0x00,0x10,0x00,0x00,0x00,0x00,0x10};

'!' 感叹号在数组开始
地址0x33*16的位置

这是单个字符显示的程序

字符串显示程序范例

```

void Dispaly_String(uint_16 x,uint_16 y,char *pStr)
{
    while(*pStr != '\0') //每次循环取出字符串其中一个字符，直到遇到\0结束
    {
        Dispaly_char_xzz(x,y,*pStr); //显示一个字符
        pStr++; //获取下一个字符显示
        x += Font8x16.width; //下一次循环要加上上一次字体宽度，跳过上一次字体位置
    }
}

```

以上就是字符和字符串显示原理伪代码分析，实际情况可以去拷贝开源网站的一些代码库，修改 API，调用 API 就是了。

中文显示原理

GB2312中文编码16X16汉字存放数组方式

```
CH16x16_Table[] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  
    0x01,0x02,0xff,0xfe,0x00,0x32,0x22,0x23,  
    0x01,0x02,0xff,0xfe,0x00,0x32,0x22,0x23,  
    0x01,0x02,0xff,0xfe,0x00,0x32,0x22,0x23,  
    0x01,0x02,0xff,0xfe,0x00,0x32,0x22,0x23,  
    0x01,0x01,0xff,0xf1,0x00,0x31,0x22,0x23,  
    0x02,0x02,0xff,0xfe,0x00,0x31,0x22,0x23,  
    0x03,0x02,0xff,0xf1,0x00,0x31,0x22,0x23,  
    0x03,0x02,0xff,0xf1,0x00,0x31,0x22,0x23}
```

每一个16X16汉字用
32个字节数组表示

GB2312 使用区位码的方式来存储

中华人民共和国国家标准 信息交换用汉字编码字符集 基本集 GB 2312-80										
01	0	1	2	3	4	5	6	7	8	9
0	、	.	-	~	…	“	”	()	々
1	-	—	॥	…	·	“	”	()	々
2	<	>	《	》	「	」	『	』	〔	〕
3	【	】	±	×	÷	:	△	▽	Σ	∏
4	U	∩	E	:::	/	⊥	/	^	○	○
5	ʃ	ø	=	≈	∞	∞	≠	≠	≠	≠
6	≤	≥	≈	∞	∞	≠	≠	≠	≠	≠
7	℃	\$	￠	£	%	§	№	☆	★	★
8	O	●	◎	◇	◆	□	■	△	▲	※
9	→	←	↑	↓	=					
02	0	1	2	3	4	5	6	7	8	9
0	i	ii	iii	iv	v	vi	vii	viii	ix	
1	x	□	□	□	□	□	1.	2.	3.	
2	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
3	14.	15.	16.	17.	18.	19.	20.	(1)	(2)	(3)
4	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
5	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(1)	(2)	(3)
6	(4)	(5)	(6)	(7)	(8)	(9)	(10)	€	□	(4)
7	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
8	□	I	II	III	IV	V	VI	VII	VIII	IX
9	X	XI	XII	□	□					
21	0	1	2	3	4	5	6	7	8	9
0	急	耽	担	丹	单	鄭	掸	胆	日	
1	氮	但	惮	淡	诞	弹	蛋	当	挡	党
2	荡	档	刀	捣	蹈	倒	岛	祷	导	到
3	稻	悼	道	盜	德	得	的	蹬	灯	登
4	等	瞪	凳	鄧	堤	低	滴	迪	敌	笛
5	狱	涤	翟	嫡	抵	底	地	蒂	帝	第
6	弟	递	缔	顛	掂	滇	碘	点	典	靛
7	垫	电	佃	甸	店	惦	奠	碇	殿	碉
8	叼	雕	凋	刁	掉	吊	钓	调	跌	爹
9	碟	蝶	迭	谍	叠					

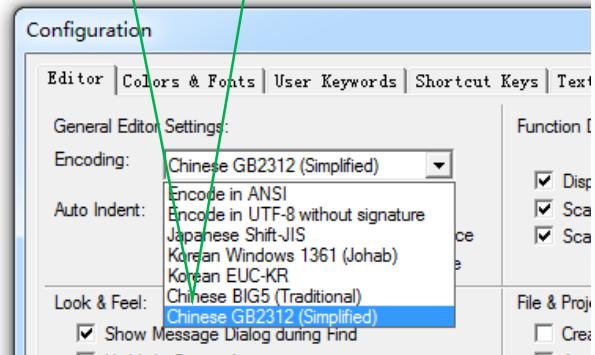
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	20	00	18	00	0C	00	04	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	18	00	24	00	24	00	18	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	01	80	01	80	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	00	00	08	20	04	40	02	80	01	00	00	00	00	00	00	00
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000C0	00	00	00	00	00	00	0C	60	0C	60	00	00	00	00	00	00
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0	00	00	00	00	00	00	00	00	02	40	02	40	02	40	04	80
000000F0	04	80	09	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100	00	00	00	00	04	00	04	00	04	20	F0	08	20	10	40	00
00000110	20	80	05	00	02	00	01	00	01	00	00	00	00	00	00	00
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	7F	FC
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	1C

这是软件生成的国标字库

The screenshot shows a Windows Notepad window titled "ch_index.TXT - 记事本". It contains a large amount of Chinese text in GB2312 encoding. The text is mostly gibberish but includes recognizable characters like '我' (Me) and '你' (You). The encoding dropdown in the Notepad's menu bar is set to "GB2312".

索引表

将 Keill IDE 设置成 GB2312 编码方式，这样你
写入的字符就会在软件内部变成国标码



```
#include <stdio.h>
#include <lcd.h>

int main(void)
{
    uint32_t i=0;
    char temp = '我';
}
```

输入：汉字

查询

'我' 这个字符国标码就是 CED2

GB2312编码 : CED2 GBK编码 : CED2 GB18030编码 : CED2

将 '我' 国标码 CED2 分解成高8位CE, 和低8位D2

CE-A0 = 区码

D2-A0 = 位码

区位码计算公式如下

$$Addr = (((Code_H - 0xA0 - 1) * 94) + (Code_L - 0xA0 - 1)) * 16 * 16 / 8$$

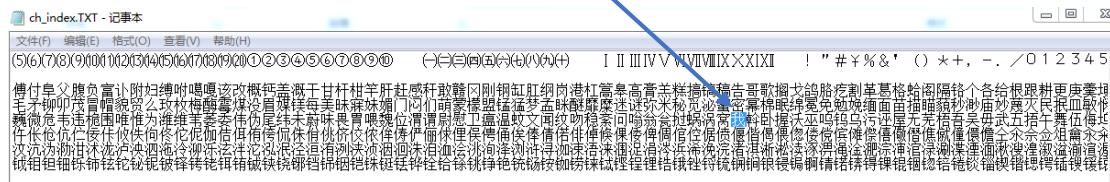
Code_H 就是'我'的高8位(CE)

Code_L 就是'我'的低8位(D2)

16*16/8是汉字大小
用的16X16

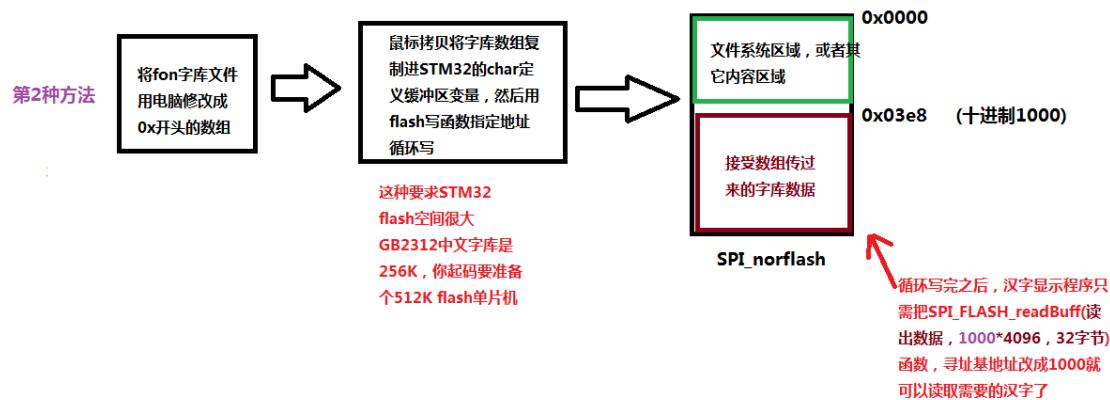
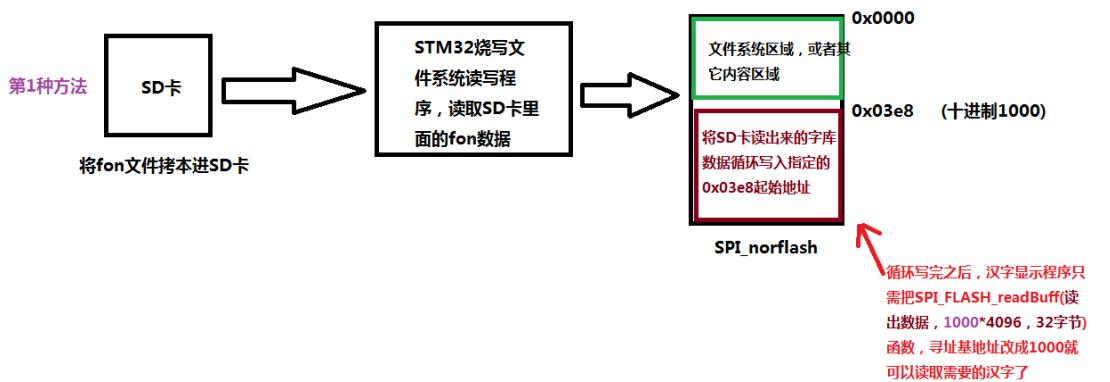
计算出来的Addr就是索引表的偏移

这个偏移*字体大小(16*16/8)就是该汉字在索引表的具体位置，也就是数组的具体开始下标



下面从 SPI_FLASH 读取汉字字库来显示中文

汉字显示之前我们要搞清楚如何将字库拷贝进 SPI_FLASH



如果用 SD 卡做字库，逻辑和读写 SPI FLASH 一样的

汉字显示程序

```
void Dispaly_CH_16X16(uint_16 x,uint_16 y,uint_16 xchar)
{
    uint_8 buf[16*16/8];//缓存区大小根据字体大小来分配,这里是32字节
    uint_16 Hight = 16;//字体高度
    uint_16 width = 16;//字体宽度
    uint_16 temp;//存放寻找区位码返回的汉字具体16进制值
    int row;
    unsigned char Hig8bit,Low8bit;
    unsigned int pos;

    SPI_FLASH_init(); //初始化SPI FLASH

    Hig8bit = xchar >> 8; //汉字国标码高8位放入区码
    Low8bit = xchar & 0x00ff; //汉字国标码低8位放入位码

    pos = (((Hig8bit-0xa0-1)*94)+(Low8bit-0xa0-1))*16*16/8; //获取该汉字在flash字库的偏移地址
    SPI_FLASH_readBuffer(buf,(1000*4096)+pos,16*16/8); //读出来的buf[32]={0x01, 0x52, 0x33.....0x32} 有32字节

    for(row = 0; row < (16*16/8); row++) //每次循环填充一行,一共填充32行
    {
        for (bit = 0; bit < 16; bit++)
        {
            if(buf[row] & (0x8000 >> bit)) //高位在前
                //点亮像素点
            else
                //不点亮像素点
        }
    }
}
```

0x0000
文件系统区域,或者其它内容区域
0x03e8 (十进制1000)
将SD卡读出来的字库数据循环写入指定的0x03e8起始地址
SPI_norflash
循环写完之后,汉字显示程序只需要把SPI_FLASH_readBuff(读出数据,1000*4096,32字节)函数,寻址基地址改成1000就可以读取需要的汉字了

如何支持中英文显示,

```
/*中英文显示函数*/
void Dispaly_CH_EN(uint_16 x,uint_16 y,char* CEchar)
{
    while(*CEchar != '\0')
    {
        if(CEchar <= 126) //显示英文字符
        {
            Dispaly_char_xzz(x,y,*CEchar);
            //显示英文,在flash或者直接数组备份一块ASCII字模区域
            CEchar++;
        }
        else
        {
            Dispaly_CH_16X16(x,y,*CEchar); //读取flash字库文件显示中文
            CEchar += 2; //中文是两个字节
        }
    }
}
```

其余的字体放大缩小,变化的字符显示,请查阅原子或者野火的范例代码。

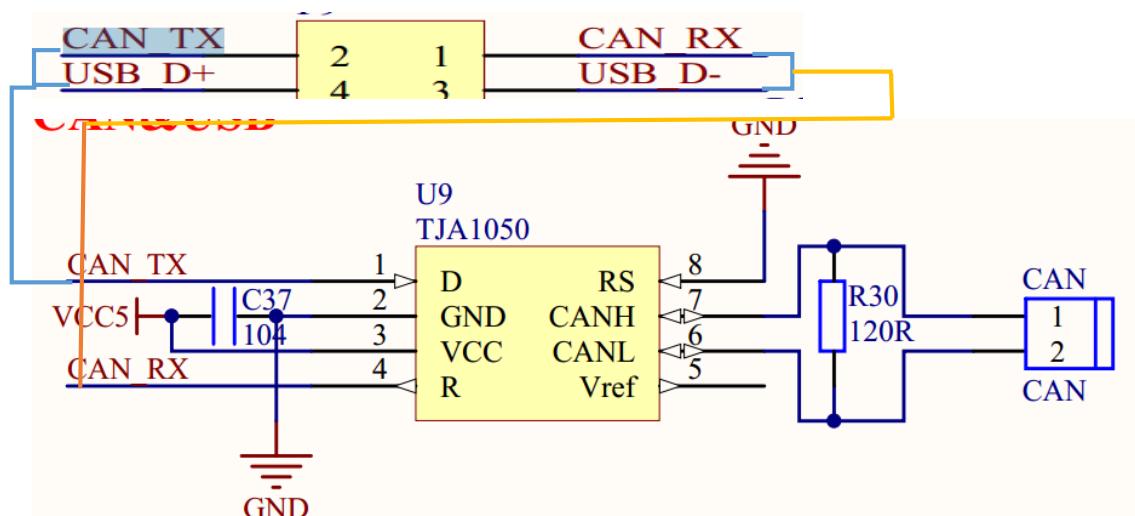
STM32 内部 CAN 总线使用

CAN 双工通信，波特率最高 1M。1M 通信距离 50 米以内，5K 通信距离 10 公里。

CAN 是多主控制，意思就是几个 CAN 连接一条线上，没有主从之分，每个节点都可以发送。CAN 总线优先级由每个芯片的 ID 来决定，ID 越小优先级越高。

USB D-	PA11 103	PA11/USART1_CTS/CAN_RX/TIM1_CH4/USBDM
USB D+	PA12 104	PA12/USART1_RTS/CAN_TX/TIM1_ETR/USBDP
TMS	PA10 105	

STM32F103 CAN 控制器输出是 CAN_RX 和 CAN_TX，我们要将这两个信号转换成显性和隐性。



CAN 收发器一般是 TJA1050 或者是 82C250 芯片

51 单片机不像 STM32 那样有内部 CAN 控制器所以需要外挂 MCP2515 或 SJA1000 AN 芯片
然后将 MCP2515 芯片外接一个 TJA1050 收发器，这样 51 单片机也具备 CAN 功能了。

CAN 物理层的形式主要有两种

一种遵循 ISO11898 标准的高速、短距离“闭环网络”，它的总线最大长度为 40m，通信速度最高为 1Mbps，总线的两端各要求有一个“120 欧”的电阻

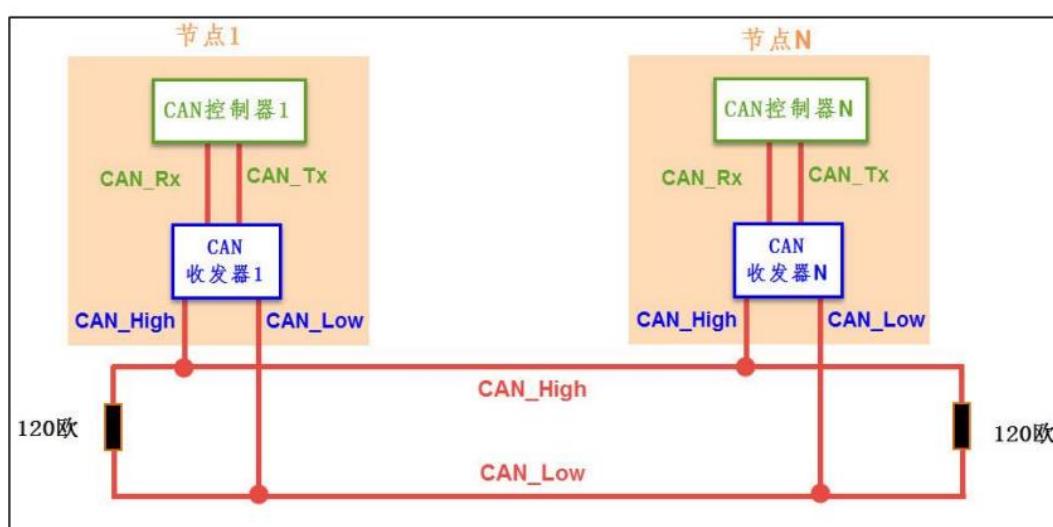


图 42-1 CAN 闭环总线通讯网络

另一种是 ISO11519-2 标准低速 CAN 网络，远距离“开环网络”最大传输距离 1Km，最高通讯速率 为 125kbps，两根总线是独立的、不形成闭环

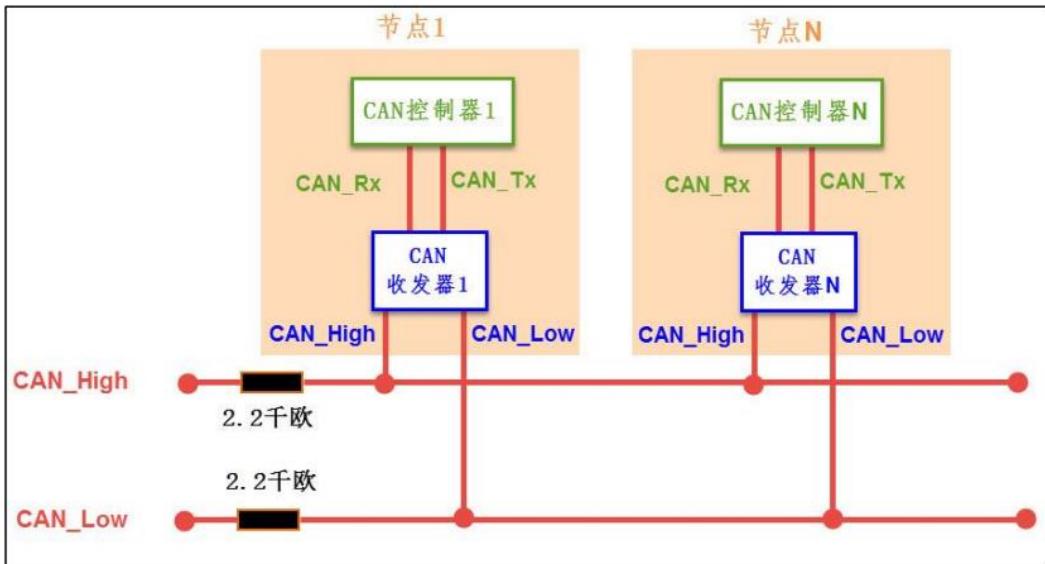


图 42-2 CAN 开环总线通讯网络

CAN 总线帧属性

遥控帧(也有人叫远程帧): CAN 控制器发送一帧远程帧，这帧后面没有数据，只是发送一个 ID 到总线上，有符合该 ID 的 CAN 控制器响应，返回数据给发送者。这就是该帧的功能。类似于我叫某人一声，某人回复我一大堆废话。

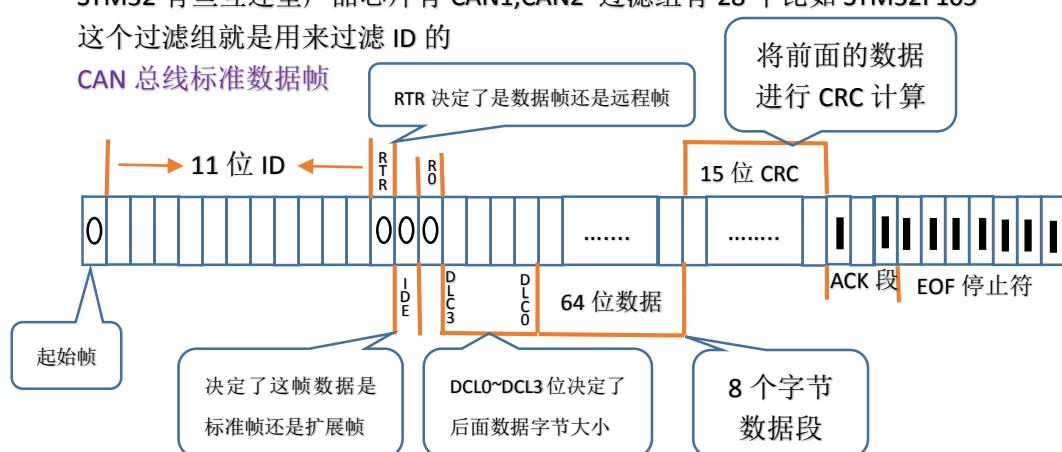
可变过滤组

STM32F103ZET6 只有 CAN1 过滤组 14 个

STM32 有些互连型产品芯片有 CAN1,CAN2 过滤组有 28 个比如 STM32F105

这个过滤组就是用来过滤 ID 的

CAN 总线标准数据帧



CAN 总线扩展数据帧

我们先上一个标准帧



CAN 总线波特率设置

在 STM32F1 系列主时钟频率在 72Mhz 情况下的 CAN 波特率参数表，其它型号不一定是这个表

SJW	BS1	BS2	Prescale	波特率
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	6	1M
CAN_SJW_1tq	CAN_BS1_4tq	CAN_BS2_3tq	5	900K
CAN_SJW_1tq	CAN_BS1_5tq	CAN_BS2_3tq	5	800K
CAN_SJW_1tq	CAN_BS1_6tq	CAN_BS2_3tq	6	600K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	12	500K
CAN_SJW_1tq	CAN_BS1_5tq	CAN_BS2_3tq	10	400K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	20	300K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	24	250K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	30	200K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	40	150K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	48	125K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	60	100K
CAN_SJW_1tq	CAN_BS1_4tq	CAN_BS2_3tq	50	90K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	75	80K
CAN_SJW_1tq	CAN_BS1_6tq	CAN_BS2_3tq	60	60K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	120	50K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	150	40K
CAN_SJW_1tq	CAN_BS1_6tq	CAN_BS2_3tq	120	30K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	300	20K
CAN_SJW_1tq	CAN_BS1_3tq	CAN_BS2_2tq	600	10K
CAN_SJW_2tq	CAN_BS1_6tq	CAN_BS2_4tq	600	5K
CAN_SJW_2tq	CAN_BS1_6tq	CAN_BS2_4tq	1000	3K
CAN_SJW_2tq	CAN_BS1_10tq	CAN_BS2_6tq	1000	2K

CAN 总线屏蔽滤波功能

STM32F 系列屏蔽滤波分两种：

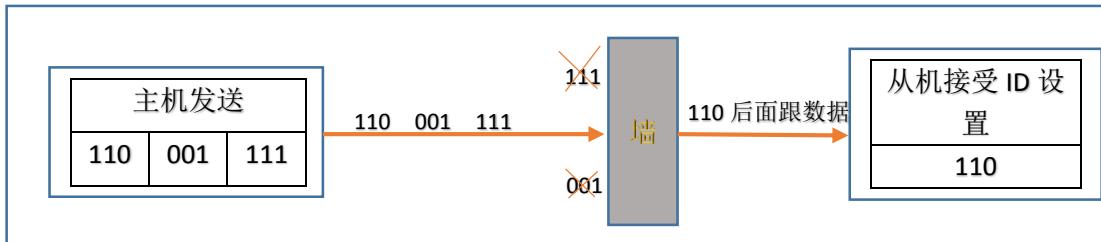
1. 屏蔽位模式

屏蔽位模式就是我设置个 101

1	0	1



2. 标识符列表模式



标识符模式就是从机设置好 ID 地址，然后主机放送过来的 ID 于从机匹配从机就接受，主机其他的 ID 从机屏蔽掉不接受。

图202 过滤器组位宽设置—寄存器组织

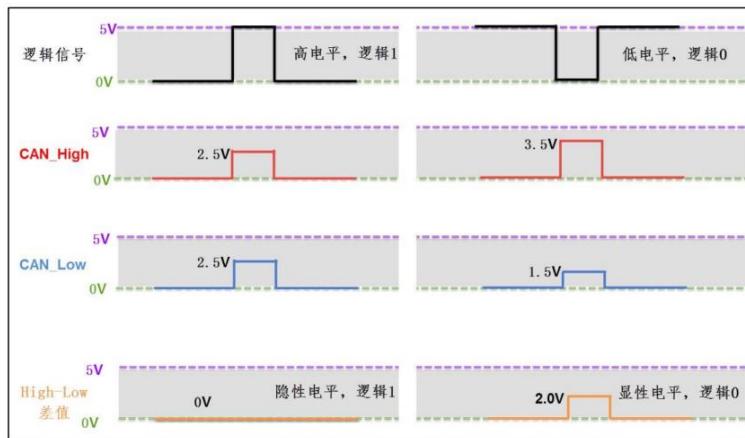
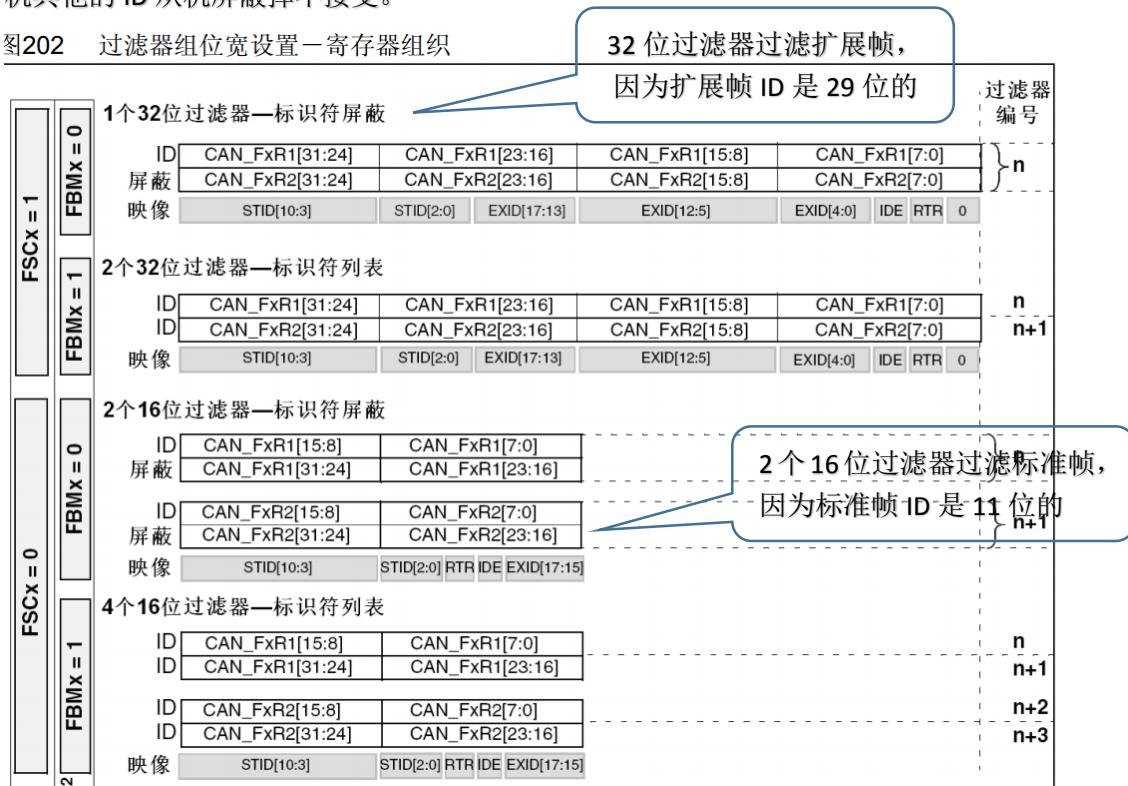


图 42-4 CAN 的差分信号（高速）

下面我们来进行 CAN 总线实验

首先看 CAN 端口要不要映射，STM32 CAN1 有 USB 口功能，如果不用 USB 可以不用映射，但是要用 USB 就必须要映射，我们这里映射试试。

8 通用和复用功能(GPIO和AFIO)			
8.1 GPIO功能描述			
8.2 GPIO寄存器描述			
8.3 复用功能GPIO和I/O端口配置(AFIO)			
8.3.1 把OSC2_IN/OSC2_OUT作为GPIO端口PC14/PC15			
8.3.2 把OSC_IN/OSC_OUT引脚作为GPIO端口PD0/PD1			
8.3.3 CAN1 复用功能重映射			
8.3.4 CAN2 复用功能重映射			
8.3.5 JTAG/SWD 复用功能重映射			
8.3.6 ADC 复用功能重映射			
8.3.7 定时器复用功能重映射			
8.3.8 USART 复用功能重映射			
8.3.9 I2C 复用功能重映射			
8.3.10 SPI1 复用功能重映射			
8.3.11 SPI2 复用功能重映射			
8.3.12 以太网复用功能重映射			
8.4 AFIO寄存器描述			
8.5 GPIO 和AFIO寄存器地址映象			

8.3.3 CAN1 复用功能重映射

CAN信号可以被映射到端口A、端口B或端口D上，如下表所示。对于端口D，在36、48和64脚的封装上没有重映射功能。

表32 CAN1复用功能重映射

复用功能 ⁽¹⁾	CAN_REMAP[1:0]="00"	CAN_REMAP[1:0]="10" ⁽²⁾	CAN_REMAP[1:0]="#11" ⁽³⁾
CAN1_RX 或 AN_RX	PA11	PB8	PD0
CAN1_TX 或 AN_TX	PA12	PB9	PD1

- 在互连型产品中是CAN1_RX和CAN1_TX；在其它带有单个CAN接口的产品中是CAN_RX和CAN_TX。
- 重映射不适用于36脚的封装
- 当PD0和PD1没有被重映射到OSC_IN和OSC_OUT时，重映射功能只适用于100脚和144脚的封装上。

```
/**  
 * @brief CAN 初始化结构体  
 */  
typedef struct {  
    uint16_t CAN_Prescaler; /*配置 CAN 外设的时钟分频，可设置为 1-1024*/  
    uint8_t CAN_Mode; /*配置 CAN 的工作模式，回环或正常模式*/  
    uint8_t CAN_SJW; /*配置 SJW 极限值 */  
    uint8_t CAN_BS1; /*配置 BS1 段长度*/  
    uint8_t CAN_BS2; /*配置 BS2 段长度 */  
    FunctionalState CAN_TTCM; /*是否使能 TTCM 时间触发功能*/  
    FunctionalState CAN_ABOM; /*是否使能 ABOM 自动离线管理功能*/  
    FunctionalState CAN_AWUM; /*是否使能 AWUM 自动唤醒功能 */  
    FunctionalState CAN_NART; /*是否使能 NART 自动重传功能*/  
    FunctionalState CAN_RFLM; /*是否使能 RFLM 锁定 FIFO 功能*/  
    FunctionalState CAN_TXFP; /*配置 TXFP 报文优先级的判定方法*/  
} CAN_InitTypeDef;
```

这是 CAN 波特率设置，按照上面波特率表填入对应参数

触发功能我们这个 CAN2.0 标准用不到，选择 DISABLE

自动离线管理就是节点出错后自动恢复选择 ENABLE

▶ AWUM: 自动唤醒，就是低功耗功能，选择 ENABLE。

▶ NART: 一个报文发送失败，是继续重复发送这个报文呢？还是放弃这个报文发送新报文，选择 ENABLE 自动重复发送失败的报文，直到发送成功为止

RFLM: 如果 ENABLE 那么 fifo 接受数据满了，会丢弃后面来的数据。

TXFP: 选择 DISABLE，按照报文 ID 优先级来选择谁先发送。

```
typedef struct {  
    uint32_t StdId; /*存储报文的标准标识符 11 位, 0-0x7FF. */  
    uint32_t ExtId; /*存储报文的扩展标识符 29 位, 0-0x1FFFFFFF. */  
    uint8_t IDE; /*存储 IDE 扩展标志 */  
    uint8_t RTR; /*存储 RTR 远程帧标志*/  
    uint8_t DLC; /*存储报文数据段的长度, 0-8 */  
    uint8_t Data[8]; /*存储报文数据段的内容 */  
} CanTxMsg;
```

StdId 是写 11 位 CAN ID, ExtId 是写 29 位 CAN ID, 我选 29 位的

扩展帧 CAN_Id_Extended
就是 29 位 ID
标准帧 CAN_Id_Standard
就是 11 位 ID

▶ RTR: 选择 CAN_RTR_Remote 是遥控帧，这样 Data[8]数组的 8 个数据就不会被发送
选择 CAN_RTR_Data 是数据帧，这样 Data[8]数组里面的 8 个数据会被发送

DLC: 就是要发送多少个数据，我这里填 8。Data[8]:是数据，你要建立 8 个元素的数组向 Data 数组里面写数据

```

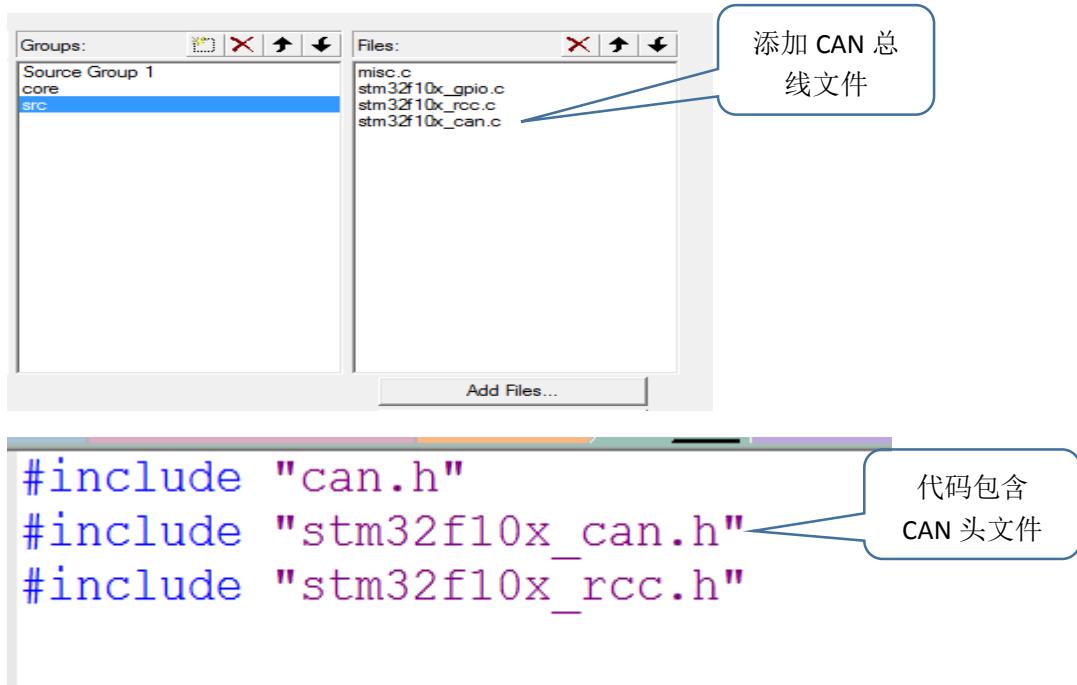
typedef struct {
    uint32_t StdId; /*存储了报文的标准标识符 11 位, 0-0x7FF. */
    uint32_t ExtId; /*存储了报文的扩展标识符 29 位, 0-0x1FFFFFFF. */
    uint8_t IDE; /*存储了 IDE 扩展标志 */
    uint8_t RTR; /*存储了 RTR 远程帧标志*/
    uint8_t DLC; /*存储了报文数据段的长度, 0-8 */
    uint8_t Data[8]; /*存储了报文数据段的内容 */
    uint8_t FMI; /*存储了 本报文是由经过筛选器存储进 FIFO 的, 0-0xFF */
} CanRxMsg;

```

接受 CAN 数据的结构体和发送数据结构体一样，只是这里是接受的数据填入这个结构体。

下面先测试 CAN 回环模式，确保代码编写没有问题

添加 can 总线的配置文件



```

#include "can.h"
#include "stm32f10x_can.h" // 代码包含 CAN 头文件
#include "stm32f10x_rcc.h"

/*CAN引脚初始化*/
void CAN_init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE); // 初始化CAN1的时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO, ENABLE);
    // 打开PA11, PA12时钟和复用时钟AFIO, CAN1我们用的PA11, PA12引脚

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12; // PA12 CAN TX
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; // CAN TX发送引脚要设置复用模式
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11; // PA11 CAN RX
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; // CAN RX接受引脚可以设置成上拉输入
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

```

```

/*CAN波特率和功能设置*/
void CAN_Config(void)
{
    CAN_InitTypeDef CAN_InitTypeStruct;

    CAN_InitTypeStruct.CAN_ABOM = ENABLE;
    CAN_InitTypeStruct.CAN_AWUM = ENABLE; //自动唤醒功能打开

    /*波特率1M设置BS1,BS2,prescaler,SJW*/
    CAN_InitTypeStruct.CAN_BS1 = CAN_BS1_3tq;
    CAN_InitTypeStruct.CAN_BS2 = CAN_BS2_2tq;
    CAN_InitTypeStruct.CAN_Prescaler = 6;
    CAN_InitTypeStruct.CAN_SJW = CAN_SJW_1tq;

    CAN_InitTypeStruct.CAN_Mode = CAN_Mode_LoopBack; //设置CAN为回环模式
    CAN_InitTypeStruct.CAN_NART = ENABLE; //支持报文重传
    CAN_InitTypeStruct.CAN_TTCM = DISABLE; //这个功能我们CAN2.0用不上
    CAN_InitTypeStruct.CAN_TXFP = DISABLE; //按照报文ID优先级来选择先发?
    CAN_InitTypeStruct.CAN_RFLM = ENABLE; //锁定fifo功能

    CAN_Init(CAN1, &CAN_InitTypeStruct); //我们用CAN1,在STM32F105里面有CAN2
}

```

```

/*CAN过滤器配置*/
void CAN_Filter(uint32_t ID)
{
    CAN_FilterInitTypeDef CAN_FilterTypeStruct;

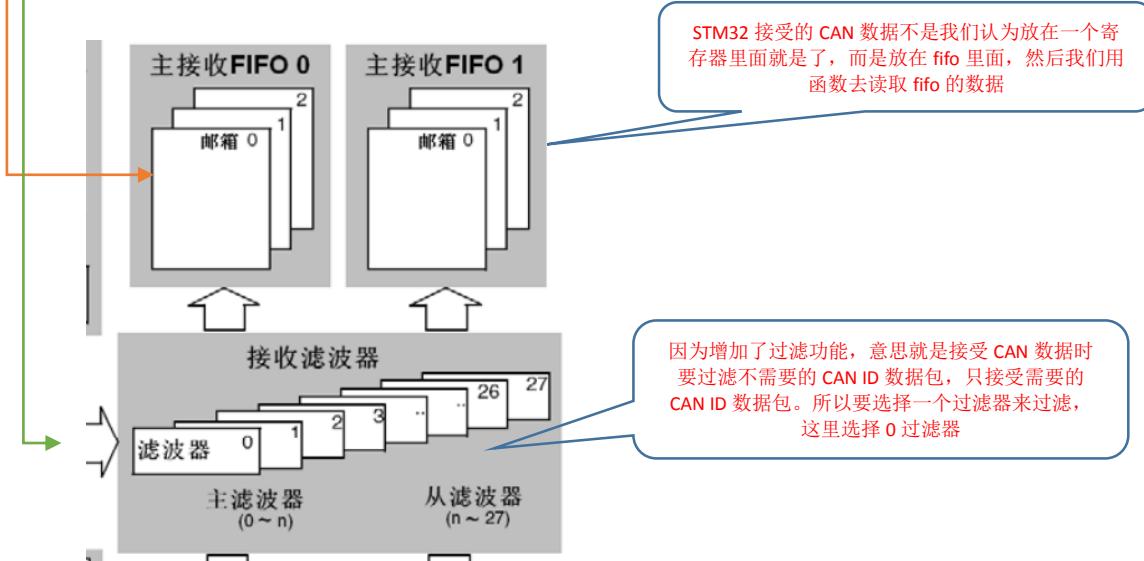
    CAN_FilterTypeStruct.CAN_FilterActivation = ENABLE; //是否启动过滤器,我这里启动
    CAN_FilterTypeStruct.CAN_FilterFIFOAssignment = CAN_Filter_FIFO0; //过滤出需要ID的数据就放在fifo0
    CAN_FilterTypeStruct.CAN_FilterNumber = 0;
    CAN_FilterTypeStruct.CAN_FilterScale = CAN_FilterScale_32bit; //用32位过滤器
    CAN_FilterTypeStruct.CAN_FilterMode = CAN_FilterMode_IdMask;

    CAN_FilterTypeStruct.CAN_FilterIdHigh = (((uint32_t)(ID<<3|CAN_Id_Extended|CAN_RTR_Data))&0xFFFF0000)>>16;
    CAN_FilterTypeStruct.CAN_FilterIdLow = ((uint32_t)(ID<<3|CAN_Id_Extended|CAN_RTR_Data))&0xFFFF;

    CAN_FilterInit(&CAN_FilterTypeStruct);

    CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE); //打开CAN1接受中断使能,FMP0就是接受邮箱0产生中断
}

```



下面把 CAN 过滤器配置函数分析完

```

/*CAN过滤器配置*/
void CAN_Filter(uint32_t ID)
{
    CAN_FilterTypeDef CAN_FilterTypeStruct;

    CAN_FilterTypeStruct.CAN_FilterActivation = ENABLE; //是否启动过滤器,我这里启动
    CAN_FilterTypeStruct.CAN_FilterFIFOAssignment = CAN_Filter_FIFO0; //过滤出需要ID的数据就放在fifo0
    CAN_FilterTypeStruct.CAN_FilterNumber = 0;
    CAN_FilterTypeStruct.CAN_FilterScale = CAN_FilterScale_32bit; //用32位过滤器
    CAN_FilterTypeStruct.CAN_FilterMode = CAN_FilterMode_IdMask;

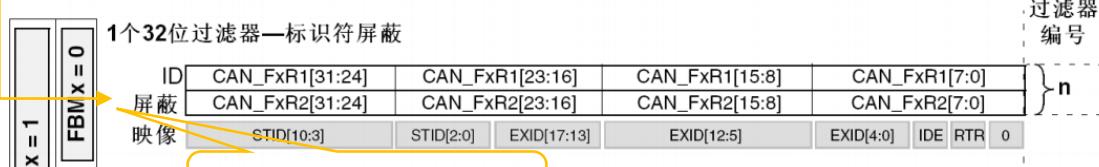
    CAN_FilterTypeStruct.CAN_FilterIdHigh = (((uint32_t)(ID<<3|CAN_Id_Extended|CAN_RTR_Data))&0xFFFF0000)>>16;
    CAN_FilterTypeStruct.CAN_FilterIdLow = ((uint32_t)(ID<<3|CAN_Id_Extended|CAN_RTR_Data))&0xFFFF;

    CAN_FilterTypeStruct.CAN_FilterMaskIdHigh = 0xffff;
    CAN_FilterTypeStruct.CAN_FilterMaskIdLow = 0xffff;

    CAN_FilterInit(&CAN_FilterTypeStruct);

    CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE); //打开CAN1接受中断使能, FMP0就是接受邮箱0产生中断
}

```



过滤器设置为屏蔽过滤功能

ID	CAN_FxR1[31:24]	CAN_FxR1[23:16]	CAN_FxR1[15:8]	CAN_FxR1[7:0]
屏蔽	CAN_FxR2[31:24]	CAN_FxR2[23:16]	CAN_FxR2[15:8]	CAN_FxR2[7:0]
映像	STID[10:3]	STID[2:0]	EXID[17:13]	EXID[12:5]
				EXID[4:0] IDE RTR 0

CAN_FilterTypeStruct.CAN_FilterIdHigh = (((uint32_t)(ID<<3|CAN_Id_Extended|CAN_RTR_Data))&0xFFFF0000)>>16;
 CAN_FilterTypeStruct.CAN_FilterIdLow = ((uint32_t)(ID<<3|CAN_Id_Extended|CAN_RTR_Data))&0xFFFF;

我们传入进来的 ID 变量要左移 3 位才能正常写入寄存器 CAN 标识符 ID 的对应位

这里又移 16 位是因为我们要写入 ID 的高 16 位到 FilterIdHigh 高寄存器里面

FilterIdLow 寄存器也是一样的将 CAN ID 低 16 位左移 3 位右移 16 位

CAN_FilterTypeStruct.CAN_FilterMaskIdHigh = 0xffff;
 CAN_FilterTypeStruct.CAN_FilterMaskIdLow = 0xffff;

比如FilterIdHigh = 1111 1111 0000 0000 (0xf0)

FilterIdLow = 0000 0000 1111 1111 (0x0f)

那么CAN ID就是0xf00f

展开二进制 1111 1111 0000 0000 0000 1111 1111

CAN控制器设置的接受ID是0xf00f

1111 1111 0000 0000 0000 1111 1111

这时如果我的屏蔽寄存器

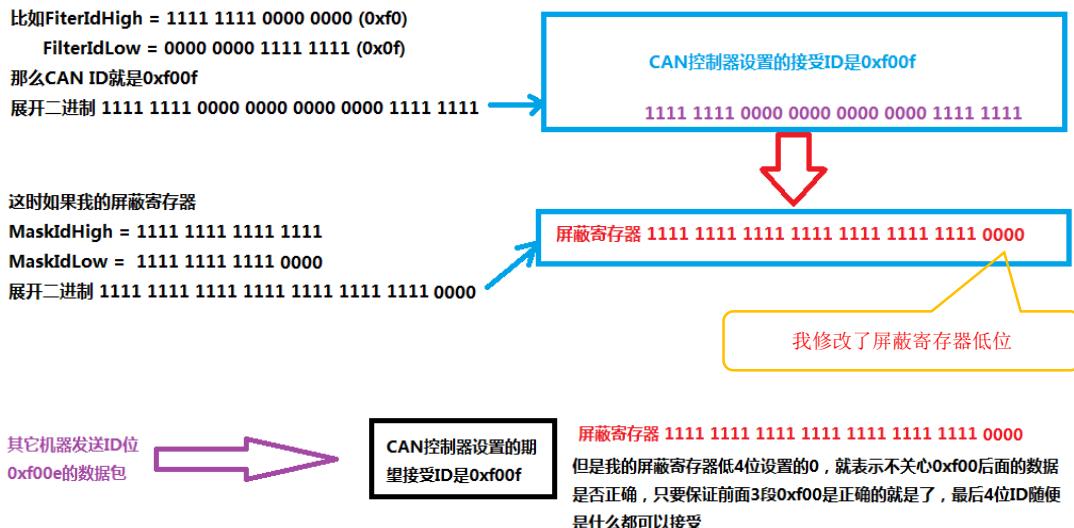
MaskIdHigh = 1111 1111 1111 1111

MaskIdLow = 1111 1111 1111 1111

展开二进制 1111 1111 1111 1111 1111 1111

屏蔽寄存器 1111 1111 1111 1111 1111 1111 1111

这种情况我本机收到的 CAN ID 必须是 0xf00f, 才能接受后面的数据。如果差 1 位不是, 比如接受的是 0xf00e 这种 ID, 那么也会抛弃数据



这就是 CAN 总线的屏蔽滤波 ID 功能

```

1 /*CAN接受中断配置*/
2 void CAN_NVIC_Config(void)
3 {
4     NVIC_InitTypeDef NVIC_InitStructure; // 定义CAN中断优先级结构体
5
6     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); // 中断分配到0组
7     NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn; // CAN1接受邮箱0中断
8     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // 抢占优先级为1
9     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1; // 响应优先级(子优先级)为1
0     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; // 中断优先级开启
1     NVIC_Init(&NVIC_InitStructure); // 初始化中断优先级
2
3
4 }

```

CAN 接受数据产生中断，配置中断优先级

```

#include "can.h"
CanTxMsg Can_Tx_data; // 定义Can发送数据包
CAN发送数据要定义发送数据包变量

int main(void)
{
    uint8_t box; // 判断邮箱是否发送完成

    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    CAN_init(); // 初始化CAN引脚
    CAN_Config(); // 初始化CAN波特率1M

    CAN_NVIC_Config(); // CAN中断优先级配置
    CAN_Filter(0); // 关闭CAN接受过滤器,任何CAN ID都接受,我默认填0

    printf("xxxxzzzz\r\n");
}

```

```

while(1)
{
    delay_ms(1000);
    Can_Tx_data.StdId = 0;//11位ID我们不用默认写0
    Can_Tx_data.ExtId = 0;//我们发送19位扩展ID
    Can_Tx_data.RTR = CAN_RTR_Data;//我们要发数据所以选择数据帧
    Can_Tx_data.IDE = CAN_Id_Extended;//发送的是扩展19位ID的数据
    Can_Tx_data.DLC = 8;//我们发送的是8个数据

    Can_Tx_data.Data[0] = 0xff;
    Can_Tx_data.Data[1] = 0xff;
    Can_Tx_data.Data[2] = 0xff;
    Can_Tx_data.Data[3] = 0xff;
    Can_Tx_data.Data[4] = 0x01;
    Can_Tx_data.Data[5] = 0x02;
    Can_Tx_data.Data[6] = 0x03;
    Can_Tx_data.Data[7] = 0x04;

    printf("CAN send 0xff 0xff 0xff 0x01 0x02 0x03 0x04\r\n");

    box = CAN_Transmit(CAN1,&Can_Tx_data);//把我们写好的数据发送到CAN总线
    while(CAN_TransmitStatus(CAN1,box) == CAN_TxStatus_Failed);//为了安全起见发送数据后检查邮箱是否发送完成
}

return 0;

```

这里就是想发送数据包填入要发送的数据

接受中断产生的数据也要定义接受数据包

CAN_Transmit 就是发送数据，然后还要检测发送是否成功

接受的数据就在接受数据包里面，取出来就是

我们现在用的是回环测试来测试代码是否正确

```

XXXXXXX
CAN send 0xff 0xff 0xff 0x01 0x02 0x03 0x04
Receive CAN ID = 0 data0 = ff data1 = ff data2 = ff data3 = ff data4 = 1 data5 = 2 data6 = 3 data7 = 4
CAN send 0xff 0xff 0xff 0x01 0x02 0x03 0x04
Receive CAN ID = 0 data0 = ff data1 = ff data2 = ff data3 = ff data4 = 1 data5 = 2 data6 = 3 data7 = 4
CAN send 0xff 0xff 0xff 0x01 0x02 0x03 0x04
Receive CAN ID = 0 data0 = ff data1 = ff data2 = ff data3 = ff data4 = 1 data5 = 2 data6 = 3 data7 = 4
CAN send 0xff 0xff 0xff 0x01 0x02 0x03 0x04
Receive CAN ID = 0 data0 = ff data1 = ff data2 = ff data3 = ff data4 = 1 data5 = 2 data6 = 3 data7 = 4

```

CAN 内部数据收发正常

现在修改为标准模式，波特率为 1M

```

CAN_InitTypeStruct.CAN_Mode = CAN_Mode_Normal;//设置CAN为标准收发模式
CAN_InitTypeStruct.CAN_NART = ENABLE;//支持报文重传
CAN_SetBaudRate(&CAN_InitStructure, CAN_BAUD_1M);

```

ID	Da0	Da1	Da2	Da3	Da4	Da5	Da6	Da7	Len	Fmt	Typ
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data
0x00000000	0xff	0xff	0xff	0x01	0x02	0x03	0x04	0x08	0x08		Extended Data

发送没有问题

选中	ID	Len	DATA0	DATA1	DATA2	DATA3	DATA4	DATA5	DATA6	DATA7	Format	Type	备注	延时
<input checked="" type="checkbox"/>	00000000	8	1	2	3	4	10	20	30	40	扩展帧	数据帧		0

```

Receive CAN ID = 0 data0 = 1 data1 = 2 data2 = 3 data3 = 4 data4 = 10 data5 = 20 data6 = 30 data7 = 40
Receive CAN ID = 0 data0 = 1 data1 = 2 data2 = 3 data3 = 4 data4 = 10 data5 = 20 data6 = 30 data7 = 40
Receive CAN ID = 0 data0 = 1 data1 = 2 data2 = 3 data3 = 4 data4 = 10 data5 = 20 data6 = 30 data7 = 40
CAN_send_0xffff_0xffff_0xffff_0x01_0x02_0x03_0x04

```

接受电脑发送过来的 ID 为 0 的数据没有问题

接受电脑发送过来的 ID 为 FF 的数据就会屏蔽掉，不接受。

这证明了屏蔽滤波寄存器起作用了，其它的 ID 自己测试，CAN 总线测试完成。

STM32 串口库函数操作

我们用 STM32F103 为例来做串口收发实验。

```
#include "stm32f10x.h"
#include "uart.h"

void uart_init(u32 bound)
{
    //GPIO端口设置
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1|RCC_APB2Periph_GPIOA, ENABLE); //使能USART1 GPIOA时钟

    //USART1_TX GPIOA.9
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9; //PA.9
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化GPIOA.9

    //USART1_RX GPIOA.10初始化
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10; //PA10
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; //浮空输入
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化GPIOA.10

    //USART1 NVIC 配置
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3; //抢占优先级3
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3; //子优先级3
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ通道使能
    NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化VIC寄存器

    //USART 初始化设置

    USART_InitStructureUSART_BaudRate = bound; //串口波特率
    USART_InitStructureUSART_WordLength = USART_WordLength_8b; //字长为8位数据格式
    USART_InitStructureUSART_StopBits = USART_StopBits_1; //一个停止位
    USART_InitStructureUSART_Parity = USART_Parity_No; //无奇偶校验位
    USART_InitStructureUSART_HardwareFlowControl = USART_HardwareFlowControl_None; //无硬件数据流控制
    USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx; //收发模式

    USART_Init(USART1, &USART_InitStructure); //初始化串口1
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); //开启串口接受中断
    USART_Cmd(USART1, ENABLE); //使能串口1
}
```

初始化串口
时钟和 GPIO

设置串口中
断优先级

初始化启
用串口

串口发送和接受程序

```
void USART1_IRQHandler(void)
{
    u8 Res;
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) //接收中断
    {
        Res = USART_ReceiveData(USART1); //读取接收到的数据
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
        USART_SendData(USART1, Res); //向串口1发送数据
        while(USART_GetFlagStatus(USART1, USART_FLAG_TC) != SET); //等待发送结束
    }
}
```



STM32 串口如何做到接受一串字符后再发送，而不是边发边收？

STM32 串口多字节接受缓存发送实验，我们以 CAN 总线的数据帧为例

```
unsigned char USART_buff[8];

u16 len=0;

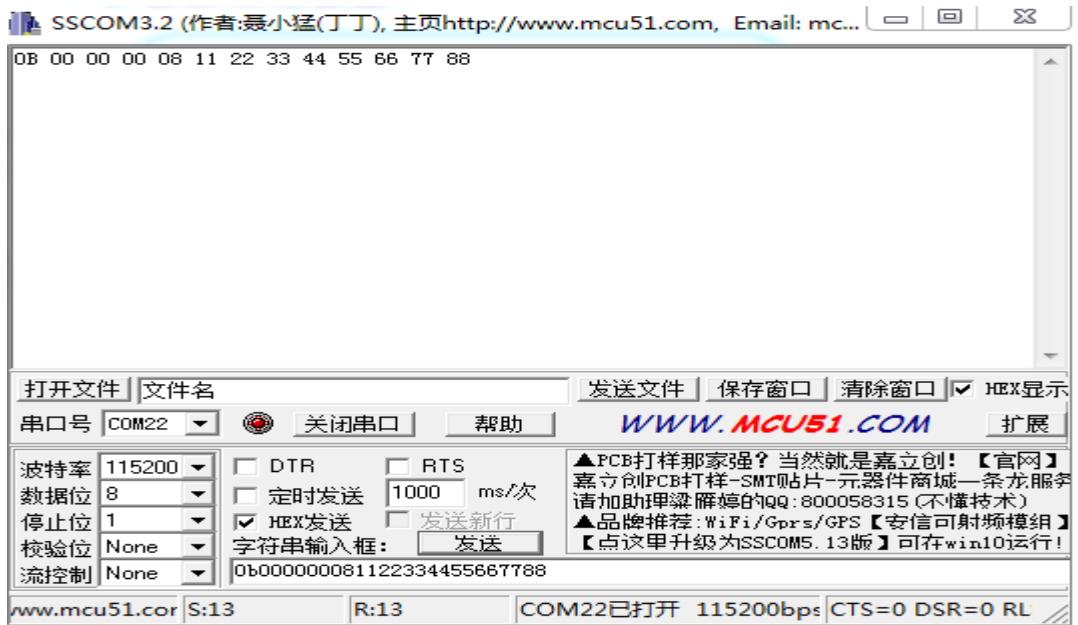
void USART1_IRQHandler(void)
{
    u8 Res;
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) //接收中断
    {
        Res =USART_ReceiveData(USART1); //读取接收到的数据
        USART_buff[len++]=Res;
    }
    USART_ClearITPendingBit(USART1, USART_IT_RXNE);
}
```

串口接收程序和上面区别不大，只是取消了在串口中断里面发送数据

```
4 while(1)
5 {
6     /*STM32接受串口数据 用CAN发送*/
7     if(len==13)
8     {
9         for(i=0;i<len;i++)
10            USART_SendData(USART1, USART_buff[i]); //向串口1发送数据
11            while(USART_GetFlagStatus(USART1, USART_FLAG_TC)!=SET); //等待发送结束
12        }
13        len=0;
14    }
15 }
```

这个 len 就决定了串口接收了多少字节数据之后在用发送程序一次发送

全局变量定义一个 USART_buff 和数据长度 len



既然程序定义了 13 个那么在串口助手里面也只能发送 13 个，如果串口后面增加了数据量也会跟着发送出去，不会被清空，自己试试。

STM32+MCP2515 CAN 控制器测试

该方法也适用于各种没有 can 接口的 51 单片机，现在只完成了 CAN 的基本收发，至于远程帧和滤波还没有实现，但是硬件测试已经完成了。

CAN 总线初始化

```
void myCAN_INIT()
{
    unsigned char temp=0;
    MCP2515_Reset(); //发送复位指令软件复位MCP2515
    delay_ms(1); //通过软件延时约1ms(不准确)

    CAN_set_var(0x0f,0xe0,0x80);//设置为配置模式
    //设置波特率为125Kbps
    //set CNF1,SJW=00,长度为1TQ,BRP=49,TQ=[2*(BRP+1)]/Fsoc=2*50/8M=12.5us
    MCP2515_WriteByte(CNF1,CAN_125Kbps);
    //set CNF2,SAM=0,在采样点对总线进行一次采样, PHSEG1=(2+1)TQ=3TQ, PRSEG=(0+1)TQ=1TQ
    MCP2515_WriteByte(CNF2,0x80|PHSEG1_3TQ|PRSEG_1TQ);
    //set CNF3,PHSEG2=(2+1)TQ=3TQ,同时当CANCTRL.CLKEN=1时设定CLKOUT引脚为时间输出使能位
    MCP2515_WriteByte(CNF3,PHSEG2_3TQ);
    MCP2515_WriteByte(0x60,0x60); //关闭接受所有数据
    //滤波
    MCP2515_WriteByte(0x00,0);
    MCP2515_WriteByte(0x01,0);
    MCP2515_WriteByte(0x02,0);
    MCP2515_WriteByte(0x03,0);
    //屏蔽
    MCP2515_WriteByte(0x20,0);
    MCP2515_WriteByte(0x21,0);
    MCP2515_WriteByte(0x22,0);
    MCP2515_WriteByte(0x23,0);

    MCP2515_WriteByte(0x2b,0x01); //接受数据产生中断
    //CAN_set_var(0x0f,0xe0,0x40); //回环模式
    CAN_set_var(0x0f,0xe0,0x00); //工作模式
}
```

CAN_set_var(0x0f,0xe0,0x80); //设置为配置模式

MCP2515 有五种工作模式，分别为：

1. 配置模式。
2. 正常模式。
3. 休眠模式。
4. 仅监听模式。
5. 环回模式。

我们一般
都用这三
种模式

```

void CAN_set_var(u8 address,u8 mask,u8 dat)
{
    MCP2515_CS=1;
    delay_us(1);
    MCP2515_CS=0;
    SPI_SendByte(0x05); //用户可以修改寄存器的位，但不是所有寄存器都可以位修改
    SPI_SendByte(address);
    SPI_SendByte(mask);
    SPI_SendByte(dat);
    MCP2515_CS=1;
}

```

因为 CANCTRL 寄存器不能像其它寄存器那样直接写值，如果你想修改 CANCTRL 寄存器的某一位，你必须先用 mask 将那一位的门打开，然后才能写值进去

表 12-1: SPI 指令集

指令名称	指令格式	说明
复位	1100 0000	将内部寄存器复位为缺省状态，并将器件设定为配置模式。
读	0000 0011	从指定地址起始的寄存器读取数据。
读 RX 缓冲器	1001 0nm0	读取接收缓冲器时，在“n,m”所指示的四个地址中的一个放置地址指针可以减轻一般读命令的开销。注：在拉升 CS 引脚为高电平后，相关的 RX 标志位（CANINTF.RXnIF）将被清零。
写	0000 0010	将数据写入指定地址起始的寄存器。
装载 TX 缓冲器	0100 0abc	装载发送缓冲器时，在“a,b,c”所指示的六个地址中的一个放置地址指针可以减轻一般写命令的开销。
RTS (请求发送报文)	1000 0nnn	指示控制器开始发送任一发送缓冲器中的报文发送序列。 1000 0nnn TXB2 请求发送 ↑ TXBO 请求发送 TXB1 请求发送
读状态	1010 0000	快速查询命令，可读取有关发送和接收功能的一些状态位。
RX 状态	1011 0000	快速查询命令，确定匹配的滤波器和接收报文的类型（标准帧、扩展帧和 / 或远程帧）。
位修改	0000 0101	允许用户将特殊寄存器中的单独位置 1 或清零。注：该命令并非适用于所有的寄存器。对不允许位修改操作的寄存器执行该命令会将屏蔽字节强行设为 FFh。请参见第 11.0 节“寄存器映射表”中的寄存器映射表，以了解适用的寄存器。

CANCTRL——CAN 控制寄存器 (地址: XFh)

R/W-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-1	R/W-1
REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0

bit 7

bit 0

我想修改 CANCTRL 位配置模式

bit 7-5

REQOP<2:0>: 请求工作模式的位

- 000 = 设定为正常工作模式
- 001 = 设定为休眠模式
- 010 = 设定为环回模式
- 011 = 设定为仅监听模式
- 100 = 设定为配置模式

先用 mask 将这三位写成 111，这样就把位 7, 6, 5 的门打开了，然后将想要的值写入位 7, 6, 5

REQOP 位不应设置为其他值，因为这些值都是无效的。

CAN_set_var(0x0f,0xe0,0x80); 0x0f 是地址

0xe0 = 1110 0000 现将 CANCTRL 前三位 mask 设置成 1

0x80 = 1000 0000 然后将 0x80 数据写进去后面三位因为 mask 是 0 所以写什么都没有用

```

//设置波特率为125Kbps
//set CNF1, SJW=00, 长度为1TQ, BRP=49, TQ=[2*(BRP-1)+1]
MCP2515_WriteByte(CNF1,CAN_125Kbps);
//set CNF2, SAM=0, 在采样点对总线进行一次采样,
MCP2515_WriteByte(CNF2,0x80|PHSEG1_3TQ|PRSEG);
//set CNF3, PHSEG2=(2+1)TQ=3TQ, 同时当CANCTRL.
MCP2515_WriteByte(CNF3,PHSEG2_3TQ);

```

设置波特率按照网上的规范设置

设置 CAN 总线接受数据的屏蔽滤波，CAN 屏蔽滤波原来请看我 STM8 操作手册

MCP2515_WriteByte(0x60,0x60); //设置屏蔽滤波设置屏蔽滤波之前要关掉 CAN 的接受功能

RXB0CTRL——接收缓冲器 0 控制寄存器（地址：60h）

U-0	R/W-0	R/W-0	U-0	R-0	R/W-0	R-0	R-0
—	RXM1	RXM0	—	RXRTR	BUKT	BUKT1	FILHITO

bit 7 未用：读为 0
bit 6-5 RXM：接收缓冲器工作模式位

11 = 关闭屏蔽 / 滤波功能：接收所有报文
10 = 只接收符合滤波器条件的带有扩展标识符的有效报文
01 = 只接收符合滤波器条件的带有标准标识符的有效报文
00 = 接收符合滤波器条件的所有带扩展标识符或标准标识符的有效报文

// 滤波

```
MCP2515_WriteByte(0x00, 0);  
MCP2515_WriteByte(0x01, 0);  
MCP2515_WriteByte(0x02, 0);  
MCP2515_WriteByte(0x03, 0);
```

// 屏蔽

```
MCP2515_WriteByte(0x20, 0);  
MCP2515_WriteByte(0x21, 0);  
MCP2515_WriteByte(0x22, 0);  
MCP2515_WriteByte(0x23, 0);
```

我们将 8 个滤波屏蔽寄存器都设置成 0，这样所有 ID 都能接受

MCP2515_WriteByte(0x2b,0x01); //我们运行 CAN 接受到报文产生中断我们去中断里取数据

CANINTE——中断使能寄存器（地址：2Bh）

| R/W-0 |
|-------|-------|-------|-------|-------|-------|-------|--------------|
| MERRE | WAKIE | ERRIE | TX2IE | TX1IE | TX0IE | RX1IE | RX0IE |

bit 7 bit 0

bit 0 RX0IE：接收缓冲器 0 满中断使能位
1 = RXB0 装载报文时中断
0 = 禁止

CAN_set_var(0x0f,0xe0,0x00); //CAN 控制器从配置模式进入正常工作模式

0-1: CANCTRL——CAN 控制寄存器（地址：XFh）

R/W-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-1	R/W-1
REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0

bit 7 7 bit 0

bit 7-5 REQOP<2:0>：请求工作模式的位
000 = 设定为正常工作模式
001 = 设定为休眠模式
010 = 设定为环回模式
011 = 设定为仅监听模式
100 = 设定为配置模式
REQOP 位不应设置为其他值，因为这些值都是无效的。

CAN 初始化之后我们就要用到前面的串口来双向传输报文了。这些屏蔽滤波功能都为 0，表示没有用。

初始化 STM32 要模拟 SPI 总线的 IO 口，和接受 CAN 报文中断的 IO 口

```
void SPI2_GPIO_init() //PB12,PB13,PB14,PB15初始化模拟SPI口
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15|GPIO_Pin_13|GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //IO口速度为50MHz
    GPIO_Init(GPIOB, &GPIO_InitStructure); //根据设定参数初始化PB15,PB12,PB13输出

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14;//PB14
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //上拉输入
    GPIO_Init(GPIOB, &GPIO_InitStructure);

}
```

中断口初始化

```
void SPI2_INT_init()
{
    GPIO_InitTypeDef GPIO_InitStructure;//GPIO结构体
    NVIC_InitTypeDef NVIC_InitStructure;//中断优先级结构体
    EXTI_InitTypeDef EXTI_InitStructure;//中断使能结构体

    /* Enable INTERRUPT GPIOB clocks */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); //打开中断复用功能

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;//PB11 该端口是来接受MCP2515的CAN接受报文中
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //上拉输入
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /*设置引脚中断功能*/
    EXTI_ClearITPendingBit(EXTI_Line1);
    EXTI_InitStructure.EXTI_Line = EXTI_Line1;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource11);
    /*设置中断优先级*/
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    NVIC_InitStructure.NVIC IRQChannel = EXTI15_10_IRQn;
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

主函数初始化

```
int main(void)
{
    u8 i; //串口接收数据大小自加变量
    unsigned char dly;
    u16 times=0;
    delay_init(); //延时函数初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置NVIC中断分组2:2位抢占优先级, 2位响应优先级
    uart_init(115200); //串口初始化为115200

    SPI2_GPIO_init(); //PB12,PB13,PB14,PB15初始化模拟SPI口
    myCAN_INIT(); //MCP2515寄存器初始化
    SPI2_INT_init(); //初始化MCP2515接受报文中断口
}
```

下面就是电脑串口发送数据->STM32->MCP2515->CAN->电脑 CAN 软件显示接受报文

```
while(1)
{
    /*STM32接受串口数据*/
    if(len==13)
    {
        for(i=0;i<len;i++)
        {
            USART_SendData(USART1, USART_buff[i]); //向串口1发送数据
            while(USART_GetFlagStatus(USART1, USART_FLAG_TC) !=SET); //等待发送结束
        }

        for(i=1;i<len;i++) { //将数据放入CAN缓冲区
            MCP2515_WriteByte(0x30+i, USART_buff[i]);
        }
        CAN_set_var(0x30, USART_buff[0], USART_buff[0]); //从CAN向外发送数据
        do
        {
            i=MCP2515_ReadByte(0x30);
            i= i&0x08; //观察报文是否发送完
        }while(i);

        len=0;
    }
}
```

TXBnSIDH——发送缓冲器 n 标准标识符高位（地址：31h, 41h, 51h）

| R/W-x |
|-------|-------|-------|-------|-------|-------|-------|-------|
| SID10 | SID9 | SID8 | SID7 | SID6 | SID5 | SID4 | SID3 |

bit 7 → bit 0

-4: TXBnSIDL——发送缓冲器 n 标准标识符低位（地址：32h, 42h, 52h）

| R/W-x |
|-------|-------|-------|-------|-------|-------|-------|-------|
| SID2 | SID1 | SID0 | — | EXIDE | — | EID17 | EID16 |

bit 7 → bit 0

bit 7-5 **SID**: 标准标识符 <2:0>

bit 4 未用: 读为 0

bit 3 **EXIDE**: 扩展标识符使能位

1 = 报文将发送扩展标识符

0 = 报文将发送标准标识符

TXBnEID8——发送缓冲器 n 扩展标识符高位（地址：33h, 43h, 53h）

| R/W-x |
|-------|-------|-------|-------|-------|-------|-------|-------|
| EID15 | EID14 | EID13 | EID12 | EID11 | EID10 | EID9 | EID8 |

bit 7 → bit 0

TXBnEID0——发送缓冲器 n 扩展标识符低位（地址：34h, 44h, 54h）

| R/W-x |
|-------|-------|-------|-------|-------|-------|-------|-------|
| EID7 | EID6 | EID5 | EID4 | EID3 | EID2 | EID1 | EID0 |

bit 7 → bit 0

7: TXBnDLC——发送缓冲器 n 数据长度码 (地址: 35h, 45h, 55h)

| R/W-x |
|-------|-------|-------|-------|-------|-------|-------|-------|
| — | RTR | — | — | DLC3 | DLC2 | DLC1 | DLC0 |

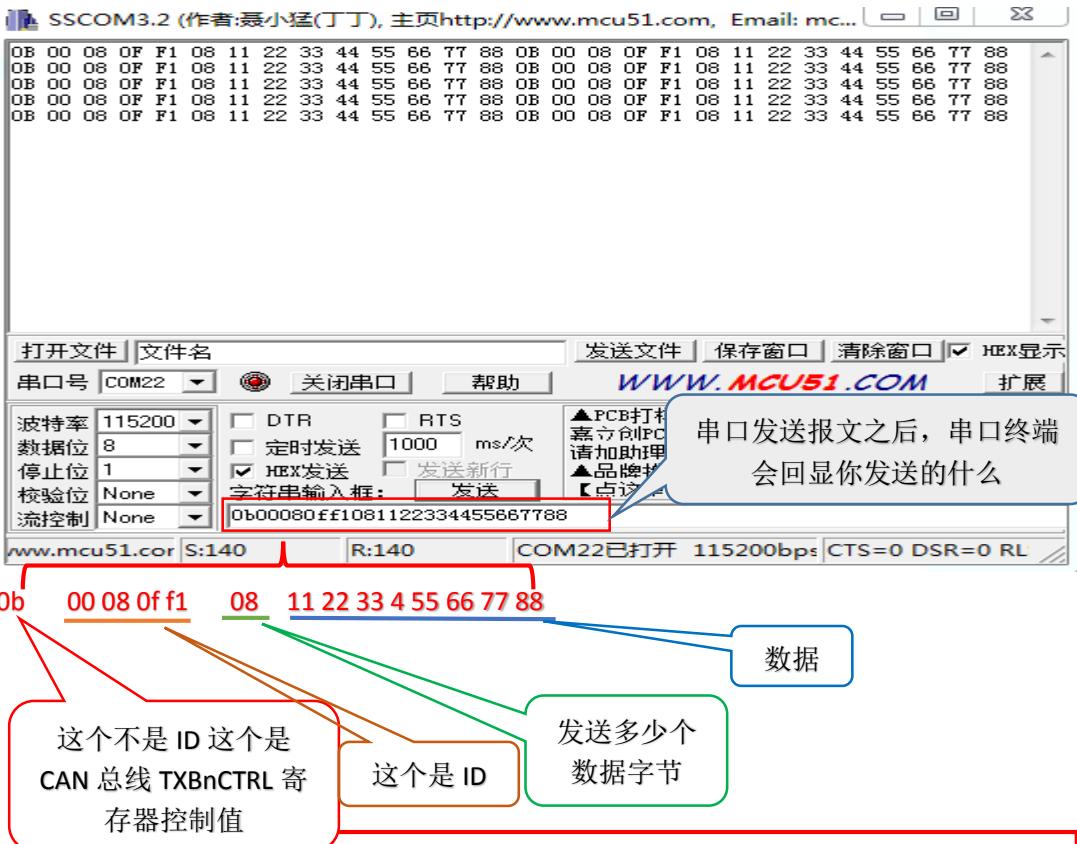
bit 7 bit 0

- bit 7 未用: 读为 0
 bit 6 **RTR:** 远程发送请求位
 1 = 发送的报文为远程发送请求
 0 = 发送的报文为数据帧
 bit 5-4 未用: 读为 0
 bit 3-0 **DLC:** 数据长度码位 <3:0>
 设定要发送的数据长度 (0 到 8 字节)
 注: 可以将 DLC 设定为大于 8 的值, 但只发送 8 个字节。

TXBnDm——发送缓冲器 n 数据字节 m (地址: 36h - 3Dh, 46h - 4Dh, 56h - 5Dh)

| R/W-x |
|----------|----------|----------|----------|----------|----------|----------|----------|
| TXBnDm 7 | TXBnDm 6 | TXBnDm 5 | TXBnDm 4 | TXBnDm 3 | TXBnDm 2 | TXBnDm 1 | TXBnDm 0 |

bit 7 bit 0



I: TXBnCTRL——发送缓冲器 n 控制寄存器 (地址: 30h, 40h, 50h)

U-0	R-0	R-0	R-0	R/W-0	U-0	R/W-0	R/W-0
—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0

bit 7 bit 0

CAN_set_var(0x30, USART_buff[0], USART_buff[0]); //我们将控制值写在数组【0】这里然后执行, 这句话执行后数组【1~14】里面的 ID 和数据就全部发送出去了。

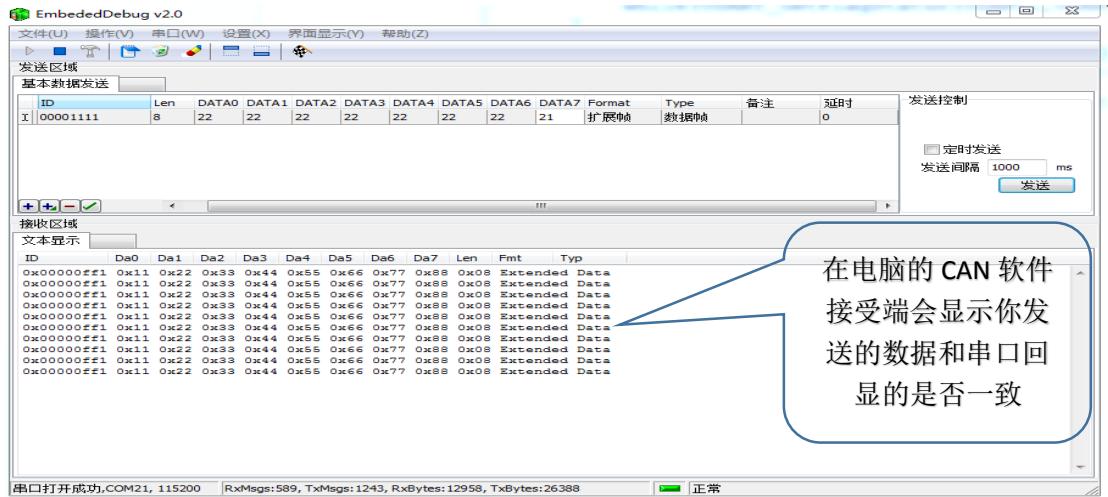
bit 3	TXREQ: 报文发送请求位
	1 = 缓冲器等待报文发送 (MCU 将此位置 1 以请求报文发送—报文发送后该位自动清零)
	0 = 缓冲器无等待发送报文 (MCU 将此位清零以请求中止报文发送)
bit 2	未用：读为 0
bit 1-0	TXP: 发送缓冲器优先级 <1:0> 位
	11 = 最高的报文发送优先级
	10 = 中偏高的报文发送优先级
	11 = 中偏低的报文发送优先级
	00 = 最低的报文发送优先级

报文发送程序执行后，我们还要确认 CAN 芯片报文是否发送完了

```
do
{
    i=MCP2515_ReadByte(0x30);
    i= i&0x08; //观察报文是否发送完
}while(i);
```

如果报文发送完了，就可以执行下一次循环再发送新的报文。

CAN 报文接受



下面就是电脑 CAN 发送数据->CAN 调试器->MCP2515->STM32->串口打印接受报文

```

void EXTI15_10_IRQHandler(void)
{
    u8 sta_int, z;

    if(EXTI_GetITStatus(EXTI_Line11) != RESET)
    {
        sta_int=MCP2515_ReadByte(0x2c);
        MCP2515_WriteByte(0x2c,0x00);
        if(sta_int&0x01)
        {
            for(z=0;z<14;z++)
            {
                USART_SendData(USART1, MCP2515_ReadByte(0x60+z));
                while(USART_GetFlagStatus(USART1,USART_FLAG_TC)!=SET); //等待发送结束
            }
        }

        EXTI_ClearFlag(EXTI_Line11);
        EXTI_ClearITPendingBit(EXTI_Line11); //清楚中断标志位
    }
}

```

我们用 STM32 中断来接受电脑 CAN 控制器发过来的报文



bit 0

RX0IF: 接收缓冲器 0 满中断标志位

- 1 = 有等待处理的中断 (必须由 MCU 清零才可使中断复位)
- 0 = 无等待处理的中断

```

for(z=0;z<14;z++)
{
    USART_SendData(USART1, MCP2515_ReadByte(0x60+z));
    while(USART_GetFlagStatus(USART1, USART_FLAG_TC)!=SET); //等待发送结束
}

```

这是接受 ID 和数据的程序

RXBnSIDH——接收缓冲器 n 标准标识符高位（地址：61h, 71h）

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7				bit 0			

-5: RXBnSIDL——接收缓冲器 n 标准标识符低位（地址：62h, 72h）

R-x	R-x	R-x	R-x	R-x	U-0	R-x	R-x
SID2	SID1	SID0	SRR	IDE	—	EID17	EID16
bit 7				bit 0			

bit 7-5 **SID:** 标准标识符位 <2:0>

这些位装载接收报文中标准标识符的最低 3 位

bit 4 **SRR:** 远程发送请求位（只有当 IDE 位 = 0 时有效）

1 = 收到标准远程发送请求帧

0 = 收到标准数据帧

bit 3 **IDE:** 扩展标识符标志位

该位表明收到的报文是标准帧还是扩展帧

1 = 收到的报文是扩展帧

0 = 收到的报文是标准帧

bit 2 未用：读为 0

bit 1-0 **EID:** 扩展标识符位 <17:16>

这些位装载接收报文中扩展标识符的最高 2 位

RXBnEID8——接收缓冲器 n 扩展标识符高位（地址：63h, 73h）

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7				bit 0			

RXBnEID0——接收缓冲器 n 扩展标识符低位（地址：64h, 74h）

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
EID7	EID6	EID5	EID4	EID3	EID2	EID1	EID0
bit 7				bit 0			

4-8: RXBnDLC——接收缓冲器 n 数据长度码（地址：65h, 75h）

U-0	R-x	R-x	R-x	R-x	R-x	R-x	R-x
—	RTR	RB1	RB0	DLC3	DLC2	DLC1	DLC0
bit 7				bit 0			

bit 7 未用：读为 0

bit 6 **RTR:** 扩展帧远程发送请求位
(只有当 RXBnSIDL.IDE = 1 时有效)

1 = 接收到扩展远程（发送请求）帧

0 = 接收到扩展数据帧

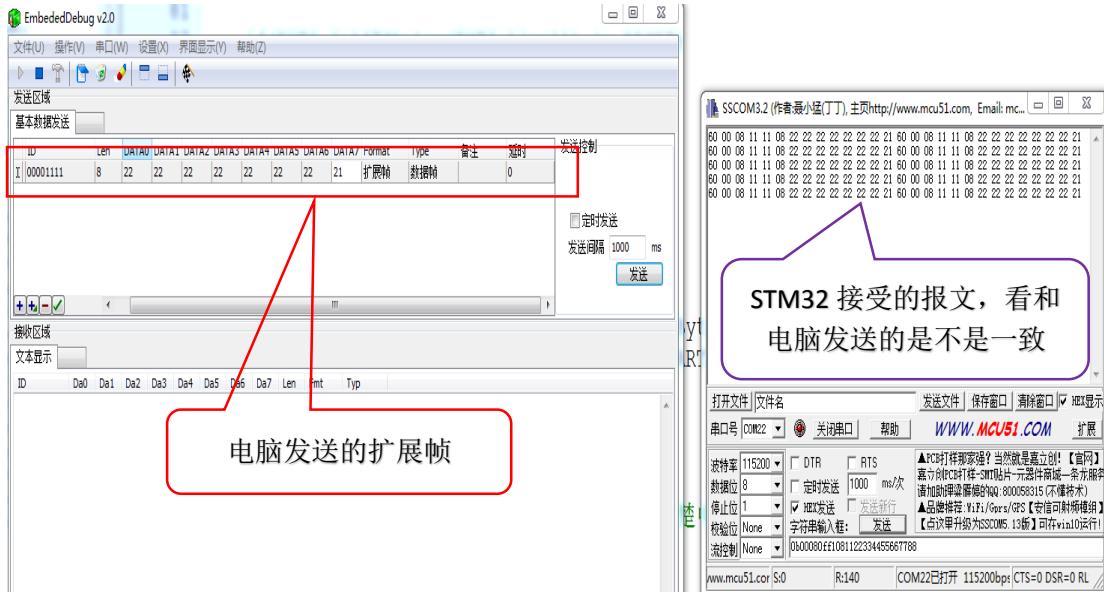
bit 5 **RB1:** 保留位 1

bit 4 **RB0:** 保留位 0

bit 3-0 **DLC:** 数据长度码位 <3:0>
表明接收到的数据字节个数

RXBnDM——接收缓冲器 n 数据字段字节 M (地址：66h - 6Dh, 76h - 7Dh)

R-x							
RBnDm7	RBnDm6	RBnDm5	RBnDm4	RBnDm3	RBnDm2	RBnDm1	RBnDm0
bit 7				bit 0			



根据以上测试，CAN 总线的硬件收发和 SPI 数据协议是完全没有问题了，剩下的就是远程帧和接受滤波调试了。

CAN 接受滤波测试

```

MCP2515_WriteByte(0x60, 0x00); // 打开屏蔽滤波功能
// 滤波
MCP2515_WriteByte(0x00, 0);
MCP2515_WriteByte(0x01, 0x08);
MCP2515_WriteByte(0x02, 0);
MCP2515_WriteByte(0x03, 0xff);
// 屏蔽
MCP2515_WriteByte(0x20, 0);
MCP2515_WriteByte(0x21, 0);
MCP2515_WriteByte(0x22, 0);
MCP2515_WriteByte(0x23, 0xff);

MCP2515_WriteByte(0x2b, 0x01); // 接受数据产生中断

```

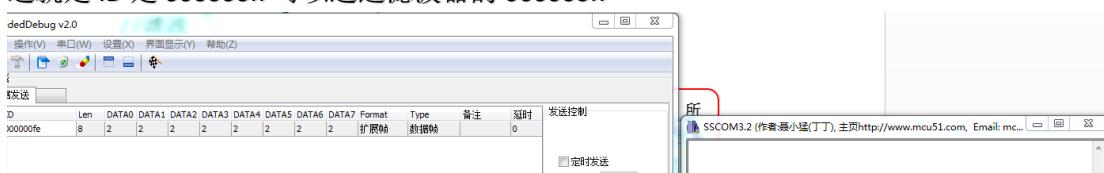
因为我们是扩展 ID，所以这里有一位要填写 1

这里是我们设置的要接受的 ID

这里是我们设置接受的 ID 是完全正确通过，还是一部分正确就通过



这就是 ID 是 000000ff 可以通过滤波器的 000000ff



这就是 ID 000000fe 无法通过滤波器 000000ff

因为根据寄存器排布上面的滤波器只能做到低 16 位，看起比较直观。

我现在将屏蔽滤波代码和波特率封装一下看能不能做到 29 位

```
/*该封装接口CAN设备ID可寻址范围000000~FFFFFF*/
void myCAN_INIT_package(unsigned long ID,unsigned int buadrate) //屏蔽滤波接口封装
{
    unsigned char temp=0;
    typedef unsigned char u8;
    u8 FxR1,FxR2,FxR3,FxR4,tmp1,tmp2,bound;
    MCP2515_Reset(); //发送复位指令软件复位MCP2515
    delay_ms(1); //通过软件延时约nms(不准确)

    /*比特率封装*/
    switch(buadrate)
    {
        case 10000:bound = CAN_10Kbps;break;
        case 25000:bound = CAN_25Kbps;break;
        case 50000:bound = CAN_50Kbps;break;
        case 100000:bound = CAN_100Kbps;break;
        case 125000:bound = CAN_125Kbps;break;
        case 250000:bound = CAN_250Kbps;break;
        case 500000:bound = CAN_500Kbps;break;
        default:
            break;
    }

    CAN_set_var(0x0f,0xe0,0x80); //设置为配置模式
    //设置波特率为125Kbps
    //set CNF1,SJW=0, 长度为1TQ,BRP=49,TQ=[2*(BRP+1)]/Fsoc=2*50/8M=12.5us
    MCP2515_WriteByte(CNF1,bound);
    //set CNF2,SAM=0,在采样点对总线进行一次采样, PHSEG1=(2+1)TQ=3TQ, PRSEG=(0+1)TQ=1TQ
    MCP2515_WriteByte(CNF2,0x80|PHSEG1_3TQ|PRSEG_1TQ);
    //set CNF3,PHSEG2=(2+1)TQ=3TQ,同时当CANCTRL.CLKEN=1时设定CLKOUT引脚为时间输出使能位
    MCP2515_WriteByte(CNF3,PHSEG2_3TQ);
    //MCP2515_WriteByte(0x60,0x60); //关闭接受所有数据
    MCP2515_WriteByte(0x60,0x00); //打开屏蔽滤波功能
    /*ID封装计算*/
    FxR1 = (u8) (((ID>>16)>>5) & 0X07);
    tmp1 = (u8) ((ID>>16)<<3);
    tmp1 = tmp1&0xe0;
    tmp2 = (u8) ((ID>>16)<<3);
    tmp2 = tmp2>>3;
    tmp2 = tmp2&0x03;
    tmp2 = tmp2|0x08;
    FxR2 = tmp1|tmp2;
    FxR3 = (u8) (ID>>8);
    FxR4 = (u8) ID;

    //滤波 想要接受的ID
    MCP2515_WriteByte(0x00,FxR1);
    MCP2515_WriteByte(0x01,FxR2);
    MCP2515_WriteByte(0x02,FxR3);
    MCP2515_WriteByte(0x03,FxR4);
    //屏蔽 到底ID正确多少算合格
    MCP2515_WriteByte(0x20,0xff);
    MCP2515_WriteByte(0x21,0xe3);
    MCP2515_WriteByte(0x22,0xff);
    MCP2515_WriteByte(0x23,0xff);

    MCP2515_WriteByte(0x2b,0x01); //接受数据产生中断
    //CAN_set_var(0x0f,0xe0,0x40); //回环模式
    CAN_set_var(0x0f,0xe0,0x00); //工作模式
}
```

这里我们屏蔽寄存器都设置成 1，这样 CAN 接受的 ID 必须和我滤波的 ID 完全对应才能接受数据

我们设置只接受 FF00FF 的 ID，波特率从最先的 125K 改成 500K

```
myCAN_INIT_package(0xff00ff, 500000); //MCP2515寄存器初始化+屏蔽滤接口封装
```



接受 ID 为 00 FF 00 FF

串口打印 ID 为 00 07 EB 00 FF 08 02 02 02 02 02 02 02

为什么这里多了个 00，
这是因为我们是从数组
0 开始取的数

我们接受的是 ff00ff 这里的
ID 怎么是 07EB00FF

```
for (z=0; z<14; z++)
{
    USART_SendData(USART1, MCP2515_ReadByte(0x60+z));
    while (USART_GetFlagStatus(USART1, USART_FLAG_TC) != SET); //等待发送结束
}
```

数组 0 是接受控制寄存器，其实这里可以不用取，没有什么意义，从数组 1 开始取就行了

RXB0CTRL——接收缓冲器 0 控制寄存器（地址：60h）

U-0	R/W-0	R/W-0	U-0	R-0	R/W-0	R-0	R-0
—	RXM1	RXM0	—	RXRTR	BUKT	BUKT1	FILHIT0
bit 7							bit 0

我们 ff00ff 的 ID 第一个 FF 是分开放
在两个寄存器上面的

RXBnSIDH——接收缓冲器 n 标准标识符高位（地址：61h, 71h）

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7	0	0	0	0	1	1	bit 0

图 4-5：RXBnSIDL——接收缓冲器 n 标准标识符低位（地址：62h, 72h）

R-x	R-x	R-x	R-x	R-x	U-0	R-x	R-x
SID2	SID1	SID0	SRR	IDE	—	EID17	EID16
bit 7	0	0	0	0	bit 0		

SID: 标准标识符位 <2:0>

这些位装载接收报文中标准标识符的最低 3 位

SRR: 远程发送请求位（只有当 IDE 位 = 0 时有效）

1 = 收到标准远程发送请求帧

0 = 收到标准数据帧

IDE: 扩展标识符标志位

该位表明收到的报文是标准帧还是扩展帧

1 = 收到的报文是扩展帧

0 = 收到的报文是标准帧

未用: 读为 0

EID: 扩展标识符位 <17:16>

我们将这三段取出来 1111 1111，也
就是 ff 了

我们把 ID = 07 EB 00 FF 前面的 07 EB 移动一下位，就成了标准的 FFOOFF ID 了

我们对发送数据包也进行了函数封装

```
/*CAN总线封包发送*/
/*ID范围 0000000~FFFFFFF
data 数据位8个字节
IDcfg 输入1为扩展ID最大29位
IDcfg 输入0为标识符ID最大16位
frame 输入1为远程帧，没有数据发送，只有ID
frame 输入0为数据帧，有ID和数据发送
*/
void myCAN_send_package(unsigned long ID ,unsigned char *data,unsigned char IDcfg,unsigned char frame)
{
    typedef unsigned char u8;
    u8 tmp1,tmp2,addr31h,addr32h,addr33h,addr34h,i;
    tmp1 = (u8)(ID>>16);
    tmp2 = ((tmp1<<3)>>3);
    addr32h = tmp2&0x03;
    addr32h |= ((tmp2<<3)&0xe0);
    if(IDcfg==1)
        addr32h = addr32h | 0x08; //扩展ID
    else
        addr32h = addr32h;

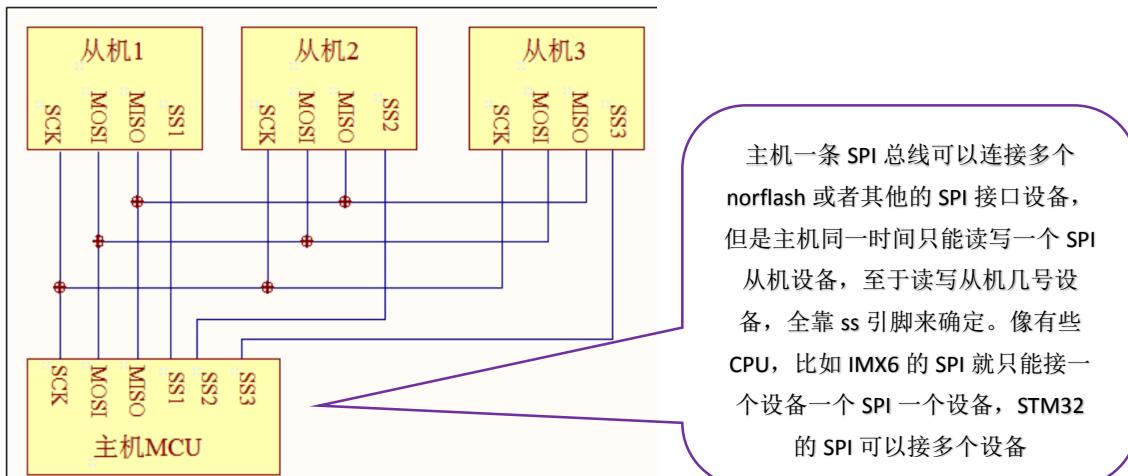
    addr31h = (u8)((ID>>16)>>5);
    addr33h = (u8)(ID>>8);
    addr34h = (u8)ID;

    MCP2515_WriteByte(0x31,addr31h);
    MCP2515_WriteByte(0x32,addr32h);
    MCP2515_WriteByte(0x33,addr33h);
    MCP2515_WriteByte(0x34,addr34h);

    if(frame==1)
        MCP2515_WriteByte(0x35,0x08|0x40);
    else
        MCP2515_WriteByte(0x35,0x08);
    MCP2515_WriteByte(0x36,data[0]);
    MCP2515_WriteByte(0x37,data[1]);
    MCP2515_WriteByte(0x38,data[2]);
    MCP2515_WriteByte(0x39,data[3]);
    MCP2515_WriteByte(0x3A,data[4]);
    MCP2515_WriteByte(0x3B,data[5]);
    MCP2515_WriteByte(0x3C,data[6]);
    MCP2515_WriteByte(0x3D,data[7]);
    CAN_set_var(0x30,0X0b,0x0b); //从CAN向外发送数据
    do
    {
        i=MCP2515_ReadByte(0x30);
        i= i&0x08; //观察报文是否发送完
    }while(i);
}
```

STM32 SPI 接口 读写 norflash 案例

SPI 通讯设备之间的常用连接方式见图 25-1。



主机一条 SPI 总线可以连接多个 norflash 或其他的 SPI 接口设备，但是主机同一时间只能读写一个 SPI 从机设备，至于读写从机几号设备，全靠 ss 引脚来确定。像有些 CPU，比如 IMX6 的 SPI 就只能接一个设备一个 SPI 一个设备，STM32 的 SPI 可以接多个设备

SPI 是全双工通信，SPI 传输速率本身是不受限制的，但是根据 CPU 不同，所以有支持 45M 传输速率的也有支持 25M 的，STM32F103 平台 SPI 速率是最高 36M，F429 平台上 45M。

引脚定义：

SS 引脚线:低电平为开始传输信号，高电平为结束传输信号

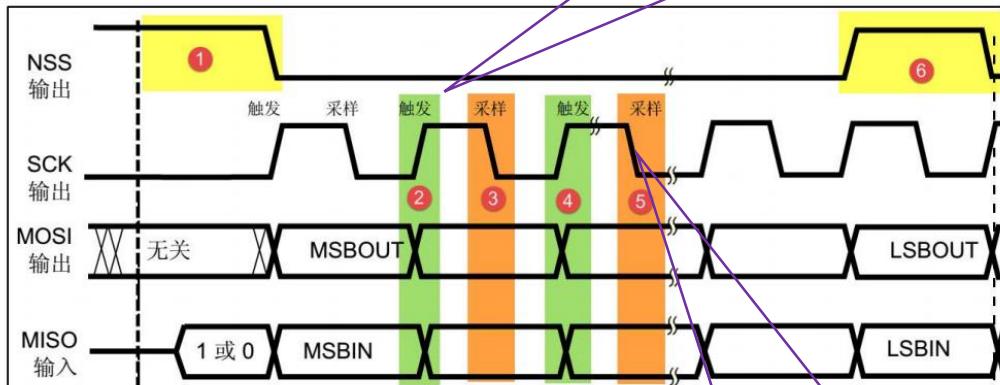
SCK 引脚线:SPI 时钟线，每个上升沿或者下降沿来读取数据 SCK 时钟速率= $PCLK$ 总线速率/2

MOSI 引脚线:主设备输出数据，从设备接受数据

MISO 引脚线:从设备输出数据，主设备接受数据

1. SPI 基本通讯过程

先看看 SPI 通讯的通讯时序，见图 25-2。

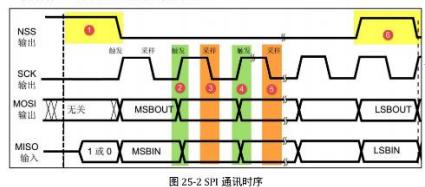


1. 触发状态数据不稳定，MCU 主机和从机不读取数据，

2. 只有在时钟下降沿主机和从机才读取数据

1. SPI 基本通讯过程

先看看 SPI 通讯的通讯时序，见图 25-2。

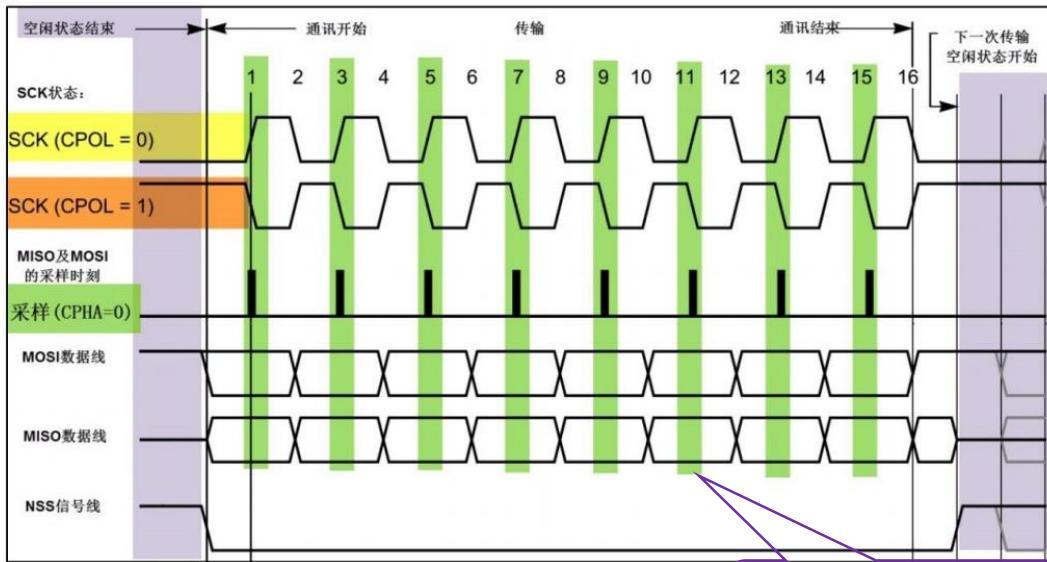


3. 八个时钟为一个字节数据，八个时钟后 NSS 引脚拉高，代表这一次数据传输完成

STM32F103 支持每次 8 个时钟 1 字节数据传输和每次

16 个时钟 2 字节数据传输

但是 STM32 的 SPI 有个特点，它的时钟可以是上升沿触发也可以是下降沿触发



CPOL=0 或者 1 是代表在 NSS 信号线一直处于高电平的时候，SCK 线的信号是一直处于高电平还是低电平

图 25-3 CPHA=0 时的 SPI 通讯模式

CPHA=0 时，时钟上升沿触发，那么主机和从机只能时钟下降沿采集 mosi/miso 的信号电平

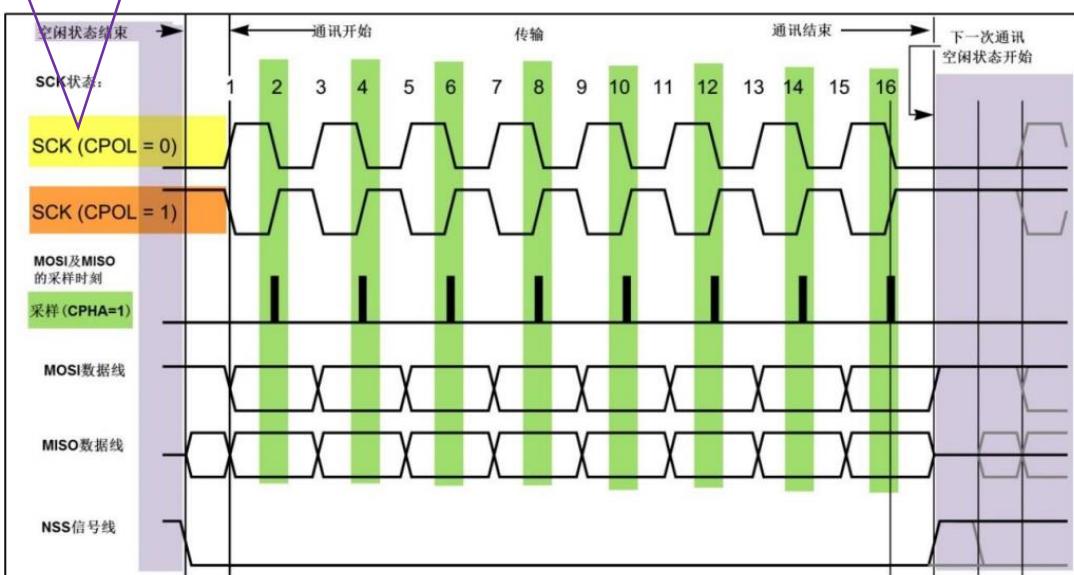


图 25-4 CPHA=1 时的 SPI 通讯模式

CPHA=1 那么就是下降沿触发，主机和从机上升沿采集 mosi/miso 电平

进入 STM32f10x_spi.h 来熟悉代码

```

0  typedef struct
1  {
2      uint16_t SPI_Direction;
3
4      uint16_t SPI_Mode;
5
6      uint16_t SPI_DataSize;
7
8      uint16_t SPI_CPOL;
9
10     uint16_t SPI_CPHA;
11
12     uint16_t SPI_NSS;
13
14     uint16_t SPI_BaudRatePrescaler;
15
16     uint16_t SPI_FirstBit;
17
18     uint16_t SPI_CRCPolynomial;
19 }SPI_InitTypeDef;

```

Table 411. SPI_Mode 值

SPI_Mode	描述
SPI_Direction_2Lines_FullDuplex	SPI 设置为双线双向全双工
SPI_Direction_2Lines_RxOnly	SPI 设置为双线单向接收
SPI_Direction_1Line_Rx	SPI 设置为单线双向接收
SPI_Direction_1Line_Tx	SPI 设置为单线双向发送

SPI_Direction 设置 spi 传输方式是什么，常规选择

SPI_Direction_2Lines_FullDuplex

SPI_Mode	描述
SPI_Mode_Master	设置为主 SPI
SPI_Mode_Slave	设置为从 SPI

SPI_Mode 设置 stm32spi 是做主机还是从机，我们 stm32 接的是外部 norflash，所以 stm32 是做主机

选择 SPI_Mode_Master

SPI_DataSize	描述
SPI_DataSize_16b	SPI 发送接收 16 位帧结构
SPI_DataSize_8b	SPI 发送接收 8 位帧结构

SPI_Datasize 配置数据帧长度(MISO/MOSI 数据位)，我们一般是一次数据写 8 位给 norflash 选择 SPI_DataSize_8b

SPI_CPOL	描述
SPI_CPOL_High	时钟悬空高
SPI_CPOL_Low	时钟悬空低

SPI_CPOL 配值 SCK 时钟线没传输数据的时候是什么电平

根据从机来选择

SPI_CPHA	描述
SPI_CPHA_2Edge	数据捕获于第二个时钟沿
SPI_CPHA_1Edge	数据捕获于第一个时钟沿

根据从器件，选择主器件的 SPI 数据采样边沿

SPI_CPHA_2Edge 是下降沿采样数据

SPI_CPHA_1Edge 是上升沿采样数据

SPI_NSS	描述
SPI_NSS_Hard	NSS 由外部管脚管理
SPI_NSS_Soft	内部 NSS 信号有 SSI 位控制

SPI_NSS_Hard 是由 MCU 自己产生片选信号，还是我们操作软件 gpio 来选择编选信号

SPI_NSS_Soft 是我们自己定义 GPIO 引脚来当做片选信号，这样我们可以定义多个 GPIO 多个片选，这样 SPI 控制器总线可以并联多个从机来传输数据，也就是一个 SPI 总线有多个 norflash 这样的设备

SPI_BaudRatePrescaler	描述
SPI_BaudRatePrescaler2	波特率预分频值为 2
SPI_BaudRatePrescaler4	波特率预分频值为 4

选择 SPI 传输速率，

SPI_FirstBit	描述
SPI_FirstBit_MSB	数据传输从 MSB 位开始
SPI_FirstBit_LSB	数据传输从 LSB 位开始

根据 norflash 的 SPI 数据格式，决定主机是发送 MSB 高位在前，还是 LSB 低位在前

SPI_CRCPolynomial CRC 校验暂时没做，因为必须是从机有 CRC 校验功能，主机才可以做。

1. 初始化 SPI 功能

```

/* Initialize the SPI1 according to the SPI_InitStructure members */
SPI_InitTypeDef SPI_InitStructure;
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_InitStructure.SPI_DataSize = SPI_DataSize_16b;
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
SPI_InitStructure.SPI_BaudRatePrescaler =
SPI_BaudRatePrescaler_128;
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
SPI_InitStructure.SPI_CRCPolynomial = 7;
SPI_Init(SPI1, &SPI_InitStructure); //选择把设置的 SPI 功能参数写入哪一路 SPI, 这里是 SPI1

```

2. 开启哪一路 spi

```
SPI_Cmd(SPI1, ENABLE); //开启 spi1 路
用实例来展示代码.....
```

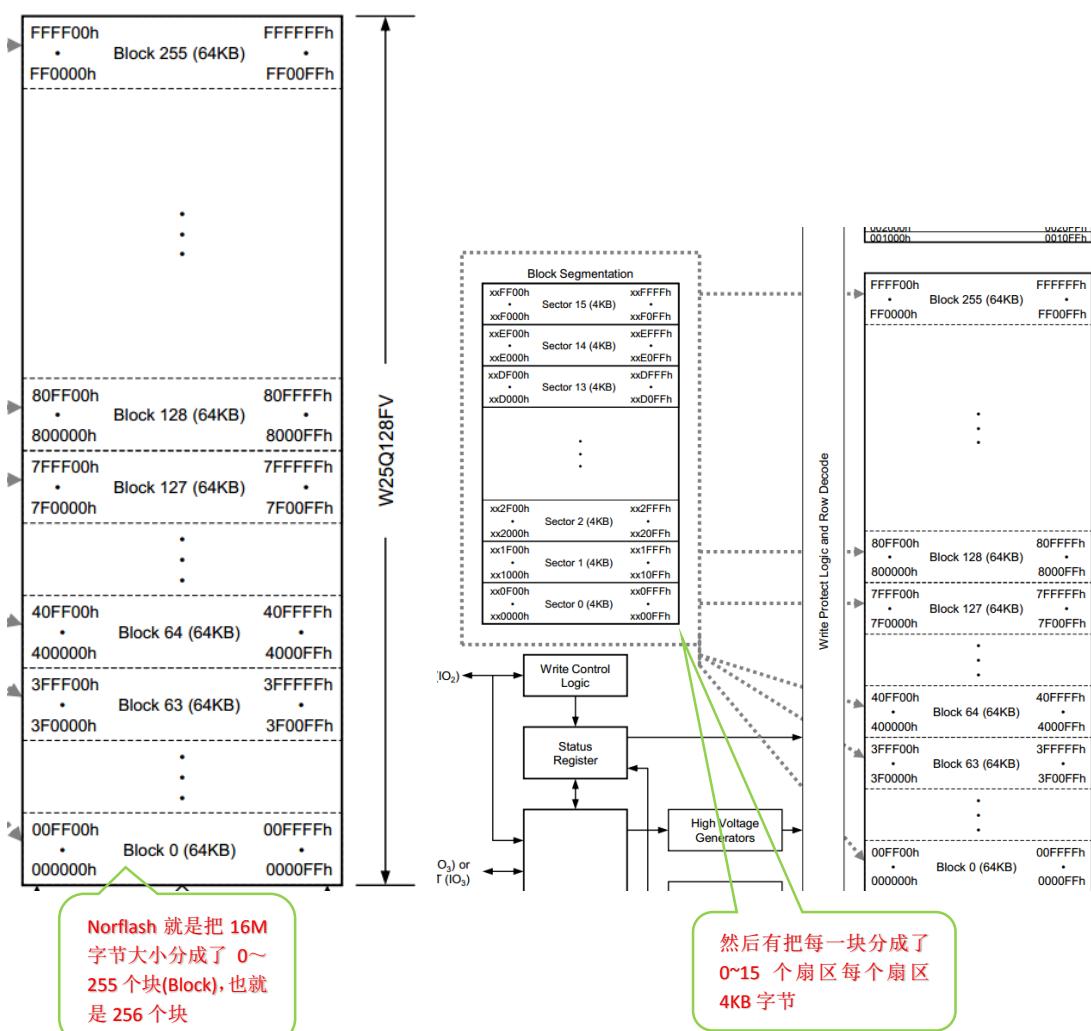
下面用 norflash 来实例操作 spi

W25Q128FV (128M-bit) 这个 128M 是 bit , bit 没有意义要换算成字节 byte 才有意义, 所以 $128/8=16M$ 这是一个 16Mbyte 的 norflash。包括 DDR 内存, EMMC 这些都喜欢用 bit 来扩大其词, 其实算成字节都很小。

– Quad SPI: CLK, /CS, IO₀, IO₁, IO₂, IO₃ 简称 **QSPI**, 就是一个时钟传输 4 个数据位, W25Q128 支持 QSPI, 但是我们用不到这么快的数据传输速度

Standard SPI: CLK, /CS, DI, DO, /WP, /Hold 普通标准 SPI, 我们就是用标准 SPI

SPI clock frequencies of up to 104MHz 这个 norflash 最大支持 104Mhz 传输速度, 但是我们 STM32F1 只支持最高 36M, 所以我们就传输 36M 速率。



注意: Flash 不管是写数据还是擦除数据, 都必须按照一个扇区去擦除或者写入, 像 EEPROM 那样一个字节一个字节读写是不行的。flash 一个扇区数 4096 个字节。所以你如果要去修改某个扇区的某个字节的数据。你只有把这个扇区的所有数据读出来存在 STM32 数组里面, 然后去修改数值里面对应的元素(地址)上面的值, 然后擦除 flash 要修改的扇区, 然后将 STM32 修改后的数组整个写入 flash 对应修改后的扇区。

比如我现在要修改 flash 里面的 20 扇区里面的 4096 字节里第 5 个字节里面的数据

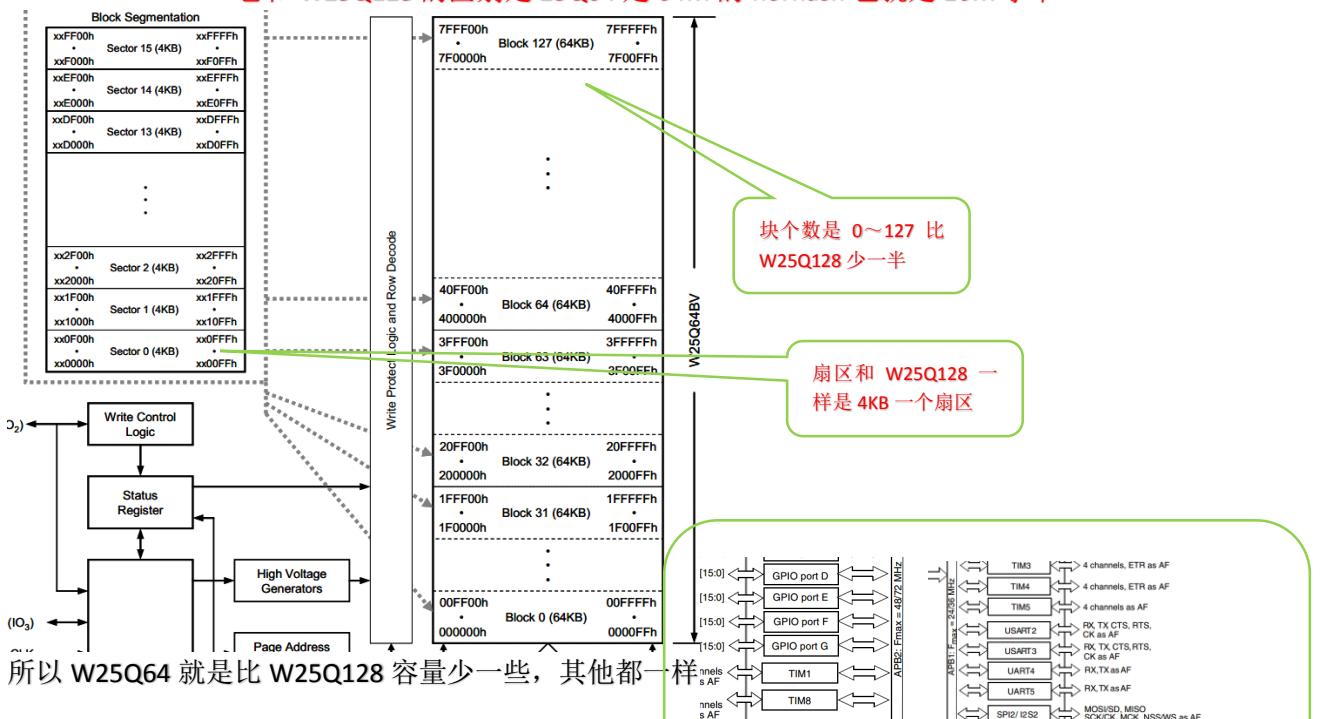
- 1.stm32 建立一个 4096 数组 buffer[4096]
- 2.stm32 读出 flash 第 20 扇区里面的 4096 字节到 buffer
- 3.stm32 修改 buffer 的第 5 个元素, buffer[5] = 3;
- 4.这样 buffer[5] 元素被修改了, 但是其他元素还是没变。
- 5.stm32 将 flash 第 20 个扇区擦除, 因为 flash 扇区里数据只能从 1 变成 0, 不能从 0 变成 1, 所以 20 扇区先全部擦除, 这样 20 扇区全部变成 1 了;
- 6.stm32 将 buffer 的 4096 个数据再次写入 flash 第 20 扇区。这样扇区是 1 的还是 1, 是 0 的就由 1 变成 0

所以 flash 存储器你去修改某一个字节是很浪费的, 因为你修改一个字节要搞得整个扇区都要一起背锅, 所以 flash 一般时候用来存储大量的固定数据。因为 flash 可以无限被 stm32 读数据, 但是不能被 stm32 无限写数据, flash 写入次数是有寿命的。所以 flash 适合用来做字库, 或者固定图片存储。但是如果你是大量的修改数据后写入 flash, 这种还是可以的因为虽然我擦除了这么多区域, 但是我又写了很多新数据进这些区域啊所劳有所的。还是看你怎么想, 怎么去优化 flash。

但是我们现在用的 norflash 擦除是按照扇区擦除, 但是写入的时候是可以一个字节一个字节写的, 只有 nandflash 是扇区擦除, 扇区写入。所以 norflash 不会坏块, nandflash 会出现坏块。

我们下面的实验用的是 W25Q64 norflash

W25Q64BV (64M-bit) 它和 W25Q128 的区别是 25Q64 是 64M 的 norflash 也就是 16M 字节



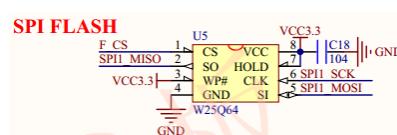
stm32f10x_spi.c

```
首先保证固件库有 spi 驱动程序
void spi_FUN_init(void)
{
    SPI_InitTypeDef SPI_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE); // 启动SPI1在APB2总线上的时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // 因为CS MISO MOSI SCK都在GPIO_A引脚,
    // 所以初始化GPIOA时钟就是了
}
```

因为 SPI1 经过 APB2 总线最小分频之后还有 36M 所以我们选择 SPI1, SPI2/SPI3 最小分频后只有 16M, 所以 SPI1 速度快些

F CS	PA2 16	PA1/ADC12_IN1/TIM1_CH1/TIM2_CH1
SD CS	PA3 17	PA2/U2_RX/ADC12_IN2/TIM2_CH3/TIM5_CH3
SDIO	GND 18	PA3/U2_RX/ADC12_IN3/TIM2_CH4/TIM5_CH4
NRF_CE	PA4 19	VSS
SPI1_CE	PA4 20	VDD
SPI1_SCK	PA5 21	PA4/SPI1_NSS/ADC12_IN4/DAC1_OUT
SPI1_MISO	PA6 22	PA5/SPI1_SCK/ADC12_IN5/DAC2_OUT
SPI1_MOSI	PA7 23	PA6/SPI1_MISO/ADC12_IN6/TIM3_CH1/TIM8_BKIN
NRF_CS	PC4 24	PA7/SPI1_MOSI/ADC12_IN7/TIM3_CH2/TIM8_CH1IN



```

//初始化SPI CS片选引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //普通GPIO做片选所以设置成推挽输出
GPIO_Init(GPIOA, &GPIO_InitStructure);

//初始化SPI SCK时钟引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //推挽复用输出
GPIO_Init(GPIOA, &GPIO_InitStructure);

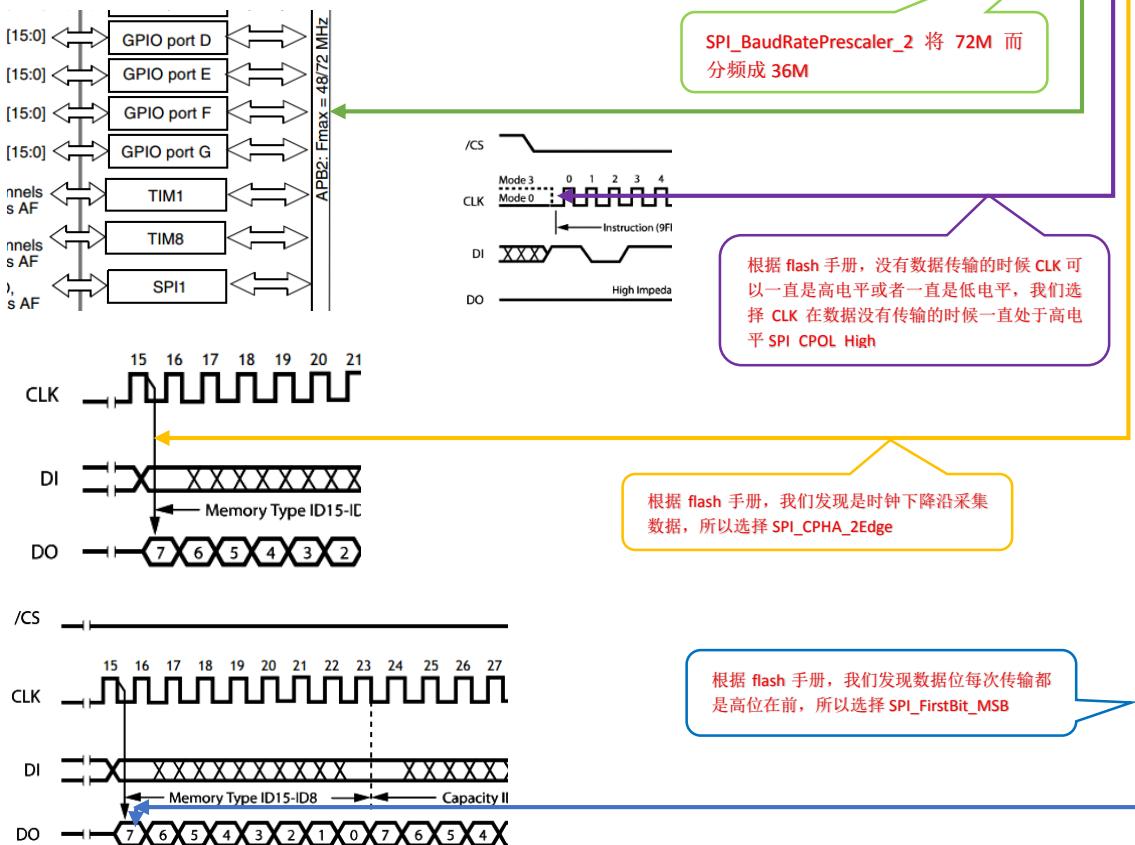
//初始化SPI MISO输入引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);

//初始化SPI MOSI输出引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //推挽复用输出
GPIO_Init(GPIOA, &GPIO_InitStructure);

GPIO_SetBits(GPIOA, GPIO_Pin_2); //为了安全起见，先将CS引脚设置成高电平
/*初始化SPI1的工作模式*/
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //标准的SPI传输方式
SPI_InitStructure.SPI_Mode = SPI_Mode_Master; //STM32是主机，所以选择SPI主机模式
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; //SPI每次发送一个字节
SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //CS硬件设置成软件控制，就是我们上边的GPIO控制
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; //根据flash手册选择高位在前
SPI_InitStructure.SPI_CRCPolynomial = 0;
SPI_Init(SPI1, &SPI_InitStructure);

SPI_Cmd(SPI1, ENABLE); //记得要打开SPI1
}

```



```

/*STM32发送一个字节给外设*/
void SPI_send_Byte(uint8_t data)
{
    //发送数据前先确保STM32内部的TXE寄存器为空，才能发送数据
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
    SPI_I2S_SendData(SPI1, data); //发送数据
    /*我们不知道这个字节是否发送完成，但是因为SPI发数据的同时也在接受数据
     * 所以我用RXNE标志来判断是否发送完成
     */
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET); //没有发送完循环等待

    SPI_I2S_ReceiveData(SPI1);
    //这里没有需要接受的数据，也必须写接受函数，否则会出现数据不对的问题
}

/*STM32接受一个外设的数据*/
uint8_t SPI_read_Byte(void)
{
    uint8_t data;
    /*因为SPI是循环同步的，所以SPI接受数据必须依靠发送数据的函数产生时钟
     * 这样SPI才能接受数据，意思就是SPI发送一个字节的时钟，同时也在接受一个字节*/
    //while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
    SPI_I2S_SendData(SPI1, 0x00); //所以我还得模拟发送数据产生时钟什么数据无所谓
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET); //没有发送完循环等待
    data = SPI_I2S_ReceiveData(SPI1);
    return data;
}

```

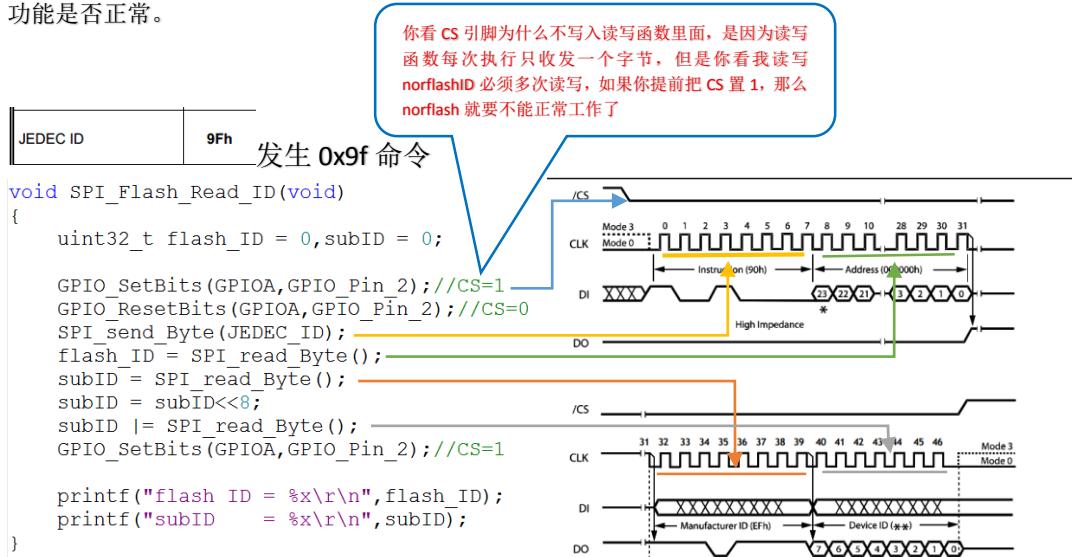
因为发送数据也要写数据接收函数来处理发送数据不对的 bug，所以下面写了个同时读写的函数，你可以选择上面的发送接收分离的函数来操作 SPI，也可以选择下面的发送读写同步的函数来操作 SPI

```

/*STM32发送一个字节给外设同时接受一个字节*/
uint8_t SPI_Send_Read_Byte(uint8_t data)
{
    //发送数据前先确保STM32内部的TXE寄存器为空，才能发送数据
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
    SPI_I2S_SendData(SPI1, data); //发送数据
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET); //没有发送完循环等待
    return SPI_I2S_ReceiveData(SPI1);
}

```

现在函数写完了我们要来测试下到底代码写得对不对，这种最好就是读取 norflash 芯片的 ID，当然除了 norflash 可以测试，其他带 SPI 接口的芯片也可以测试，那些芯片应该有状态寄存器让你来测试 SPI 接口的功能是否正常。



这样 norflashID 就读写完成了

```

void SPI_Flash_Read_ID(void)
{
    uint32_t flash_ID = 0, subID = 0;

    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA, GPIO_Pin_2); //CS=0
    SPI_Send_Read_Byte(JEDEC_ID);
    flash_ID = SPI_Send_Read_Byte(0x00);

    subID = SPI_Send_Read_Byte(0x00);
    subID = subID << 8;
    subID |= SPI_Send_Read_Byte(0x00);
    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1

    printf("flash ID = %x\r\n", flash_ID);
    printf("subID      = %x\r\n", subID);
}

```

XXXXXXXX
 flash ID = ef
 subID = 4017
 XXXXXXXX
 flash ID = ef
 subID = 4017
 XXXXXXXX
 flash ID = ef
 subID = 4017
 XXXXXXXX
 flash ID = ef

11.2.1 Manufacturer and Device Identification

MANUFACTURER ID	(M7-M0)	
Winbond Serial Flash	EFh	
Device ID	(ID7-ID0)	(ID15-ID0)
Instruction	ABh, 90h	9Fh
W25Q64BV	10h	4017h

结果和 norflash 手册要求的制造商 ID 和设备 ID 一样的

代码清单：

```

void spi_FUN_init(void)
{
    SPI_InitTypeDef SPI_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1,ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_SetBits(GPIOA,GPIO_Pin_2);
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
    SPI_InitStructure.SPI_NSS = SPI NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStructure.SPI_CRCPolynomial = 0;
    SPI_Init(SPI1, &SPI_InitStructure);

    SPI_Cmd(SPI1, ENABLE);
}

```

```

void SPI_send_Byte(uint8_t data)
{
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
    SPI_I2S_SendData(SPI1, data);

    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
    SPI_I2S_ReceiveData(SPI1);
}

uint8_t SPI_read_Byte(void)
{
    uint8_t data;

    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
    SPI_I2S_SendData(SPI1, 0x00);
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
    data = SPI_I2S_ReceiveData(SPI1);
    return data;
}

uint8_t SPI_Send_Read_Byte(uint8_t data)
{
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
    SPI_I2S_SendData(SPI1, data);
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
    return SPI_I2S_ReceiveData(SPI1);
}

```

SPI 操作 norflash(W25Q64)进行数据块读写

有了上一节的 SPI 驱动，我们就可以按照格式读写 norflash 了

擦除 norflash 扇区

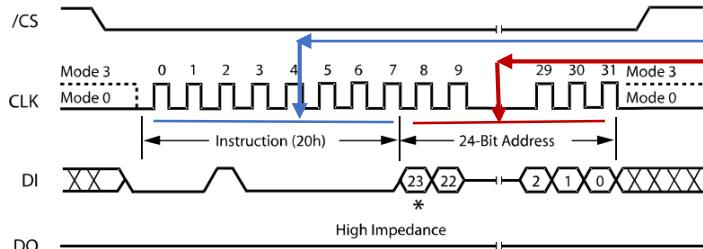
MANUFACTURER	CODE	STATE 1	STATE 3	STATE 4
	D8h	A23-A16	A15-A8	A7-A0
	52h	A23-A16	A15-A8	A7-A0
	20h	A23-A16	A15-A8	A7-A0

这里有每次擦除 flash 64KB,32KB 扇区的我们不用 我们只用每次擦除 4KB 的。那么发送 20h

```
/*擦除flash指定扇区*/
void spi_Erase_sector(uint32_t addr)
{
    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA, GPIO_Pin_2); //CS=0

    SPI_Send_Read_Byte(ERASE_SECTOR); //发送0x20命令
    SPI_Send_Read_Byte((addr>>16)&0xff); //发送地址的高8位
    SPI_Send_Read_Byte((addr>>8)&0xff); //发送地址中8位
    SPI_Send_Read_Byte(addr&0xff); //发送地址低8位

    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1 flash地址写入完毕
}
```

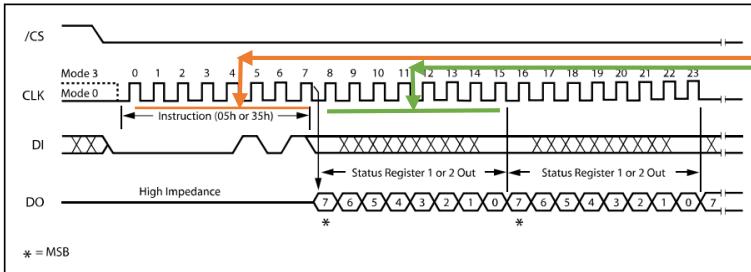


/*因为擦除flash非常消耗时间,所以要写等待flash内部擦除完成*/

```
void wait_flash_Erase()
{
    uint8_t status_reg = 0;

    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA, GPIO_Pin_2); //CS=0

    SPI_Send_Read_Byte(READ_STATUS); //发送读取状态寄存器命令
    do
    {
        status_reg = SPI_Send_Read_Byte(0x00); //发送无用的数据产生时钟,把flash返回的数据读出来
    }while((status_reg & 0x01) == 1); //如果状态寄存器是1就表示flash还没有擦除好,继续等待
    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1 flash擦除完成,就拉高cs引脚
}
```



不停的发送时钟读取
每次收到的字节,直
到读取到 0x01 表示
flash 擦除完成

Figure 6. Read Status Register Instruction Sequence Diagram

```

/*读取flash扇区里面的内容*/
/*addr:要读取flash扇区的首地址
 * readbuff:将读取的一个扇区4096字节写入buff,
 * byte_number:读取多少个字节
 */
void READ_norflash_sector(uint32_t addr,uint8_t *readbuff,uint32_t byte_number)
{
    GPIO_SetBits(GPIOA,GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA,GPIO_Pin_2); //CS=0

    SPI_Send_Read_Byte(READ_DATA); //发送读取扇区数据命令
    SPI_Send_Read_Byte((addr>>16)&0xff); //发送地址的高8位
    SPI_Send_Read_Byte((addr>>8)&0xff); //发送地址中8位
    SPI_Send_Read_Byte(addr&0xff); //发送地址低8位

    while(byte_number--)
    {
        *readbuff = SPI_Send_Read_Byte(0x00); //发送无关数据产生时钟,才能读取
        readbuff++; //buff移动到下一个地址
    }

    GPIO_SetBits(GPIOA,GPIO_Pin_2); //CS=1
}

```

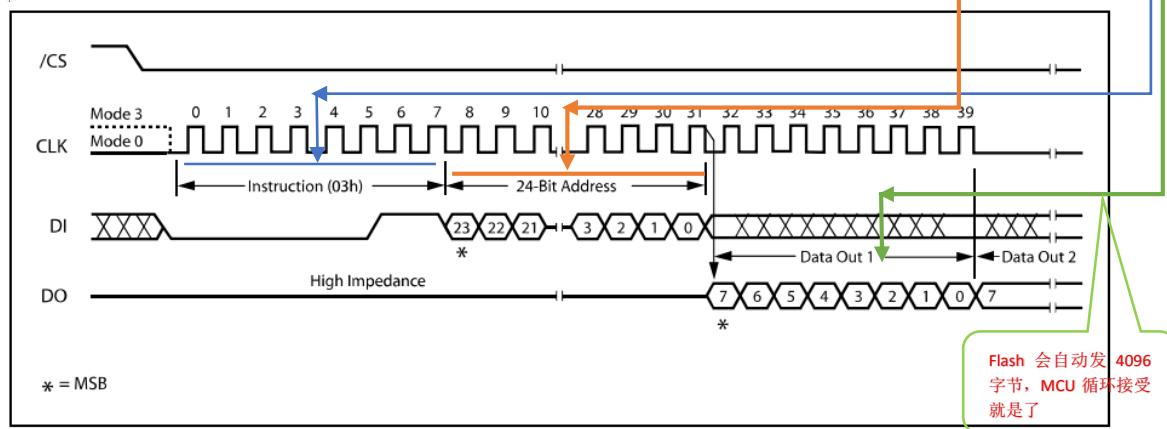


Figure 8. Read Data Instruction Sequence Diagram

```

int main(void)
{
    uint8_t buff[4096]; //buff在栈区定义这么大是有问题的
    uint32_t i=0;
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    spi_FUN_Init(); //SPI初始化

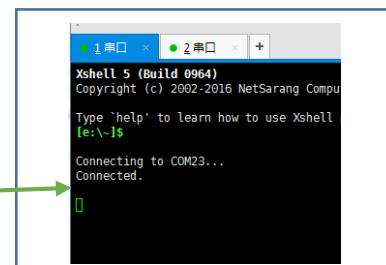
    printf("xxxxzzzz\r\n");
    SPI_Flash_Read_ID();
    delay_ms(5000);

    while(1){
        //擦除flash 0扇区 你只需要写入0扇区, flash会自动从0扇区开始擦除4096个地址上的数据
        spi_Erase_Sector(0);
        wait_flash_Erase(); //等待flash内部擦除完成
        READ_norflash_sector(0,buff,4096); //把flash的0扇区4096字节读出来

        for(i=0;i<4096;i++)
        {
            printf("0x%02x ",buff[i]);
            if(i%10 == 0) //这里只是让数据换行, 好看些
                printf("\r\n");
        }
        delay_ms(5000);
    }
}

```

我们发现程序并没有跑起来, 启动就卡死了



```

int main(void)
{
    uint8_t buff[4096]; //buff在栈区定义这么大是有问题的
    uint32_t i=0;
    RCC_Configuration(); //初始化时钟
    USART_config(115200); //串口初始化

    Stack_Size EQU 0x00000400
    Stack_Mem AREA STACK, NOINIT, READWRITE, ALIGN=3
    Stack_Mem SPACE Stack_Size
    __initial_sp

; <h> Heap Configuration
;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

    Heap_Size EQU 0x00000200

```

我们发现 hd.s 汇编程序 Stack_size 栈区大小才 0x400(也就是 1024 字节大小)

堆区也是少得可怜
0x200(512 字节)

第 1 种方法就是修改 Stack_size 的值，改成 4096 的大小，但是我还没有查看 STM32 的资料，允许最大栈区是多大？

第 2 中方法，就是把 buff 数组放在全局区，全局区也是很大的，我这里就先放在全局区。

```

uint8_t buff[4096]; //buff在全局区就不会溢出了
int main(void)
{
    uint32_t i=0;
    RCC_Configuration(); //初始化时钟
    USART_config(115200); //初始化串口
    spi_FUN_init(); //SPI初始化

    printf("xxxxxxxx\r\n");
    SPI_Flash_Read_ID();
    delay_ms(5000);

    while(1){
        //擦除flash 0扇区 你只需要写入0扇区，flash会自动从0扇区开始擦除4096个地址上的数据
        spi_Erase_sector(0);
        wait_flash_Erase(); //等待flash内部擦除完成
        READ_norflash_sector(0,buff,4096); //把flash的0扇区4096字节读出来

        for(i=0;i<4096;i++)
        {
            printf("0x%02x ",buff[i]);
            if(i%10 == 0) //这里只是让数据换行，好看些
                printf("\r\n");
        }
    }
}

```

Connecting to COM23...
Connected.

2 0xf 0xfd 0xe0 0xff 0x0 0x10 0x1 0x21 0x10 0x3 0x41 0x10 0x5 0x61 0x10 0x7 0x81 0x10 0x9 0x1 0x10 0xb 0xc1 0x10 0xd 0xe1 0x10 0xf 0x1 0x11 0x11 0x11 0x21 0x11 0x13 0x41 0x11 0x15 0x61 0x11 0x17 0x81 0x11 0x19 0x11 0x11 0x1b 0x1 0x11 0x11 0x11 0x11 0x12 0x27 0x81 0x12 0x29 0x1 0x12 0x2b 0x1 0x11 0x12 0x21 0x21 0x12 0x23 0x41 0x12 0x1 0x11 0x12 0x21 0x21 0x12 0x23 0x41 0x12 0x2b 0x1 0x11 0x12 0x27 0x81 0x12 0x29 0x1 0x12 0x2b 0x1 0x12 0x2d 0x1 0x12 0x2f 0x1 0x13 0x31 0x21 0x13 0x33 0x41 0x13 0x35 0x61 0x13 0x37 0x81 0x13 0x39 0xa1 0x13 0x3b 0x1 0x13 0x3d 0x1 0x13 0x3f 0x1 0x14 0x41 0x21 0x14 0x43 0x41 0x14 0x45 0x61 0x14 0x47 0x81 0x14 0x49 0x1 0x14 0x4b 0x1 0x14 0x4d 0x1 0x14 0x4f 0x1 0x15 0x51 0x21 0x15 0x53 0x41 0x15 0x55 0x61 0x15 0x57 0x81 0x15 0x59 0x1 0x15 0x5 0x1 0x15 0x5d 0x1 0x15 0x5f 0x1 0x16 0x61 0x21 0x16 0x63 0x41 0x16 0x65 0x61 0x16 0x67 0x81 0x16 0x69 0x1 0x16 0x6b 0x1 0x16 0x6d 0x1 0x16 0x6f 0x1 0x17 0x71 0x21 0x17 0x73 0x41 0x17 0x75 0x61 0x17 0x77 0x81 0x17 0x79 0x1 0x17 0x7b 0x1

程序正常运行了，但是我发现扇区并没有被擦除，因为擦除后扇区每个字节应该都是 0xff

看来还要修改下

这里有个很坑的地方，就是 MCU 写 flash 扇区或者 flash 自己写自己的扇区，都要设置写使能

```
/*norflash写使能*/
void norflash_Write_ENABLE(void)
{
    GPIO_SetBits(GPIOA,GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA,GPIO_Pin_2); //CS=0

    SPI_Send_Read_Byte(WRITE_ENABLE); //写使能命令

    GPIO_SetBits(GPIOA,GPIO_Pin_2); //CS=1
}
```

```
/*擦除flash指定扇区*/
void spi_Erase_sector(uint32_t addr)
{
    norflash_Write_ENABLE(); //写使能
    GPIO_SetBits(GPIOA,GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA,GPIO_Pin_2); //CS=0

    SPI_Send_Read_Byte(ERASE_SECTOR); //发送0x20命令
    SPI_Send_Read_Byte((addr>>16)&0xff); //发送地址高8位
    SPI_Send_Read_Byte((addr>>8)&0xff); //发送地址中8位
    SPI_Send_Read_Byte(addr&0xff); //发送地址低8位

    GPIO_SetBits(GPIOA,GPIO_Pin_2); //CS=1 flash地址
}
```

```
Connecting to COM23...
Connected.

ffff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff 0fff
```

这样执行程序，flash 扇区擦除成功

如果是新买的 norflash 里面扇区数据本来就是 0xff，那怎么才能证明我是成功擦除的呢？

这就要我们先实现写扇区函数后再来测试

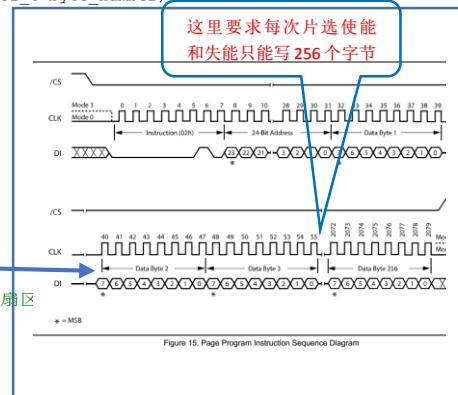
```
/*写数据进flash扇区*/
/*addr:要写入flash扇区的起始地址
 * writebuff:将256个字节写入flash扇区的缓存
 * byte_number:写多少个字节
 */
void WRITE_norflash_sector(uint32_t addr,uint8_t *writebuff,uint32_t byte_number)
{
    spi_Erase_sector(addr); //写扇区之前要擦除扇区
    wait_flash_Erase(); //等待flash内部擦除完成

    norflash_Write_ENABLE(); //擦除扇区完了之后，写扇区也要使能
    GPIO_SetBits(GPIOA,GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA,GPIO_Pin_2); //CS=0

    SPI_Send_Read_Byte(WRITE_DATA); //发送写数据命令
    SPI_Send_Read_Byte((addr>>16)&0xff); //发送地址的高8位
    SPI_Send_Read_Byte((addr>>8)&0xff); //发送地址中8位
    SPI_Send_Read_Byte(addr&0xff); //发送地址低8位

    while(byte_number--)
    {
        SPI_Send_Read_Byte(*writebuff); //发送每一个字节写入flash扇区
        writebuff++; //buff移动到下一个地址
    }

    GPIO_SetBits(GPIOA,GPIO_Pin_2); //CS=1
```



```

uint8_t readbuff[4096]; //buff在全局区就不会溢出了
uint8_t writebuff[4096]; //写数据缓存区

int main(void)
{
    uint32_t i=0;
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    SPI_FUN_Init(); //SPI初始化

    delay_ms(5000);

    for(i=0;i<256;i++)
    {
        writebuff[i] = i;
    }

    WRITE_norflash_sector(0, writebuff, 30);
    //虽然擦除的是一个扇区4096个字节，但是写入的时候一次只能写256个字节，
    //所以需要多次执行WRITE_norflash_sector函数来写数据，我这里先写30个字节测试下
    while(1){

        READ_norflash_sector(0, readbuff, 4096); //把flash的0扇区4096字节读出来
    }
}

```

Connecting to COM23...
Connected.

```

0x0
0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa
0xb 0xc 0xd 0xe 0xf 0x10 0x11 0x12 0x13 0x14
0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0xff
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff

```

这就是成功写了 30 个字节，如果你想测试擦除命令，现在就可以擦除这片扇区看看擦除功能是否正常。其实在写 flash 函数里面的时候也验证过擦除函数的。

现在我们来完善写 flash 扇区的程序，做到一次写入 4096 字节，顺便让写 flash 的时候处于安全写入

```

/*写数据进flash扇区*/
/*addr:要写入flash扇区的起始地址
 * writebuff:将256个字节写入flash扇区的缓存
 * byte_number:写多少个字节
 */
void WRITE_norflash_sector(uint32_t addr, uint8_t *writebuff, uint32_t byte_number)
{
    if(byte_number > 256) //判断如果写入大于256个字节就报错，退出程序
    {
        printf("write flash data error\r\n");
        return;
    }
    else
    {
        spi_Erase_sector(addr); //写扇区之前要擦除扇区
        wait_flash_Erase(); //等待flash内部擦除完成

        norflash_Write_FNARTE(); //擦除扇区完了之后，写扇区也要使能
    }
}

```

写 flash 扇区程序加入了字节数判断，因为是页写，所以执行这个程序一次最大只能写入 256 字节

下面才是随意大小字节写 flash 的程序

```

/*写数据进flash多个扇区*/
/*WriteAddr:要写入flash扇区的起始地址
 * writebuff:将任意数量字节写入flash扇区的缓存
 * byte_number:写多少个字节
 */
void WRITE_norflash_multisection(uint32_t WriteAddr,uint8_t *writebuff,uint32_t byte_number)
{
    uint8_t page = 0,addr = 0,count = 0,temp = 0,nofullpage = 0;
    addr = WriteAddr%256;
    count = 256 - addr;
    page = byte_number / 256;//计算写多少页
    nofullpage = byte_number % 256;//计算剩下不满一页的字节数
    if(addr == 0)//地址对齐
    {
        if(page == 0)//如果byte_number的值不是4096而是小于256,就按照256页写就是
        {
            WRITE_norflash_sector(WriteAddr,writebuff,byte_number);//这个函数一次只能256字节
        }
        else
        {
            while(page--)
            {
                WRITE_norflash_sector(WriteAddr,writebuff,256);
                WriteAddr+=256;
                writebuff+=256;
            }
            WRITE_norflash_sector(WriteAddr,writebuff,nofullpage);//把最后剩下的多余不满256字节的数据写完
        }
    }
    else
    {
        byte_number= byte_number - count;
        page = byte_number / 256;//计算写多少页
        nofullpage = byte_number % 256;//计算剩下不满一页的字节数
        WRITE_norflash_sector(WriteAddr,writebuff,count);
        WriteAddr = WriteAddr + count;
        writebuff = writebuff + count;

        while(page--)
        {
            WRITE_norflash_sector(WriteAddr,writebuff,256);
            WriteAddr = WriteAddr + 256;
            writebuff = writebuff + 256;
        }
        if(nofullpage !=0 )
        {
            WRITE_norflash_sector(WriteAddr,writebuff,nofullpage);//如果有多余不满一页的数据就把它写完
        }
    }
}

```

这种取余运算比如 $5\%256=5$, 小于 256 的被除数结果等于本身, 大于 256 的被除数比如 $257\%256=1$, $258\%256=2$, 如果是 $512\%256=0$, 因为 512 能被 256 整除

计算我这次传入进来的 writebuff 数据量要写多少页, 比如我传入的是 4097, 那么 4097 个数/256 绝对不是完全等于 16 页, 还有最后一个字节

如果大于 256, 就要写一页, 然后 writeAddr 地址和数组 writebuff 的首地址向后 256 个地址偏移, 那么这句执行完了之后地址和数组就在 257 地址位置了, 这是 257 地址就是下一页数据的首地址, 如此循环写完被 256 整除的所有页

最后剩下没有被整除的页, 也许就几个字节, 写入地址和数组偏移的最后一页。

如果我传入进来要写的首地址是 5, 或者 16, 258, 1028, 这些无法被 256 整除的地址

norflash数据

0	1	2	3	4	5	256	257	258	259
---	---	---	---	---	---	-------	-----	-----	-----	-----

norflash地址

如果 WriterAddr = 0
那么 addr = 0
那么 count = 256
证明写入的地址首地址是 256 的整数倍, 比如 512, 1024, 2048, 4096。
我这里加入首地址是 0

这些不是 256 整数倍的地址, 进入 else 语句

norflash数据

0	1	2	3	4	5	256	257	258	259	1024	1028
---	---	---	---	---	---	-------	-----	-----	-----	-----	-------	------	------	------

norflash地址

WriteAddr 传入的首地址是 5
首地址也可能是 258
首地址也可能是 1028
这些都不能被 256 整除

下面进行任意大小数据写 flash 函数测试

```
uint8_t readbuff[4096]; //buff在全局区就不会溢出了
uint8_t writebuff[4096]; //写数据缓存区
spi_FUN_init(); //SPI初始化
delay_ms(5000);
for(i=0;i<4096;i++)
{
    writebuff[i] = i;
}
WRITE_norflash_multisection(0,writebuff,4096);
while(1)
{
    READ_norflash_sector(0,readbuff,4096); //把flash的0扇区4096字节读出来
    for(i=0;i<4096;i++)
    {
        printf("0x%02x ",readbuff[i]);
        if(i%10 == 0) //这里只是让数据换行，好看些
            printf("\r\n");
    }
    delay_ms(5000);
}
```

```
Connecting to COM23...
Connected.

0x0
0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa
0xb 0xc 0xd 0xe 0xf 0x10 0x11 0x12 0x13 0x14
0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e
0x1f 0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28
0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f 0x30 0x31 0x32
0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3a 0x3b 0x3c
0x3d 0x3e 0x3f 0x40 0x41 0x42 0x43 0x44 0x45 0x46
0x47 0x48 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f 0x50
0x51 0x52 0x53 0x54 0x55 0x56 0x57 0x58 0x59 0x5a
0x5b 0x5c 0x5d 0x5e 0x5f 0x60 0x61 0x62 0x63 0x64
0x65 0x66 0x67 0x68 0x69 0x6a 0x6b 0x6c 0x6d 0x6e
0x6f 0x70 0x71 0x72 0x73 0x74 0x75 0x76 0x77 0x78
0x79 0x7a 0x7b 0x7c 0x7d 0x7e 0x7f 0x80 0x81 0x82
0x83 0x84 0x85 0x86 0x87 0x88 0x89 0x8a 0x8b 0x8c
0x8d 0x8e 0x8f 0x90 0x91 0x92 0x93 0x94 0x95 0x96
0x97 0x98 0x99 0x9a 0x9b 0x9c 0x9d 0x9e 0x9f 0xa0
0xa1 0xa2 0xa3 0xa4 0xa5 0xa6 0xa7 0xa8 0xa9 0xa0
0xab 0xac 0xad 0xae 0xaf 0xb0 0xb1 0xb2 0xb3 0xb4
0xb5 0xb6 0xb7 0xb8 0xb9 0xb0 0xbc 0xbd 0xb6
0xbf 0xc0 0xc1 0xc2 0xc3 0xc4 0xc5 0xc6 0xc7 0xc8
0xc9 0xca 0xcb 0xcc 0xcd 0xce 0xcf 0xd0 0xd1 0xd2
0xd3 0xd4 0xd5 0xd6 0xd7 0xd8 0xd9 0xd0 0xd8
0xd9 0xe0 0xe1 0xe2 0xe3 0xe4 0xe5 0xe6
0xe7 0xe8 0xe9 0xea 0xeb 0xec 0xed 0xee 0xef 0xf0
0xf1 0xf2 0xf3 0xf4 0xf5 0xf6 0xf7 0xf8 0xf9 0xfa
0xfb 0xfc 0xfd 0xfe 0xff 0x0 0x1 0x2 0x3 0x4
0x5
```

根据测试出的数据感觉没有问题了，为什么 0x00~0x0f 在重复显示呢？是因为我循环到 256 字节后，还是只能显示 8 位，所以显示的永远都是低 8 位

如果心里面还没有底，下面用字符串来测试

```
uint8_t writestr[4096] = "1,2,3,4,5,6,7,8,9,10,\r\n"
11,12,13,14,15,16,17,18,\r\n"
19,20,21,22,23,24,25,26,27,\r\n"
28,29,30,31,32,33,34,35,36,\r\n"
37,38,39,40,41,42,43,44,45,\r\n"
46,47,48,49,50,51,52,53,54,\r\n"
55,56,57,58,59,60,61,62,63,\r\n"
64,65,66,67,68,69,70,71,72,\r\n"
73,74,75,76,77,78,79,80,81,\r\n"
82,83,84,85,86,87,88,89,90,\r\n"
91,92,93,94,95,96,97,98,99,\r\n"
100,101,102,103,104,105,106,\r\n"
107,108,109,110,111,112,113,\r\n"
114,115,116,117,118,119,120,\r\n"
121,122,123,124,125,126,127,\r\n"
128,129,130,131,132,133,134,\r\n"
135,136,137,138,139,140,141,\r\n"
142,143,144,145,146,147,148,\r\n"
149,150,151,152,154,155,156,\r\n"
157,158,159,160,161,162,163,\r\n"
164,165,166,167,168,169,170,\r\n"
171,172,173,174,175,176,177,\r\n"
178,179,180,181,182,183,184,\r\n"
185,186,187,188,189,190,191,
192,193,194,195,196,197,198,
199,200,201,202,203,204,255,
256,301,302,303,304,305,306";
```

```
int main(void)
{
    uint32_t i=0;
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    spi_FUN_init(); // SPI 初始化

    delay_ms(5000);

    WRITE_norflash_multisection(0,writestr,4096); // 将字符串写入norflash

    while(1)
    {
        READ_norflash_sector(0,readstr,4096); // 把flash的0扇区4096字节读出来
        printf("-----%s-----\r\n",readstr); // 输出超过256字节的字符串测试
        delay_ms(5000);
    }

    return 0;
}
```

写一堆字符串给 norflash，要超过 256 字节哦

```
Connecting to COM23...
Connected.

-----1,2,3,4,5,6,7,8,9,10,
11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,
28,29,30,31,32,33,34,35,36,
37,38,39,40,41,42,43,44,45,
46,47,48,49,50,51,52,53,54,
55,56,57,58,59,60,61,62,63,
64,65,66,67,68,69,70,71,72,
73,74,75,76,77,78,79,80,81,
82,83,84,85,86,87,88,89,90,
91,92,93,94,95,96,97,98,99,
100,101,102,103,104,105,106,
107,108,109,110,111,112,113,
114,115,116,117,118,119,120,
121,122,123,124,125,126,127,
128,129,130,131,132,133,134,
135,136,137,138,139,140,141,
142,143,144,145,146,147,148,
149,150,151,152,154,155,156,
157,158,159,160,161,162,163,
164,165,166,167,168,169,170,
171,172,173,174,175,176,177,
178,179,180,181,182,183,184,
185,186,187,188,189,190,191,
192,193,194,195,196,197,198,
199,200,201,202,203,204,255,
256,301,302,303,304,305,306-----
```

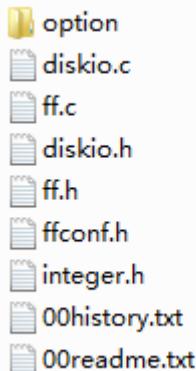
测试没有问题

文件系统移植到 norflash

文件系统代码在这个地址下载 http://elm-chan.org/fsw/ff/00index_e.html

- 进入 Download: [Previous Releases](#)

我下载的版本是 Fatfs R0.11a



这是解压后的 src 文件

- [disk_status](#) - Get device status
- [disk_initialize](#) - Initialize device
- [disk_read](#) - Read sector(s)
- [disk_write](#) - Write sector(s)
- [disk_ioctl](#) - Control device dependent functions
- [get_fattime](#) - Get current time

这些接口函数需要我们去实现，就是在这些函数里面放入我们的 norflash 读写程序。

先分析每个文件的作用

Integer.h 文件，数据类型定义

```
#ifndef _FF_INTEGER
#define _FF_INTEGER

#ifndef _WIN32 /* Development platform */

#include <windows.h>
#include <tchar.h>

#else /* Embedded platform */

/* This type MUST be 8-bit */
typedef unsigned char BYTE;

/* These types MUST be 16-bit */
typedef short SHORT;
typedef unsigned short WORD;
typedef unsigned short WCHAR;

/* These types MUST be 16-bit or 32-bit */
typedef int INT;
typedef unsigned int UINT;

/* These types MUST be 32-bit */
typedef long LONG;
typedef unsigned long DWORD;

#endif

#endif
```

这个文件就是重新定义数据类型，比如
int 类型在 51 单片机是 16 位，在 stm32
或者 linux 系统就是 32 位

diskio.c 底层驱动函数移植文件

```
DSTATUS disk_status (
    BYTE pdrv      /* Physical drive number to identify the drive */
)
{
    DSTATUS stat;
    int result;

    switch (pdrv) {
        case ATA :
            result = ATA_disk_status();

            // translate the reslut code here

            return stat;

        case MMC :
            result = MMC_disk_status();

            // translate the reslut code here

            return stat;

        case USB :
            result = USB_disk_status();

            // translate the reslut code here

            return stat;
    }
    return STA_NOINIT;
}
```

disk_status 是返回我存储设备现在的状态，比如设备现在正在读写，或者设备已经损坏这些信息

```
DSTATUS disk_initialize (
    BYTE pdrv      /* Physical drive number to identify the drive */
)
{
    DSTATUS stat;
    int result;

    switch (pdrv) {
        case ATA :
            result = ATA_disk_initialize();

            // translate the reslut code here

            return stat;

        case MMC :
            result = MMC_disk_initialize();

            // translate the reslut code here

            return stat;

        case USB :
            result = USB_disk_initialize();

            // translate the reslut code here

            return stat;
    }
    return STA_NOINIT;
}
```

ATA 存储设备初始化

MMC/SD 卡存储设备初始化

USB 存储设备/U 盘初始化

像我们的 spi norflash 芯片初始化函数就是写在这里

```
DRESULT disk_read (
    BYTE pdrv,      /* Physical drive number to identify the drive */
    BYTE *buff,     /* Data buffer to store read data */
    DWORD sector,   /* Sector address in LBA */
    UINT count      /* Number of sectors to read */
)
{
    DRESULT res;
    int result;

    switch (pdrv) {
        case ATA :
            // translate the arguments here

            result = ATA_disk_read(buff, sector, count);

            // translate the reslut code here

            return res;

        case MMC :
            // translate the arguments here

            result = MMC_disk_read(buff, sector, count);

            // translate the reslut code here

            return res;

        case USB :
            // translate the arguments here

            result = USB_disk_read(buff, sector, count);
    }
}
```

这是 read 函数的底层实现，我们只需要将 SPI norflash 的读扇区函数放在这里就是了

```

#if _USE_WRITE
DRESULT disk_write (
    BYTE pdrv,           /* Physical drive number to identify the drive */
    const BYTE *buff,     /* Data to be written */
    DWORD sector,         /* Sector address in LBA */
    UINT count            /* Number of sectors to write */
)
{
    DRESULT res;
    int result;

    switch (pdrv) {
        case ATA :
            // translate the arguments here

            result = ATA_disk_write(buff, sector, count);

            // translate the result code here

            return res;

        case MMC :
            // translate the arguments here

            result = MMC_disk_write(buff, sector, count);

            // translate the result code here
    }

    #if _USE_IOCTL
    DRESULT disk_ioctl (
        BYTE pdrv,           /* Physical drive number (0..) */
        BYTE cmd,             /* Control code */
        void *buff            /* Buffer to send/receive control data */
    )
    {
        DRESULT res;
        int result;

        switch (pdrv) {
            case ATA :
                // Process of the command for the ATA drive

                return res;

            case MMC :
                // Process of the command for the MMC/SD card
    }
}

```

这是 write 函数实现，spi norflash 写扇区函数就放在这里

这是 ioctl 函数实现，应用层想知道现在 norflash 的存储空间有多大，使用了多少。那么存储空间计算函数就放在这里

所以我们移植的时候只需要修改 diskio.c 文件里面的内容就是了，其它文件都不用动。

ff.c 文件就是文件系统核心文件，这个文件我们不用关心，移植上去就是了。

Option 目录下 cc936.c 文件是简体中文 GBK 和 unicode 码相互转化的文件，移植就是了。

ffconf.h 是配置文件，选择里面的宏配置文件系统支持的功能

```

/*
 * FatFs - FAT file system module configuration file R0.11a (C)ChaN, 2015
 */

#define _FFCONF_ 64180 /* Revision ID */

/*
 * Function Configurations
 */

#define _FS_READONLY 0 /* This option switches read-only configuration. (0:Read/Write or 1:Read-only)
/* Read-only configuration removes writing API functions, f_write(), f_sync(),
f_unlink(), f_mkdir(), f_chmod(), f_rename(), f_truncate(), f_getfree()
and optional writing functions as well. */

#define _FS_MINIMIZE 0 /* This option defines minimization level to remove some basic API functions
/* 0: All basic functions are enabled.
/* 1: f_stat(), f_getfree(), f_unlink(), f_mkdir(), f_chmod(), f_utime(),
f_truncate() and f_rename() function are removed.

#define _CODE_PAGE 932 /* This option specifies the character page to be used on the target sys
/* Incorrect setting of the code page can cause a file open failure.
/*
/* 1 - ASCII (No extended character. Non-LFN cfg. only)
/* 437 - U.S.
/* 720 - Arabic
/* 737 - Greek
/* 771 - KBL
/* 775 - Baltic
/* 850 - Latin 1
/* 852 - Latin 2
/* 855 - Cyrillic
/* 857 - Turkish
/* 860 - Portuguese
/* 861 - Icelandic
/* 862 - Hebrew
/* 863 - Canadian French
/* 864 - Arabic
/* 865 - Nordic
/* 866 - Russian
/* 869 - Greek 2
/* 932 - Japanese (DBCS)
/* 936 - Simplified Chinese (DBCS)
/* 949 - Korean (DBCS)
/* 950 - Traditional Chinese (DBCS)
*/

```

这个置 1 就是让文件系统只有只读功能，比如我的 norflash 已经烧写好固定数据了，该产品只需要读取 norflash 里面的数据，那么我这里可以设置成只读功能，那么编译出来的程序就要小很多，就没有必要把写入的功能打开，否则会增加程序大小。

这个文件系统是支持中文，日文，还是英文，这里默认是日文。到时候记得改成中文

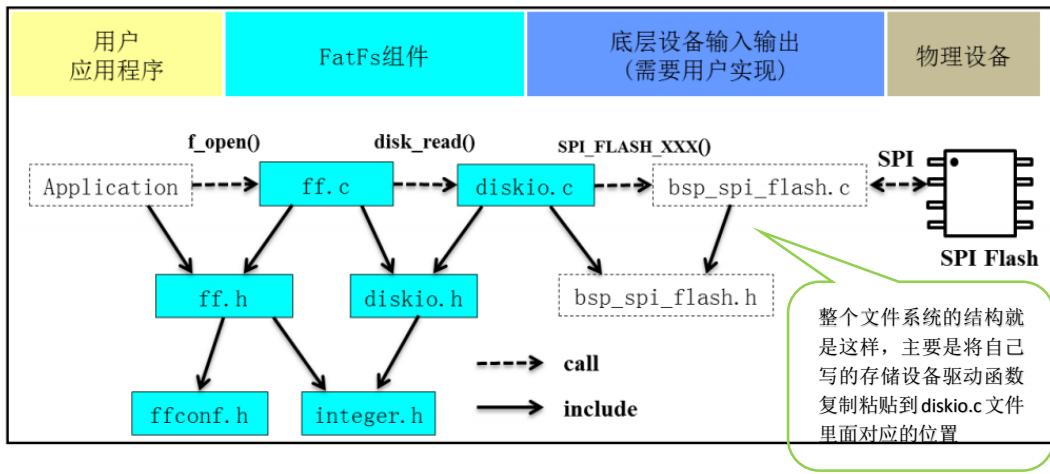
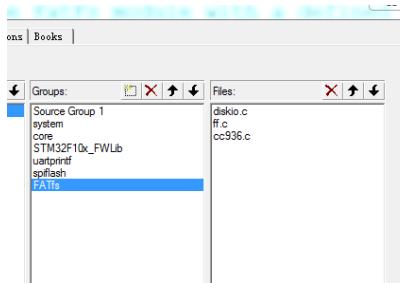


图 26-5 FatFs 程序结构图

开始移植 FATfs 文件系统



将 src 目录的文件系统内容加入进工程

进入 diskio.c 去修改里面的内容

```

10 #include "diskio.h"      /* FatFs lower layer API */
11 #include "usbdisk.h"     /* Example: Header for USB disk */
12 #include "atadrive.h"    /* Example: Header for ATA drive */
13 #include "sdcard.h"      /* Example: Header for SD card */

```

将不属于该平台的三个头文件去掉

```

10 #include "diskio.h"      /* FatFs lower layer API */
11
12
13 /* Definitions of physical drive number for each drive */
14 #define ATA      0      /* Example: Map ATA harddisk to ph */

```

只剩下 diskio.h

```

3 DSTATUS disk_status (
4     BYTE pdrv      /* Physical drive nmuber to ic */
5 )
6 {
7     DSTATUS stat;
8     int result;
9
10    switch (pdrv) {
11        case ATA :
12            result = ATA_disk_status();
13
14            // translate the reslut code here
15
16            return stat;
17
18        case MMC :
19            result = MMC_disk_status();
20
21            // translate the reslut code here
22
23            return stat;
24
25        case USB :
26            result = USB_disk_status();
27
28    }

```

这是其它存储设备的状态化函数，我们是 norflash，所以这三个函数不要

三个函数屏蔽掉了

```

DSTATUS disk_initialize (
    BYTE pdrv           /* Physical drive number to */
)
{
    DSTATUS stat;
    int result;

    switch (pdrv) {
    case ATA :
        result = ATA_disk_initialize();
        // translate the reslut code here

        return stat;

    case MMC :
        result = MMC_disk_initialize();
        // translate the reslut code here

        return stat;

    case USB :
        result = USB_disk_initialize();
    }

DRESULT disk_read (
    BYTE pdrv,          /* Physical drive number to identify the c
    BYTE *buff,         /* Data buffer to store read data */
    DWORD sector,       /* Sector address in LBA */
    UINT count          /* Number of sectors to read */
)
{
    DRESULT res;
    int result;

    switch (pdrv) {
    case ATA :
        // translate the arguments here
        result = ATA_disk_read(buff, sector, count);
        // translate the reslut code here

        return res;

    case MMC :
        // translate the arguments here
        result = MMC_disk_read(buff, sector, count);
    }

DRESULT disk_write (
    BYTE pdrv,          /* Physical drive number to identif
    const BYTE *buff,   /* Data to be written */
    DWORD sector,       /* Sector address in LBA */
    UINT count          /* Number of sectors to write */
)
{
    DRESULT res;
    int result;

    switch (pdrv) {
    case ATA :
        // translate the arguments here
        result = ATA_disk_write(buff, sector, count);
        // translate the reslut code here

        return res;

    case MMC :
        // translate the arguments here
        result = MMC_disk_write(buff, sector, count);
    }

DWORD get_fattime (void)
{
    /*这个地方写RTC时钟驱动,然后读取时间返回给上层*/
    return 0; //这里应该返回时间参数
}

```

在 diskio.c 最后一行增加标准的获取

时间函数，函数名必须是这样。

现在 diskio.c 不用的函数删除完了

在 cc936.c 文件里面将#error 取消掉

| #error This file is not needed in current configuration. Remove from the project

因为#error 不是我们编译器能识别的

这下全部编译通过

分析移植新代码带 diskio.c

```
#define ATA      0 /* Example: Map ATA harddisk to physical drive 0 */
#define MMC      1 /* Example: Map MMC/SD card to physical drive 1 */
#define USB      2 /* Example: Map USB MSD to physical drive 2 */

27     DSTATUS stat;
28     int result;

29     switch (pdrv) {
30 case ATA :
31     //result = ATA_disk_status();
32
33     // translate the reslut code here
34
35     return stat;
36
37 case MMC :
38     //result = MMC_disk_status();
39
40     // translate the reslut code here
41
42     return stat;
43
44 case USB :
45     // ...
```

这三个宏的编号可以自己定义，但是自定义的编号要记住，因为在应用层 open 的时候会先输入编号来指定我现在数据存入哪一个设备，比如 open(1:text) 意思就是 text 文件存在 MMC/SD 卡的里面。

```
/* Definitions of physical drive number for each drive */
//#define ATA      0 /* Example: Map ATA harddisk to physical drive 0 */
//#define MMC      1 /* Example: Map MMC/SD card to physical drive 1 */
//#define USB      2 /* Example: Map USB MSD to physical drive 2 */
#define SPI_NORFLASH 1

11     case 0 :
12     //result = ATA_disk_status();
13
14     // translate the reslut code here
15
16     return stat;
17
18 case SPI_NORFLASH :
19     //result = MMC_disk_status();
20
21     // translate the reslut code here
22
23     return stat;
24
25 case 2 :
26     //result = USB_disk_status();
27
28     // translate the reslut code here
29
30     return stat;
31 }
32 return STA_NOINIT;
```

我们的 norflash 设备我也定义为 1，到时候 open 的时候记得写 1....就可以了

我们修改 disk_status, disk_initialize, disk_read, disk_write
这些函数的 switch 编号，改成我 flash 的编号，其余没用的
变化写数字

这样就编译通过了

```
#include "diskio.h"      /* FatFs lower layer API */
#include "spi_flash.h"
#include "norflash.h"
```

在 diskio.c 里面添加 SPI 驱动和 norflash 头文件以便 diskio.c 使用

```

i2 DSTATUS disk_initialize (
i3     BYTE pdrv           /* Physical drive number to
i4 )
i5 {
i6     DSTATUS stat;
i7     int result;
i8
i9     switch (pdrv) {
i0         case 0 :
i1             //result = ATA_disk_initialize();
i2
i3             // translate the reslut code here
i4
i5             return stat;
i6
i7         case SPI_NORFLASH :
i8             spi_FUN_init(); //SPI初始化
i9
i0             return stat;
i1
i2         case 2 :
i3             //从这里开始是 norflash 初始化位置

```

现在有个问题，就是 `disk_initialize` 要求返回一个值表示 norflash 初始化成功，我们要怎么返回呢？

```

void SPI_Flash_Read_ID(void)
{
    uint32_t flash_ID = 0, subID = 0;

    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA, GPIO_Pin_2); //CS=0
    SPI_Send_Read_Byte(JEDEC_ID);
    flash_ID = SPI_Send_Read_Byte(0x00);

    subID = SPI_Send_Read_Byte(0x00);
    subID = subID << 8;
    subID |= SPI_Send_Read_Byte(0x00);
    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1

    flash_ID = ((flash_ID << 16) | subID);
    printf("FLASH ID = %x\r\n", flash_ID);
}

```

在 `diskio.c` 文件的 `disk_initialize` 函数里面找到对应的 `SPI_NORFLASH` 初始化位置，复制初始化 SPI 驱动代码过来

这个是我上上章节写的测试 norflash 的函数，如果返回 norflash ID，就证明 stm32 和 norflash 通信成功，我现在将这个函数改成有返回值的函数

FLASH ID = ef4017
FLASH ID = ef4017

返回 `ef4017` 表示成功

```

uint32_t SPI_Flash_Read_ID(void)
{
    uint32_t flash_ID = 0, subID = 0;

    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1
    GPIO_ResetBits(GPIOA, GPIO_Pin_2); //CS=0
    SPI_Send_Read_Byte(JEDEC_ID);
    flash_ID = SPI_Send_Read_Byte(0x00); //返回 norflash 的 ID，把这个函数复制到 diskio.c 的 disk_status
    subID = SPI_Send_Read_Byte(0x00);
    subID = subID << 8;
    subID |= SPI_Send_Read_Byte(0x00);
    GPIO_SetBits(GPIOA, GPIO_Pin_2); //CS=1
    flash_ID = ((flash_ID << 16) | subID);
    printf("FLASH ID = %x\r\n", flash_ID);
    return flash_ID;
}

DSTATUS disk_status (
    BYTE pdrv           /* Physical drive number to identify */
)
{
    DSTATUS stat;
    int result;

    switch (pdrv) {
        case 0 :
            //result = ATA_disk_status();
            // translate the reslut code here
            return stat;
        case SPI_NORFLASH :
            if(SPI_Flash_Read_ID() == 0xef4017)
            {
                //norflash通信正常
                stat = 0;
            }
            else
            {
                //norflash通信不正常
                stat = STA_NOINIT;
            }
    }
    return stat;
}

```

然后将这个 `disk_status` 放在 `diskio.c` 的 `disk_initialize` 函数下的 `SPI_NORFLASH` 位置

为什么要把 norflash 通信状态写在状态函数里面，而不写在 `disk_initialize` 初始化函数里面还简单些呢？

```

switch (pdrv) {
case 0 :
    //result = ATA_disk_status();
    // translate the reslut code here
    return stat;

case SPI_NORFLASH :
    if(SPI_Flash_Read_ID() == 0xef4017)
    {
        //norflash通信正常
        stat = 0;
    }
    else
    {
        //norflash通信不正常
        stat = STA_NOINIT;
    }
    //stat |= STA_PROETCT //flash通信正常，只是flash硬件设置了写保护

```

这是因为有些 flash 通信是正常的，只是被硬件设置了写保护，或者其它功能，那么在使用过程中的时候就在状态函数里面加这些功能是很理想的。我应用层想读取 flash 的状态执行 status 函数就是了，不用每次读 flash 状态都要去初始化 SPI 接口一次

```

DSTATUS disk_initialize (
    BYTE pdrv           /* Physical drive nmuber to identify the drive */
)
{
    DSTATUS stat;
    int result;

    switch (pdrv) {
    case 0 :
        //result = ATA_disk_initialize();

        // translate the reslut code here
        return stat;

    case SPI_NORFLASH :
        spi_FUN_init(); //SPI初始化
        return disk_status(SPI_NORFLASH); //传入SPI_NORFLASH节点，表示初始化的是NORFLASH
        return stat;

    case 2 :
        //result = NSR_disk_initialize();

```

将状态函数写入 disk_initialize 函数里面，一定要保证先是执行的 SPI 接口初始化，然后再去用 disk_status 读取 flash ID

初始化 norflash 是否成功会返回给应用层

```

    DRESULT disk_read (
        BYTE pdrv,          /* Physical drive nmuber to identify the drive */
        BYTE *buff,         /* Data buffer to store read data */
        DWORD sector,       /* Sector address in LBA */
        UINT count          /* Number of sectors to read */
    )

    DRESULT res;
    int result;

    switch (pdrv) {
    case 0 :
        // translate the arguments here
        //result = ATA_disk_read(buff, sector, count);

        // translate the reslut code here
        return res;

    case SPI_NORFLASH :
        READ_norflash_sector(sector*4096,buff,count*4096); //norflash每个扇区数4096字节
        res = RES_OK; //默认flash读取数据正常
        return res;

```

/*读取flash扇区里面的内容*/
/*addr:要读取flash扇区的首地址
* readbuff:将读取的一个扇区4096字节写入buff,
* byte_number:读取多少个字节
*/
void READ_norflash_sector(uint32_t addr,uint8_t *readbuff,uint32_t byt

将 flash 读函数写入这里

指定扇区首地址，如果是 1 扇区，那么首地址就要*4096，因为 4096 前面是 0 扇区，4096 字节后是 1 扇区，2 扇区就是 2*4096

这是要写多少个扇区，norflash 每个扇区数 4096

这一个扇区 4096 是 norflash，可能其他型号 flash 一个扇区不止 4096.到时候就到修改这里

```

DRESULT disk_write (
    BYTE pdrv,           /* Physical drive number to identify the drive */
    const BYTE *buff,     /* Data to be written */
    DWORD sector,         /* Sector address in LBA */
    UINT count            /* Number of sectors to write */
)
{
    DRESULT res;
    int result;

    switch (pdrv) {
    case 0 :
        // translate the arguments here

        //result = ATA_disk_write(buff, sector, count);

        // translate the result code here

        return res;

    case SPI_NORFLASH :
        WRITE_norflash_multisector(sector*4096, (uint8_t *)buff, count*4096);
        //这里和disk_read函数一样，只是buff要强转一下，其实uint8_t和BYTE都是8位的意思
        res = RES_OK; //默认flash写数据正常
        return res;

    case 2 :

```

最后要在 ioctl 函数里面实现 norflash 设备的查询命令，比如容量，大小之类的

函数	条件(ffconf.h)	备注
disk_status		
disk_initialize		
disk_read		
disk_write	_FS_READONLY == 0	底层设备驱动函数
get_fattime		
disk_ioctl (CTRL_SYNC)		
disk_ioctl (GET_SECTOR_COUNT)	_USE_MKFS == 1	
disk_ioctl (GET_BLOCK_SIZE)		
disk_ioctl (GET_SECTOR_SIZE)	_MAX_SS != _MIN_SS	
disk_ioctl (CTRL_TRIM)	_USE_TRIM == 1	
ff_convert	_USE_LFN != 0	Unicode 支持，为支持简体中文，添加 cc936.c 到工程即可
ff_wtoupper		
ff_cre_syncobj		
ff_del_syncobj		FATFs 可重配置，需要多任务系统支持（一般不需要）
ff_req_grant		
ff_rel_grant		
ff_mem_alloc		长文件名支持，缓冲区设置在堆空间（一般设置 USE_LFN = 2）
ff_mem_free		

```

DRESULT disk_ioctl (
    BYTE pdrv,           /* Physical drive number (0...) */
    BYTE cmd,             /* Control code */
    void *buff            /* Buffer to send/receive control data */
)
{
    DRESULT res;

```

```

    switch (pdrv) {
    case 0 :

```

```

        return res;

```

```

    case SPI_NORFLASH :

```

```

        switch (cmd)
        {

```

```

            case GET_SECTOR_COUNT:
                *(DWORD *)buff = 2048;
                break;

```

```

            case GET_SECTOR_SIZE :
                *(WORD *)buff = 4096;
                break;

```

```

            case GET_BLOCK_SIZE :
                *(DWORD *)buff = 1;
                break;

```

GET_SECTOR_COUNT 命令返回总扇区数，根据手册 norflash 有 0~127 个 block(128 个块)，每个 block 有 16 个扇区，所以总扇区为 $128 \times 16 = 2048$ 个扇区

GET_SECTOR_SIZE 返回每个扇区字节数，norflash 一个扇区 4096 个字节

GET_BLOCK_SIZE 返回每次擦除扇区个数，这里不能写 4096，如果每次擦除一个扇区那就写 1，每次擦除 2 个扇区就写 2，我这里是 1

FATfs 文件系统配置

```
#include "ff.h" //使用文件系统必须添加这个头文件

FATFS fsp; //文件系统使用要先有这个句柄，这个句柄很大所以定义在全局变量

int main(void)
{
    FRESULT ret; //mount返回值，返回(0)FR_OK表示挂载成功

    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    delay_ms(5000);

    ret = f_mount(&fsp, "1:", 1); //挂载文件系统，将norflash挂载进文件系统
    printf("res = %d\r\n", ret);

    while(1);

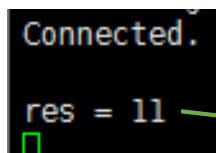
    return 0;
}
```

一个存储设备一个 FATFS 变量，如果是 SD 卡和 norflash 那么就要定义两个 FATFS 变量

第 3 个参数写 1 是快速挂载，写 0 是稍后挂载，反正我写 1

传入 FATFS 变量地址，“1”是指明挂载的哪一个存储设备

```
/* Definitions of physical disk */
#define ATA 0 /* E */
#define MMC 1 /* E */
#define USB 2 /* E */
#define SPI_NORFLASH 1
因为我的 norflash 指定的是分支 1 所以写“1”表示挂载 norflash
```



返回 11，表示挂载失败，提示逻辑驱动号无效

```
typedef enum {
    FR_OK = 0, /* (0) Succeeded */
    FR_DISK_ERR, /* (1) A hard error occurred in the low level disk I/O layer */
    FR_INI_ERR, /* (2) Assertion failed */
    FR_NOT_READY, /* (3) The physical drive cannot work */
    FR_NO_FILE, /* (4) Could not find the file */
    FR_NO_PATH, /* (5) Could not find the path */
    FR_INVALID_NAME, /* (6) The path name format is invalid */
    FR_DENIED, /* (7) Access denied due to prohibited access or directory */
    FR_EXIST, /* (8) Access denied due to prohibited access */
    FR_INVALID_OBJECT, /* (9) The file/directory object is invalid */
    FR_WRITE_PROTECTED, /* (10) The physical drive is write protected */
    FR_INVALID_DRIVE, /* (11) The logical drive number is invalid */
    FR_NOT_ENABLED, /* (12) The volume has no work area */
}
```

逻辑驱动号无效是因为我的 ffconf.h 配置问题

```
#define _VOLUMES 1
/* Number of volumes (logical drives) to be used. */
```

这里 VOLUMES 写的是最大支持 1 个存储设备

```
switch (pdv) {
    case 0:
        //result = ATA_disk_status();
        //translate the result code here
        return stat;

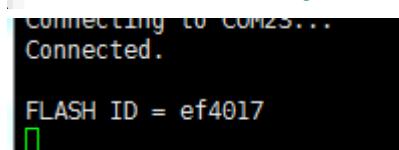
    case SPI_NORFLASH:
        if(SPI_Flash_Read_ID() == 0xef4017)
        {
            //norflash通信正常
            stat = 0;
        }
        else
        {
            //norflash通信不正常
            stat = STA_NOINIT;
        }
        //stat |= STA_PROTECT //flash通信正常，只是flash硬件设置了写保护
        return stat;

    case 2:
        //result = NOR_FLASH_initialize();
}
```

但是 diskio 设置的分支是三个设备，0,SPI_NORFLASH,2

```
#define _VOLUMES 3
/* Number of volumes (logical drives) to be used. */
```

我这里改成最大支持 3 个设备



修改后执行到设备初始化了，但是还是卡死了后面的 ret 没有打印出来

这种情况一般跟内存溢出有关系。

查询 ffconf.h，发现缓冲区大小太小，缓冲区大小要和 norflash 扇区大小一致

#define _MIN_SS	512
#define _MAX_SS	512

```
#define _MIN_SS      512
#define _MAX_SS      4096    修改后缓冲区和 norflash 扇区一样大了 4096 字节
```

```
Connected.

FLASH ID = ef4017
res = 13      FR_NO_FILESYSTEM, /* (13) There is no valid FAT volume */

/* (13) There is no valid FAT volume */
```

执行后发现 res 返回 13，表示这个设备上面不存在文件系统

这种原因是没有格式化 norflash



```
#define _USE_MKFS      0
/* This option switches f_mkfs */ 设置_USE_MKFS 为 1 才能使用格式化函数
```

在 ffconf.h 文件下修改

```
#define _USE_MKFS      1
/* This option switches f_m
```

FATFS fsp; //文件系统使用要先有这个句柄，这个句柄很大所以定义在全局变量

```
int main(void)
{
    FRESULT ret; //mount返回值，返回(0)FR_OK表示挂载成功

    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    delay_ms(5000);

    ret = f_mount(&fsp, "1:", 1); //挂载文件系统，将norflash挂载进文件系统
    printf("res = %d\r\n", ret);

    if (ret == FR_NO_FILESYSTEM) //如果norflash没有格式化，那就格式化
    {
        f_mkfs("1:", 0, 0);
    }
}
```

写入存储介质的盘符，然后格式化

```
while(1);
```

```
Connected.
FLASH ID = ef4017
res = 13
FLASH ID = ef4017
```

第一次执行开始格式化

```
FLASH ID = ef4017
res = 0
```

第二次执行格式化成功

这里有个关键的问题

```

int main(void)
{
    FRESULT ret; //mount返回值, 返回(0)FR_OK表示挂载成功

    RCC_configuration(); //初始化时钟
    USART_config(115200); //初始化串口
    delay_ms(5000);

    ret = f_mount(&fsp, "1:", 1); //挂载文件系统, 将norflash挂载进文件系统
    printf("res = %d\r\n", ret);

    if (ret == FR_NO_FILESYSTEM) //如果norflash没有格式化, 那就格式化
    {
        f_mkfs("1:", 0, 0);
        ret = f_mount(NULL, "1:", 1); //格式化完成后要取消挂载
        ret = f_mount(&fsp, "1:", 1); //然后再重新挂载才能使用
        printf("res2 = %d\r\n", ret);
    }
}

```

格式化后要先取消挂载再重新挂载, 取消挂载文件句柄写 NULL

现在测试读写文件进 norflash

```
FATFS fsp; //文件系统使用要先有这个句柄, 这个句柄很大所以定义在全局变量
FIL fp; //读写文件要创建文件句柄
```

```

int main(void)
{
    FRESULT ret; //mount返回值, 返回(0)FR_OK表示挂载成功
    uint32_t bytenum = 0;
    char INbuf[] = "123456,abcdefg"; //写入文件的数据
    char OUTbuf[30];

    RCC_configuration(); //初始化时钟
    USART_config(115200); //初始化串口
    delay_ms(5000);

    ret = f_mount(&fsp, "1:", 1); //挂载文件系统, 将norflash挂载进文件系统
    printf("res = %d\r\n", ret);

    if (ret == FR_NO_FILESYSTEM) //如果norflash没有格式化, 那就格式化
    {
        f_mkfs("1:", 0, 0);
        ret = f_mount(NULL, "1:", 1); //格式化完成后要取消挂载
        ret = f_mount(&fsp, "1:", 1); //然后再重新挂载才能使用
        printf("res2 = %d\r\n", ret);
    }

    ret = f_open(&fp, "1:abce.txt", FA_OPEN_ALWAYS | FA_WRITE | FA_READ); //打开文件
    printf("open ret = %d\r\n", ret);
    if (ret == FR_OK) //文件打开成功
    {
        printf("open file success\r\n");
        ret = f_write(&fp, INbuf, sizeof(INbuf), &bytenum);
        if (ret == FR_OK)
        {
            printf("write file success bytenum = %d\r\n", bytenum);
        }
        else
        {
            printf("write file failed\r\n");
        }
        f_lseek(&fp, 0); //文件光标从新移动到开头
        ret = f_read(&fp, OUTbuf, f_size(&fp), &bytenum); //读出文件的数据
        if (ret == FR_OK)
        {
            printf("read file success bytenum = %d\r\n", bytenum);
            printf("-----%s----\r\n", OUTbuf);
        }
        else
        {
            printf("read file failed\r\n");
        }
    }
    f_close(&fp); //这个一定不要忘了, 如果不调用close可能导致文件不被保存
    while(1);
}

```

1. 文件句柄要有

2. 挂载文件

3. 创建文件, "1:abce.txt"
意思就是在 norflash 根目录下创建 abce.txt 文件
FA_OPEN_ALWAYS 表示如目录下没有同名的文件, 就创建该文件, FA_WRITE 和 FA_READ 是读写权限, 必须要写

```

Connecting to com2...
Connected.

FLASH ID = ef4017
res = 0
FLASH ID = ef4017
open ret = 0
open file success
FLASH ID = ef4017
write file success bytelenum = 15
FLASH ID = ef4017
FLASH ID = ef4017
read file success bytelenum = 15
-----123456,abcdefg-----
FLASH ID = ef4017
FLASH ID = ef4017

```

读写测试成功

FATfs 文件系统支持中文和长文件名

第 1 个问题长文件名

我想创建一个长文件名的文件

```

ret = f_open(&fp, "1:aaaaaaaaaaaaaaaaaaaa.txt", FA_OPEN_ALWAYS|FA_WRITE|FA_READ); // 打开文件

printf("open ret = %d\r\n", ret);
if(ret == FR_OK) // 文件打开成功
{
    Connected.

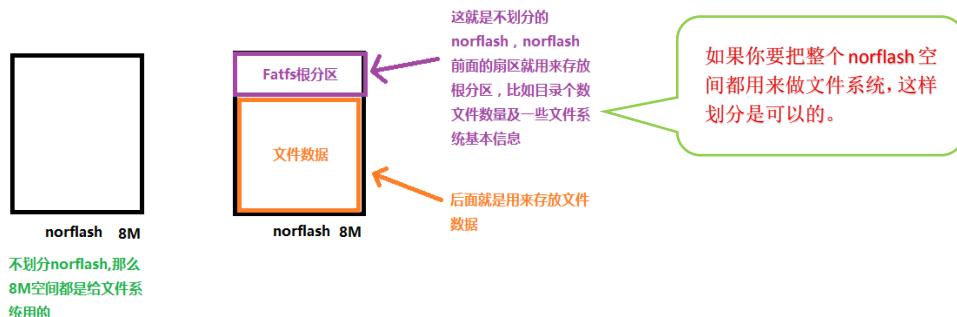
    FLASH ID = ef4017
    res = 0
    FLASH ID = ef4017
    open ret = 6
}

```

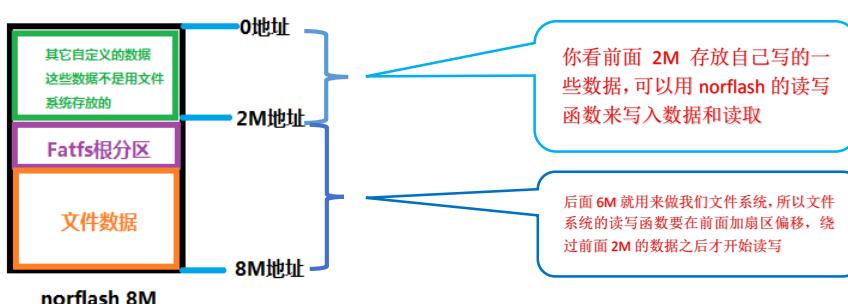
文件打开失败，返回数字 6
FR_INVALID_NAME, /* (6) The path name format is invalid */
6 的意思就是文件名无效

ret = f_open(&fp, "1:文件系统文件创建.txt", FA_OPEN_ALWAYS|FA_WRITE|FA_READ); // 打开文件
创建中文的文件名也是不支持的，同样返回数字 6，文件名无效

在解决中文名和长文件名支持之前，先对文件系统进行划分



但是有一天发现我要用 Fatfs 文件系统建立的文件及数据用不了 8M 这么多，可能 6M 就够了，那么我们就可以把 norflash 后 6M 用来做文件系统，前面 2M 用来存放其它的数据



所以我们要修改前面 diskio.c 里面的内容

1K = 1024字节

1M = 1024K

1M = 1024字节 X 1024K (1024 * 1024) = 1048576字节

1048576(1M) X 6 = 6291456字节(1M)

这 6M 的字节占用 norflash 多少个扇区呢？我们现在的 W25Q64 norflash 一个扇区数 4096 字节，所以 $6291456 / 4096 = 1536$ 个扇区

```
DRESULT disk_ioctl (
    BYTE pdrv,      /* Physical drive number (0..) */
    BYTE cmd,        /* Control code */
    void *buff       /* Buffer to send/receive control data */
)
{
    DRESULT res;

    switch (pdrv) {
        case 0 :
            return res;
        case SPI_NORFLASH :
            switch (cmd) {
                case GET_SECTOR_COUNT:
                    *(DWORD *)buff = 2048;
                    break;
                case GET_SECTOR_SIZE :
                    *(WORD *)buff = 4096;
                    break;
                case GET_BLOCK_SIZE :
                    *(DWORD *)buff = 1;
                    break;
            }
    }
}
```

```
DRESULT disk_ioctl (
    BYTE pdrv,      /* Physical drive number (0..) */
    BYTE cmd,        /* Control code */
    void *buff       /* Buffer to send/receive control data */
)
{
    DRESULT res;

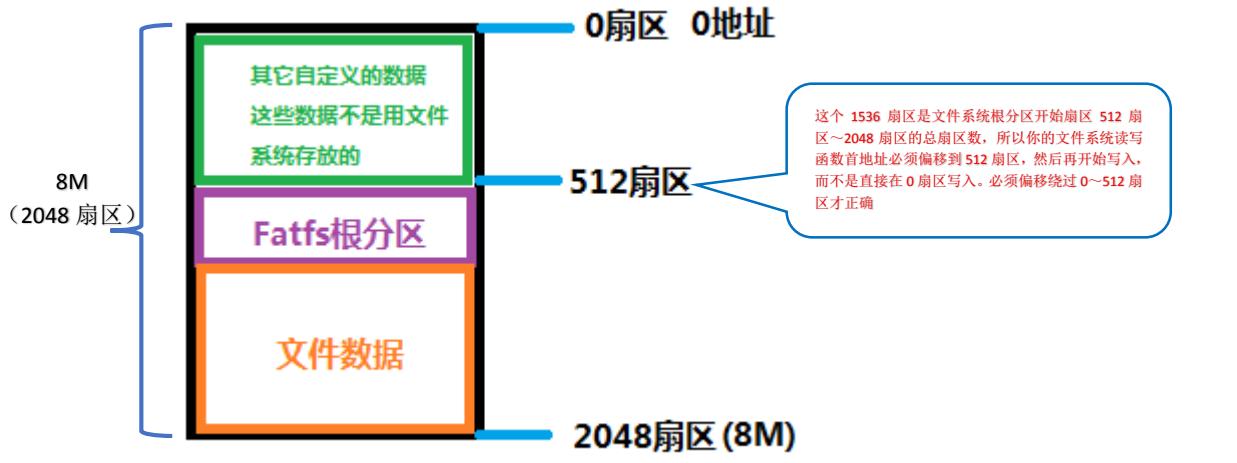
    switch (pdrv) {
        case 0 :
            return res;
        case SPI_NORFLASH :
            switch (cmd) {
                case GET_SECTOR_COUNT:
                    *(DWORD *)buff = 1536; //1536个扇区 6M空间
                    break;
                case GET_SECTOR_SIZE :
                    *(WORD *)buff = 4096;
                    break;
                case GET_BLOCK_SIZE :
                    *(DWORD *)buff = 1;
                    break;
            }
    }
}
```

这是以前占用 8M 空间的扇区数

现在改成 6M 空间占用的扇区数

Norflash一个扇区 4096 字节，这些都不用修改

这里有个关键点要注意



```
#if USE_WRITE
DRESULT disk_write (
    BYTE pdrv,          /* Physical drive number to identify the drive */
    const BYTE *buff,   /* Data to be written */
    DWORD sector,       /* Sector address in LBA */
    UINT count          /* Number of sectors to write */
)
{
    DRESULT res;

    switch (pdrv) {
        case 0 :
            // translate the arguments here
            //result = ATA_disk_write(buff, sector, count);
            // translate the result code here
            return res;
        case SPI_NORFLASH :
            sector = sector+512;//偏移2M字节 就是2097152字节/4096(一个扇区4096字节) = 512扇区
            WRITE_norflash_multisector(sector*4096,(uint8_t *)buff,count*4096);
            //这里和disk_read函数一样，只是buff要强转一下，其实uint8_t和BYTE都是8位的意思
            res = RES_OK;//默认flash写数据正常
    }
}
```

修改写函数，比如上层传下来 sector 是 0 扇区，我们就让其偏移 512 个扇区，也就是 0 地址偏移 2097152 个地址，这些指针就指在了后面 6M 的第一个扇区地址，因为这里才是文件系统的开始位置。如果上层传入的是 2, 5, 6 扇区，同样都是便宜 512 扇区

```

DRESULT disk_read (
    BYTE pdrv,      /* Physical drive number to identify the drive */
    BYTE *buff,     /* Data buffer to store read data */
    DWORD sector,   /* Sector address in LBA */
    UINT count      /* Number of sectors to read */
)
{
    DRESULT res;

    switch (pdrv) {
    case 0 :
        // translate the arguments here
        //result = ATA_disk_read(buff, sector, count);
        // translate the result code here
        return res;

    case SPI NORFLASH :
        sector = sector+512; //偏移2M
        READ_norflash_sector(sector*4096,buff,count*4096); //norflash每个扇区数4096
        res = RES_OK; //默认flash读取数据正常
        return res;
    }
}

```

读函数也要同样便宜这么多

这样 norflash 的文件系统划分就搞定了。其余的 EMMC,NANDFLASH, SD 卡这些存储介质用的 FATFS 时，文件系统也可以这样划分。

现在设置长文件名支持

在 ffconf.h 文件里面修改

```

4 #define _CODE_PAGE 936
5 /* This option specifies the
6 / Incorrect setting of the
7 /
8 / 1 - ASCII (No extended)
9 / 437 - U.S.
0 / 720 - Arabic
1 / 737 - Greek

```

确保 _CODE_PAGE 选择 936 支持中文编码

```

93 #define _USE_LFN 0
94 #define _MAX_LFN 255
95 /* The _USE_LFN option switches the LFN feature.
96 /
97 / 0: Disable LFN feature. _MAX_LFN has no effect.
98 / 1: Enable LFN with static working buffer on the BSS. Always NOT thread-safe.
99 / 2: Enable LFN with dynamic working buffer on the STACK.
00 / 3: Enable LFN with dynamic working buffer on the HEAP.

```

这个地方不能为 0，必须有大于 0 的数字，文件系统才能支持长文件名

写 1 把长文件名放在全局区
写 2 把长文件名放在栈区
写 3 把长文件名放在堆区

```

#define _USE_LFN 1
#define _MAX_LFN 255

```

我选择长文件名放在全局区，全局区大一些。

然后 _MAX_LEN 255 表示文件名长度最多可以写到 255 个字节

```

ret = f_open(&fp,"1:文件系统文件创建.txt",FA_OPEN_ALWAYS|FA_WRITE|FA_READ); //打开文件
printf("open ret=%d\r\n",ret);
if(ret==FR_OK)//文件打开成功
{
    printf("open file success\r\n");
    ret=f_write(&fp,INbuf,sizeof(INbuf),&bytelen);
    if(ret==FR_OK)
    {
        Connecting to COM23...
Connected.

FLASH ID = ef4017
res = 0
FLASH ID = ef4017
open ret = 0
open file success
FLASH ID = ef4017
write file success bytelen = 15
FLASH ID = ef4017
FLASH ID = ef4017
read file success bytelen = 15
-----123456abcdefg-----
FLASH ID = ef4017
FLASH ID = ef4017

```

你看长文件名的 txt 文件就能支持了，而且支持中文文件名

FATfs 常用函数操作

1.flash 容量大小读取

f_getfree 获取 flash 总扇区数和空闲扇区数

FATFS fsp; //文件系统使用要先有这个句柄，这个句柄很大所以定义在全局变量

```
int main(void)
{
    FATFS *fat;
    FRESULT ret; //mount返回值，返回(0)FR_OK表示挂载成功
    uint32_t bytenum = 0;
    DWORD flash_size = 0, total_sector = 0, free_sector = 0;

    RCC_configuration(); //初始化时钟
    USART_config(115200); //初始化串口
    delay_ms(5000);

    ret = f_mount(&fsp, "1:", 1); //挂载文件系统，将norflash挂载进文件系统
    if(ret == FR_OK)
        printf("fatfs mount success \r\n");
    else
        printf("fatfs mount failed \r\n");

    if(ret == FR_OK)
        printf("fatfs mount success \r\n");
    else
        printf("fatfs mount failed \r\n");

    if(ret == FR_NO_FILESYSTEM) //如果norflash没有格式化，那就格式化
    {
        f_mkfs("1:", 0, 0);
        ret = f_mount(NULL, "1:", 1); //格式化完成后要取消挂载
        ret = f_mount(&fsp, "1:", 1); //然后再重新挂载才能使用
        printf("formatting fatfs\r\n");
    }
    f_getfree("1:", &flash_size, &fat);
    total_sector = (fat->n_fatent - 2) * fat->cszie; //计算flash总扇区数
    free_sector = flash_size * fat->cszie; //计算flash空扇区数也就是可用的空间
    printf("flash total sector = %lu KB\r\n", total_sector * 4); //总扇区输出
    printf("flash free total = %lu KB\r\n", free_sector * 4); //可用扇区输出

    //这里乘以4意思就是一个扇区是4KB，现在得到的是扇区个数，然后*4就是总容量单位是KB，如果是字节单位就不能*4
}
```

这个变量得到当前 flash 剩余多少个簇
传入一个文件系统指针，得到全局变量 fsp 的确切地址，地址放入 fat
n_fatent 就是 簇的数量+2
cszie 一个簇等于多少个扇区，返回扇区数
这句话就是空余簇的数目 × 每个簇的扇区数 = 剩余的扇区数

2. f_printf 使用

首先在 ffconf.h 里面打开 _USE_STRFUNC 宏

```
#define _USE_STRFUNC 1
/* This option switches string functions, f_gets(), f_putc(), f_puts() and
   f_printf().
int main(void)
{
    FRESULT ret; //mount返回值，返回(0)FR_OK表示挂载成功
    uint32_t bytenum = 0;
    char readbuff[50];

    RCC_configuration(); //初始化时钟
    USART_config(115200); //初始化串口
    delay_ms(5000);

    ret = f_mount(&fsp, "1:", 1); //挂载文件系统，将norflash挂载进文件系统
    if(ret == FR_OK)
        printf("fatfs mount success \r\n");
    else
        printf("fatfs mount failed \r\n");

    if(ret == FR_NO_FILESYSTEM) //如果norflash没有格式化，那就格式化
    {
        f_mkfs("1:", 0, 0);
        ret = f_mount(NULL, "1:", 1); //格式化完成后要取消挂载
        ret = f_mount(&fsp, "1:", 1); //然后再重新挂载才能使用
        printf("formatting fatfs\r\n");
    }
}
```

和上面的 fatfs 案例一样，先挂载文件系统

```

ret = f_open(&fp, "1:print.txt", FA_OPEN_ALWAYS | FA_WRITE | FA_READ); // 创建个文件
if(ret == FR_OK)
{
    printf("print.txt file open success\r\n");
}
else
{
    printf("print.txt file open failed\r\n");
}

ret = f_lseek(&fp, 0); // 不管文件有没有之前写入数据, 文件光标都移动到文件头

// f_lseek(&fp, f_size(&fp));
// 这是计算文件被写过的数据大小, 然后将光标移动到文件写过的字符串尾部, 然后接着写
if(ret == FR_OK)
{
    f_printf(&fp, "write str to success "); // 写字符串给文件
    f_printf(&fp, "write variable to file = %d\r\n", 20); // 写变量的数据到文件
}
else
{
    printf("file lseek failed\r\n");
}
ret = f_lseek(&fp, 0);
// 写完数据后, 如果要读取文件内容就要把光标重新移动到开始处, 当然也可以用 f_close 来解决
if(ret == FR_OK)
{
    ret = f_read(&fp, readbuff, f_size(&fp), &bytenum);
    printf("fp file read number = %d\r\n", bytenum);
    printf("-----%s-----\r\n", readbuff);
}
else
{
    printf("read lseek failed\r\n");
}
f_close(&fp);

Connected.

FLASH ID = ef4017
fatfs mount success
FLASH ID = ef4017
print.txt file open success
FLASH ID = ef4017
fp file read number = 50
-----write str to success write variable to file = 20-----
FLASH ID = ef4017
FLASH ID = ef4017

```

所以用 `f_write` 和 `f_printf` 操作其实差不多, 看自己需求和习惯选择性使用

3.f_stat 获取文件信息

挂载文件系统程序是必不可少的, 这里就省略了

准备一个存放文件信息的结构体变量

```

FILINFO info; // 获取文件信息的结构体变量, 文件信息获取后放在info里面
ret = f_stat("1:print.txt", &info);
if(ret == FR_OK)
{
    printf("file size = %ld\r\n", info.fsize); // 获取文件大小
    printf("file date = %u \r\n", info.fdate); // 获取文件时间
}
else
{
    printf("file info get failed\r\n");
}
```

`fsiz` 获取文件大小, `fdate` 获取文件创建时间或者修改时间

这个获取的时间要做移位运算, 因为我这里没有写 RTC 底层函数, 所以就不做运算, 网上找资料

Connected.

```

FLASH ID = ef4017
fatfs mount success
FLASH ID = ef4017
file size = 50
file date = 0

```

4.norflash 文件内容详细搜索

```
FRESULT scan_files(char* path) /* Start node to be scanned (***a*/
{
    FRESULT res;
    DIR dir;
    UINT i;
    static FILINFO fno;

    res = f_opendir(&dir, path); /* Open the directory */
    if (res == FR_OK) {
        for (;;) {
            res = f_readdir(&dir, &fno); /* Read a directory item */
            if (res != FR_OK || fno.fname[0] == 0) break; /* Break on error or end of dir */
            if (fno.fattrib & AM_DIR) { /* It is a directory */
                i = strlen(path);
                sprintf(&path[i], "%s", fno.fname);
                res = scan_files(path); /* Enter the directory */
                if (res != FR_OK) break;
                path[i] = 0;
            } else { /* It is a file. */
                printf("%s/%s\n", path, fno.fname);
            }
        }
        f_closedir(&dir);
    }
    return res;
}
```

然后在主函数下调用

```
ret = f_mount(&fsp, "1:", 1); //挂载文件系统,将norflash挂载进文件系统
if(ret == FR_OK)
    printf("fatfs mount success \r\n");
else
    printf("fatfs mount failed \r\n");

if(ret == FR_NO_FILESYSTEM) //如果norflash没有格式化,那就格式化
{
    f_mkfs("1:", 0, 0);
    ret = f_mount(NULL, "1:", 1); //格式化完成后要取消挂载
    ret = f_mount(&fsp, "1:", 1); //然后再重新挂载才能使用
    printf("Formatting fatfs\r\n");
}

scan_files("1:"); //传入磁盘编号, norflash我们设置的是1, 搜索文件打印出来

while(1);

Connecting to COM23...
Connected.

FLASH ID = ef4017
fatfs mount success
FLASH ID = ef4017
FLASH ID = ef4017
1:/tobe~1.TXT
1:/print.txt
FLASH ID = ef4017
FLASH ID = ef4017
```

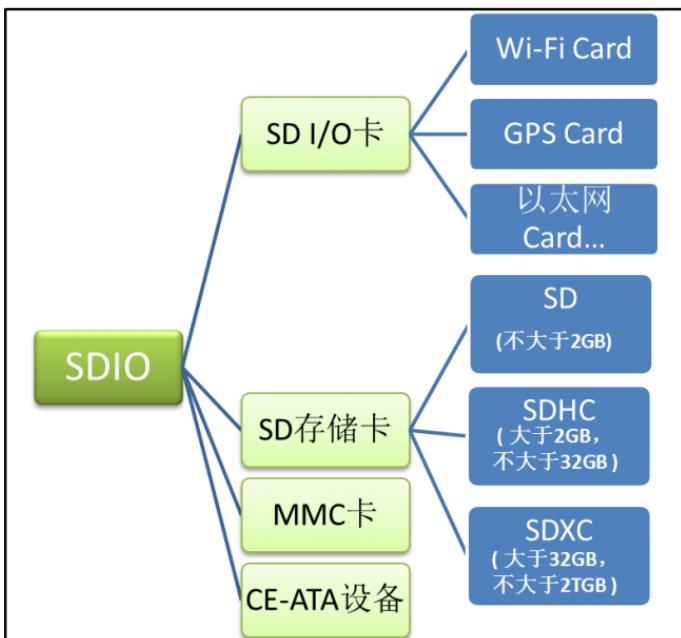
这就是 flash 存储的文件, 我这里只有两个文件, 有个是中文文件, 所以打印出来是乱码。

定义了一个文件搜索的子函数, 函数完整内容在下面贴出

这就是完整的搜索文件函数实现,
如果你的 flash 里面文件很多, 记得去修改 STM32 栈空间大小, 因为这里面有递归调用

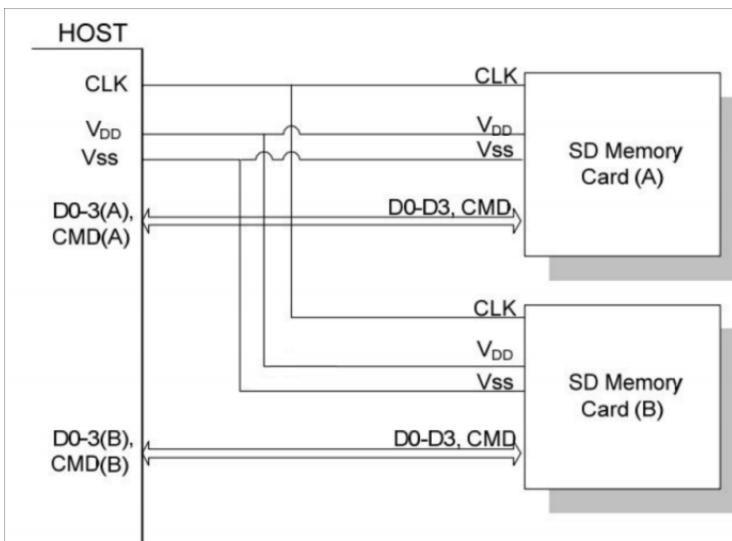
前面文件系统初始化都一样, 这里只需要传入你的 flash 盘符编号, 就可以搜索出你的 flash 存储了哪些文件

SDIO 接口驱动 SD 卡



SDIO 接口支持的设备，SD 有 SD

规格分为 SDHC 规格的，SDXC 规格的，规格不一样容量就不一样传输速度也不一样。



理论上 SDIO 可以串行连接多个 SD 卡或者其它 SD 接口的设备，但是 STM32 的 SDIO 只支持连接一个设备。

SDIO 不管是从主机控制器向 SD 卡传输，还是 SD 卡向主机控制器传输都只以 CLK 时钟线的上升沿为有效。SD 卡操作过程会使用两种不同频率的时钟同步数据，一个是识别卡阶段时钟频率 FOD，最高为 400kHz，另外一个是数据传输模式下时钟频率 FPP，默认最高为 25MHz，如果通过相关寄存器配置使 SDIO 工作在高速模式，此时数据传输模式最高频率为 50MHz。

名称	bit 宽度	描述
CID	128	卡识别号(Card identification number):用来识别的卡的个体号码(唯一的)
RCA	16	相对地址(Relative card address):卡的本地系统地址，初始化时，动态地由卡建议，主机核准。
DSR	16	驱动级寄存器(Driver Stage Register):配置卡的输出驱动
CSD	128	卡的特定数据(Card Specific Data):卡的操作条件信息
SCR	64	SD 配置寄存器(SD Configuration Register):SD 卡特殊特性信息
OCR	32	操作条件寄存器(Operation conditions register)
SSR	512	SD 状态(Status):SD 卡专有特征的信息
CSR	32	卡状态(Card Status):卡状态信息

这个是 SD 卡全球唯一 ID

这个是 SDIO 总线如果挂接了多张 SD 卡，那么主机给每个卡分配的地址

这是 SD 卡里面的几个寄存器

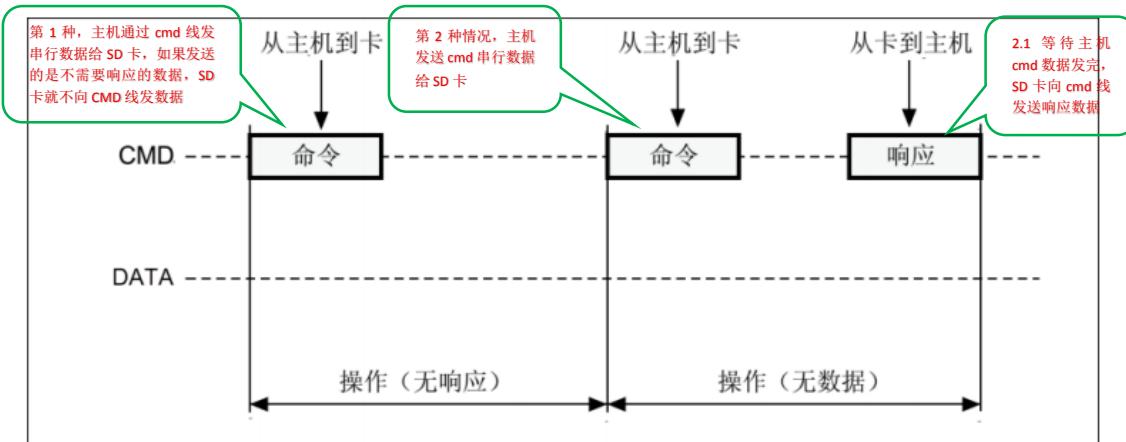
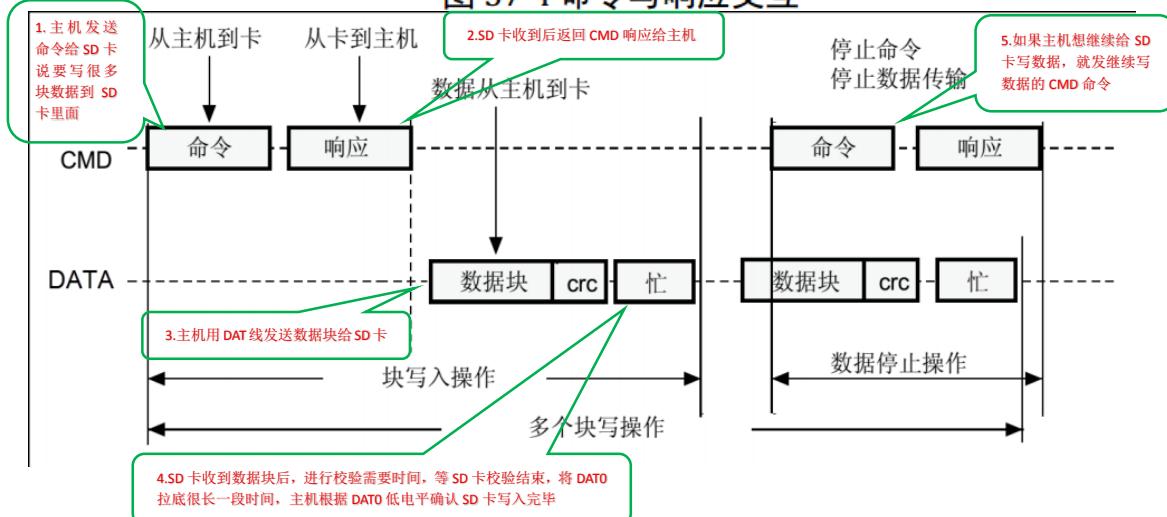


图 37-4 命令与响应交互



SD 卡数据包传输方式

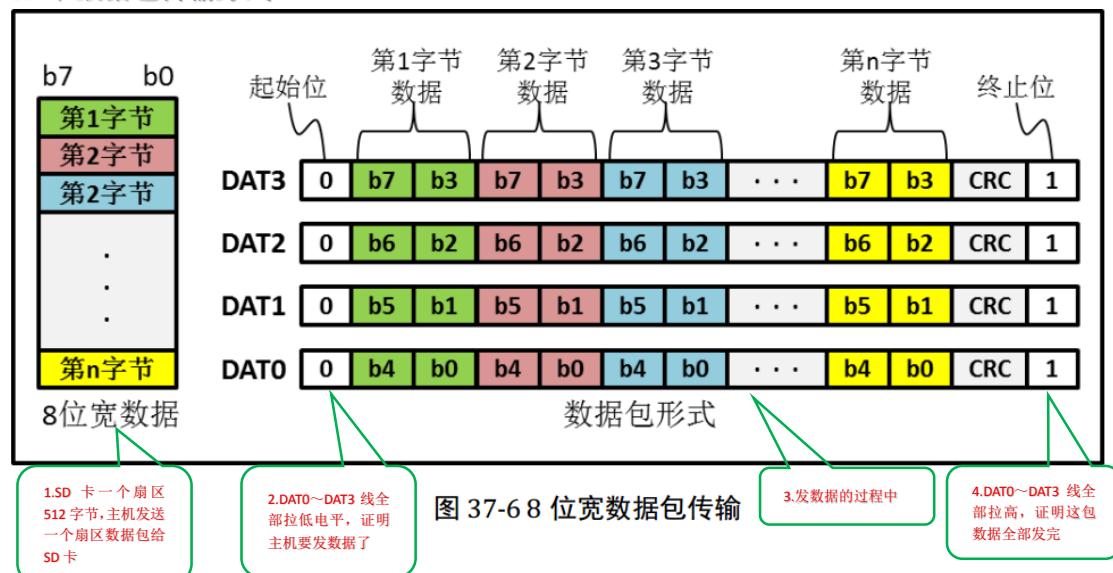


图 37-6 8 位宽数据包传输

还有一种特殊的数据包，叫宽位数据包

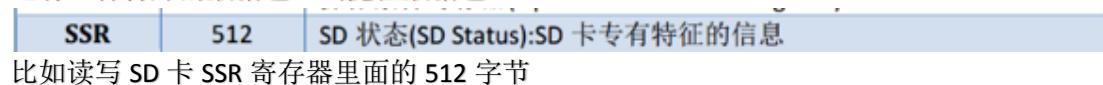




图 37-7 宽位数据包传输

代码分析

```

typedef struct {
    uint32_t SDIO_ClockEdge;           // 时钟沿
    uint32_t SDIO_ClockBypass;         // 旁路时钟
    uint32_t SDIO_ClockPowerSave;      // 节能模式
    uint32_t SDIO_BusWide;             // 数据宽度
    uint32_t SDIO_HardwareFlowControl; // 硬件流控制
    uint8_t SDIO_ClockDiv;             // 时钟分频
} SDIO_InitTypeDef;

```

```

#define IS_SDIO_CLOCK_EDGE(EDGE)
#define SDIO_ClockEdge_Falling ((uint32_t)0x00000000)
#define SDIO_ClockEdge_Rising ((uint32_t)0x00000001)

```

CLK 时钟是上升沿采集数据还是下降沿采集，我们选择 SDIO_ClockEdge_Rising 上升沿采集

```

#define IS_SDIO_CLOCK_BYPASS(BYPASS)
#define SDIO_ClockBypass_Disable ((uint32_t)0x00000000)
#define SDIO_ClockBypass_Enable ((uint32_t)0x00000001)

```

我们选择失能，
SDIO_ClockBypass_Disable，不需要时钟直通，需要对输入时钟进行分频后输出给 CLK

CLK 平时在不传输数据的时候也会不停的发送时钟，但是我选择 SDIO_ClockPowerSave_Enable 节能模式之后，CLK 只有在发送数据的时候才会产生时钟，比较省电。当然节能模式下开启 CLK 要慢一些，但是不会慢很多所以可以接受。

```

#define IS_SDIO_BUS_WIDE(WIDE)
#define SDIO_BusWide_1b ((uint32_t)0x00000000)
#define SDIO_BusWide_4b ((uint32_t)0x00000080)
#define SDIO_BusWide_8b ((uint32_t)0x00001000)

```

SD 卡数据总线是 DAT0~DAT3，是四根线，所以我们选择 SDIO_BusWide_4b

```

#define IS_SDIO_HARDWARE_FLOW_CONTROL(CONTROL)
#define SDIO_HardwareFlowControl_Disable ((uint32_t)0x00000000)
#define SDIO_HardwareFlowControl_Enable ((uint32_t)0x00004000)

```

打开 SDIO_HardwareFlowControl_Enable 使能，这样就不会出现 fifo 溢出

CLK 线时钟频率=SDIOCLK/([CLKDIV+2])。

SDIOCLK 在 STM32F1 系列里面是 72MHz 和主时钟一致，我们的 SD 卡 CLK 只需要 24M 时钟频率，那么我们就计算 CLKDIV 的值，然后写入 SDIO_ClockDiv，所以我们这里写 CLKDIV=1

STM32 发送命令给设备的代码,就是通过 CMD 引脚发送串行数据

```
typedef struct {
    uint32_t SDIO_Argument; // 命令参数
    uint32_t SDIO_CmdIndex; // 命令号
    uint32_t SDIO_Response; // 响应类型
    uint32_t SDIO_Wait; // 等待使能
    uint32_t SDIO_CPSM; // 命令路径状态机
} SDIO_CmdInitTypeDef;
```

命令序号	类型	参数	响应	缩写	描述
基本命令(Class 0)					
CMD0	bc	[31:0]填充位	-	GO_IDLE_STATE	复位所有的卡到 idle 状态。
CMD2	bcr	[31:0]填充位	R2	ALL_SEND_CID	通知所有卡通过 CMD 线返回 CID 值。
CMD3	bcr	[31:0]填充位	R6	SEND_RELATIVE_ADDR	通知所有卡发布新 RCA。
CMD4	bc	[31:16]DSR[15:0] 填充位	-	SET_DSR	编程所有卡的 DSR。
CMD7	ac	[31:16]RCA[15:0] 填充位	R1b	SELECT/DESELECT_CARD	选择/取消选择 RCA 地址卡。

```
#define SDIO_Response_Long ((uint32_t)0x000000C0)
#define SDIO_Response_No ((uint32_t)0x00000000)
#define SDIO_Response_Short ((uint32_t)0x00000040)
```

主机 SDIO 发送命令和参数给 SD 卡后,等待 SD 卡响应,这里就看是要等待 SD 卡什么响应,有长响应,短响应,无响应

```
#define IS_SDIO_WAIT(WAIT)
#define SDIO_Wait_1T ((uint32_t)0x00000100)
#define SDIO_Wait_No ((uint32_t)0x00000000)
#define SDIO_Wait_Pend ((uint32_t)0x00000020)
```

如果你主机要等待 SD 卡的长响应,短响应这些,你得打开等待使能,如果你不打开等待使能,那么你会忽略 SD 卡的响应

```
#define IS_SDIO_CPSM(CPSM) (((CPSM) == SDIO_CPSM_Enable) || ((CPSM) == SDIO_CPSM_Disable))
#define SDIO_CPSM_Disable ((uint32_t)0x00000000)
#define SDIO_CPSM_Enable ((uint32_t)0x00000040)
```

打开命令状态机 SDIO_CPSM_Enable

STM32 发送数据给设备的代码,就是通过 DAT0~DAT3 引脚发送串行数据

```
typedef struct {
    uint32_t SDIO_DataTimeOut; // 数据传输超时
    uint32_t SDIO_DataLength; // 数据长度
    uint32_t SDIO_DataBlockSize; // 数据块大小
    uint32_t SDIO_TransferDir; // 数据传输方向
    uint32_t SDIO_TransferMode; // 数据传输模式
    uint32_t SDIO_DPSM; // 数据路径状态机
} SDIO_DataInitTypeDef;
```

设置发送一次的数据长度,比如一次发送 1024, 4096.....字节

```
#define IS_SDIO_TRANSFER_DIR(DIR)
#define SDIO_TransferDir_ToCard ((uint32_t)0x00000000)
#define SDIO_TransferDir_ToSDIO ((uint32_t)0x00000002)
```

数据传输方向,是数据从 SD 卡发往主机,还是主机发往 SD 卡

```

#define IS_SDIO_BLOCK_SIZE(SIZE)
#define SDIO_DataBlockSize_1024b ((uint32_t)0x000000A0)
#define SDIO_DataBlockSize_128b ((uint32_t)0x00000070)
#define SDIO_DataBlockSize_16384b ((uint32_t)0x000000E0)
#define SDIO_DataBlockSize_16b ((uint32_t)0x00000040)
#define SDIO_DataBlockSize_1b ((uint32_t)0x00000000)
#define SDIO_DataBlockSize_2048b ((uint32_t)0x000000B0)
#define SDIO_DataBlockSize_256b ((uint32_t)0x00000080)
#define SDIO_DataBlockSize_2b ((uint32_t)0x00000010)
#define SDIO_DataBlockSize_32b ((uint32_t)0x00000050)
#define SDIO_DataBlockSize_4096b ((uint32_t)0x000000C0)
#define SDIO_DataBlockSize_4b ((uint32_t)0x00000020)
#define SDIO_DataBlockSize_512b ((uint32_t)0x00000090)
#define SDIO_DataBlockSize_64b ((uint32_t)0x00000060)
#define SDIO_DataBlockSize_8192b ((uint32_t)0x000000D0)
#define SDIO_DataBlockSize_8b ((uint32_t)0x00000030)

```

设置 SD 卡一个块是多少字节

```

#define IS_SDIO_TRANSFER_MODE(MODE)
#define SDIO_TransferMode_Block ((uint32_t)0x00000000)
#define SDIO_TransferMode_Stream ((uint32_t)0x00000004)

```

SDIO_TransferMode 就是发送数据块给 SD 卡, SDIO_TransferMode_Stream 是发送数据流给 SD 卡, 我们这里选择块传输

SD 卡读写测试, 代码工程内容

_htmresc
 Libraries
 Project
 Utilities
 stm32f10x_stdperiph_lib_um.chm

因为 SD 卡协议很复杂, 我们都是移植 ST 官方的标准库

测试 SD 卡程序在 Project/STM32F10x_StdPeriph_Examples/SDIO/uSDCard 这个目录的 main 文件

```

#include "stm32f10x.h"
#include "stm32_eval_sdio_sd.h"

int main(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
    this is done through SystemInit() function which is called from startup
    file (startup_stm32f10x_xx.s) before to branch to application main.
    To reconfigure the default setting of SystemInit() function, refer to
    system_stm32f10x.c file
    */

    /* Initialize LEDs available on STM3210X-EVAL board *****/
    STM_EVAL_LEDInit(LED1);
    STM_EVAL_LEDInit(LED2);
    STM_EVAL_LEDInit(LED3);
    STM_EVAL_LEDInit(LED4);

    /* Interrupt Config */
    NVIC_Configuration();

    /*----- SD Init -----*/
    if((Status = SD_Init()) != SD_OK)
    {
        STM_EVAL_LEDOn(LED4);
    }
}

```

这个测试 SD 卡的 C 文件还调用了, stm32_eval_sdio_sd.h 驱动 SD 卡协议的程序

而且在 stm32f10x_it.h 里面还加入了 SDIO 中断处理函数

main.c
 stm32f10x_it.c
 system_stm32f10x.c
 stm32f10x_conf.h
 stm32f10x_it.h
 readme.txt

Utilities/STM32_EVAL/Common/stm32_eval_sdio_sd.c 这是驱动文件的目录路径

```
#include "stm32_eval.h"
```

我们发现 stm32_eval_sdio_sd.c 文件还调用了
stm32_eval.c 文件里面的底层驱动函数

Utilities/STM32_EVAL/STM3210E_EVAL/stm3210e_eval.c 这是底层驱动文件的目录

```
#include "stm3210e_eval.h"
#include "stm32f10x_spi.h"
#include "stm32f10x_i2c.h"
#include "stm32f10x_sdio.h"
#include "stm32f10x_dma.h"

void SD_LowLevel_DeInit(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /*!< Disable SDIO Clock */
    SDIO_ClockCmd(DISABLE);

    /*!< Set Power State to OFF */
    SDIO_SetPowerState(SDIO_PowerState_OFF);

    /*!< DeInitializes the SDIO peripheral */
}
```

stm3210e_eval.c 这个文件就是调用固件
库里面 SDIO 的寄存器函数了，还调用了 stm32_eval.c 里面的内容。

所以移植文件的时候，我们要保证如下：

1. 固件库的 stm32f10x_sdio.c/h 文件包含进工程

2. Utilities/STM32_EVAL/Common/stm32_eval_sdio_sd.c 文件包含进工程

3. 注意因为 stm32_eval_sdio_sd.c 要调用 stm3210e_eval.h 里面的函数，所以 stm3210e_eval.c/h 要加入进工程，文件在 Utilities\STM32_EVAL\STM3210E_EVAL/stm3210e_eval.c

4. Utilities/STM32_EVAL/stm32_eval.c 包含进工程

```
SDIO_firmware
  +-- stm32_eval_sdio_sd.c
  +-- stm32_eval.c
  +-- stm3210e_eval.c
```

包含进工程后目录是这样的

编译过程中会报错

```
#elif defined USE_STM3210E_EVAL
  #include "stm3210e_eval/stm3210e_eval.c"
```

```
.\out\SDcard_driver.axf: Error: L6200E: Symbol SD DMAEndOfTransferStatus multiply defined (by stm3210e_eval.o and stm32_eval.o).
.\out\SDcard_driver.axf: Error: L6200E: Symbol SD_LowLevel_DMA_RxConfig multiply defined (by stm3210e_eval.o and stm32_eval.o).
.\out\SDcard_driver.axf: Error: L6200E: Symbol SD_LowLevel_DMA_TxConfig multiply defined (by stm3210e_eval.o and stm32_eval.o).
.\out\SDcard_driver.axf: Error: L6200E: Symbol SD LowLevel DeInit multiply defined (by stm3210e_eval.o and stm32_eval.o).
```

这两个错误一起解决

```
///#ifdef USE_STM3210B_EVAL
/// #include "stm32f10x.h"
/// #include "stm3210b_eval/stm3210b_eval.h"
///#elif defined USE_STM3210B_EVAL
/// #include "stm32f10x.h"
/// #include "stm3210b_eval/stm3210b_eval.h"
///#elif defined USE_STM3210E_EVAL
/// #include "stm32f10x.h"
///##include "stm3210e_eval.h"
///#elif defined USE_STM3210C_EVAL
/// #include "stm32f10x.h"
/// #include "stm3210c_eval/stm3210c_eval.h"
///#elif defined USE_STM32L152_EVAL
/// #include "stm32l1xx.h"
/// #include "stm32l152_eval/stm32l152_eval.h"
///#elif defined USE_STM32100E_EVAL
/// #include "stm32f10x.h"
/// #include "stm32100_eval/stm32100e_eval.h"
///#else
/// #error "Please select first the STM32 EVAL board to be used (in stm32_eval.h)"
///#endif
```

将 stm32_eval.c/h 的 include 包含的头文件全部屏蔽就可以了。红框是造成重复包含的主要原因，这是 ST 官方例程的坑！！

我们主要使用的是 `stm32_eval_sdio_sd.c` 里面的函数

在主函数文件实现 SD 卡测试代码，根据测试代码去寻找 `stm32_eval_sdio_sd.c` 里面的函数

```
void SDIO_NVIC_Configuration(void) //在firmware的C文件唯独没有初始化SDIO中断优先级，所以这里先初始化
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Configure the NVIC Preemption Priority Bits */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

    NVIC_InitStructure.NVIC_IRQChannel = SDIO_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

void SD_test()
{
    SDIO_NVIC_Configuration();
    if((Status = SD_Init()) != SD_OK)
    {
        printf("SD卡初始化失败\r\n");
    }
    else
    {
        printf("SD卡初始化成功\r\n");
    }
}
```

这是主程序 C 文件

```
SD_Error SD_Init(void) ←
{
    SD_Error errorstatus = SD_OK;

    /* SDIO Peripheral Low Level Init */
    SD_LowLevel_Init(); ← 这是初始化 SDIO 的 GPIO 管脚为 SD 功能，而且打开管脚时钟和 DMA2 时钟，所以我说 ST 官方例程写的好呢！
    SDIO_DeInit(); ← 因为这里居然不写 SDIO 中断优先级初始化函数
    errorstatus = SD_PowerON(); ← 把所有 SDIO 寄存器设置为初始化默认值

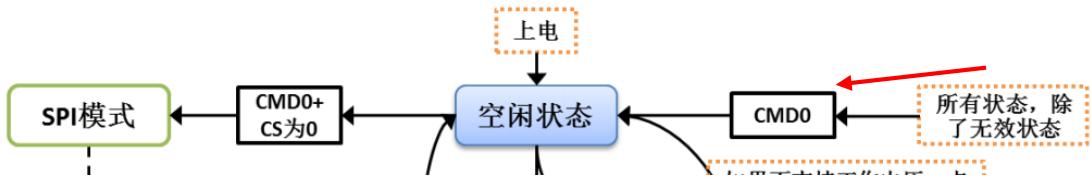
    if (errorstatus != SD_OK)
    {
        /*!< CMD Response TimeOut (wait for CMDSENT flag) */
        return(errorstatus);
    }

    errorstatus = SD_InitializeCards();

    if (errorstatus != SD_OK)
        ← 这是 stm32_eval_sdio_sd.c 文件
    SD_Error SD_PowerON(void) ←
    {
        SD_Error errorstatus = SD_OK;
        uint32_t response = 0, count = 0, validvoltage = 0;
        uint32_t SDType = SD_STD_CAPACITY; ← 先暂时默认设置成 2G 以下的 SD 卡

        /*!< Power ON Sequence ----- */
        /*!< Configure the SDIO peripheral */
        /*!< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_INIT_CLK_DIV) */
        /*!< on STM32F2xx devices, SDIOCLK is fixed to 48MHz */
        /*!< SDIO_CK for initialization should not exceed 400 KHz */
        SDIO_InitStructure.SDIO_ClockDiv = SDIO_INIT_CLK_DIV; ← SDIO_INIT_CLK_DIV=0xb2(十进制 178)
        SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
        SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;
        SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;
        SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b; ← 72M/180=400K，这里我计算的是 f103 的时钟
        SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_Disable;
        SDIO_Init(&SDIO_InitStructure); ← 其余设置前面初始化结构体已经说得很明白了，这里先用 1 位数据线来识别 SD 卡版本
    }
```

这函数还是在 `stm32_eval_sdio_sd.c` 文件里面实现的



```
/*!< CMD0: GO_IDLE_STATE -----*/
/*!< No CMD response required */
SDIO_CmdInitStructure.SDIO_Argument = 0x0;
SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_GO_IDLE_STATE;
SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_No;
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);
```

发送 CMD0 成功后

errorstatus = CmdError();

if (errorstatus != SD_OK)

{

/*!< CMD Response TimeOut (wait for CMDSENT)
return(errorstatus);
}

发送 CMD0 命令
#define SD_CMD_GO_IDLE_STATE ((uint8_t)0)
SD 卡的 CMD0 是不需要参数的, 所以 SDIO_Argument 写 0 就是

检查命令是否正常
发送出去给 SD 卡

表 37-2 SD 部分命令描述

命令序号	类别	参数	响应	缩写	描述
CMD0	bcr	[31:0]填充位	-	GO_IDLE_STATE	基本命令 (Class 0) 复位所有的卡到 idle 状态。

因为 SD 卡响应列没有相应数据, 所以选择
SDIO_Response_No 无响应
这个 CMD0 命令就是复位 SD 卡

发送 CMD8

```
/*!< CMD8: SEND_IF_COND -----*/
/*!< Send CMD8 to verify SD card interface operating condition */
/*!< Argument: - [31:12]: Reserved (shall be set to '0')
   - [11:8]: Supply Voltage (VHS) 0x1 (Range: 2.7-3.6 V)
   - [7:0]: Check Pattern (recommended 0xAA) */
/*!< CMD Response: R7 */
SDIO_CmdInitStructure.SDIO_Argument = SD_CHECK_PATTERN;
SDIO_CmdInitStructure.SDIO_CmdIndex = SDIO_SEND_IF_COND;
SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);

errorstatus = CmdResp7Error();
```

CMD8 命令有 R7, 所以有响应, 设置短响应

如果响应是 R7, 所以执行 Resp7 函数来检查是否有响应, 有响应就是 V2.0 的 SD 卡

所以把 CardType 卡类型设置成 2.0 版本

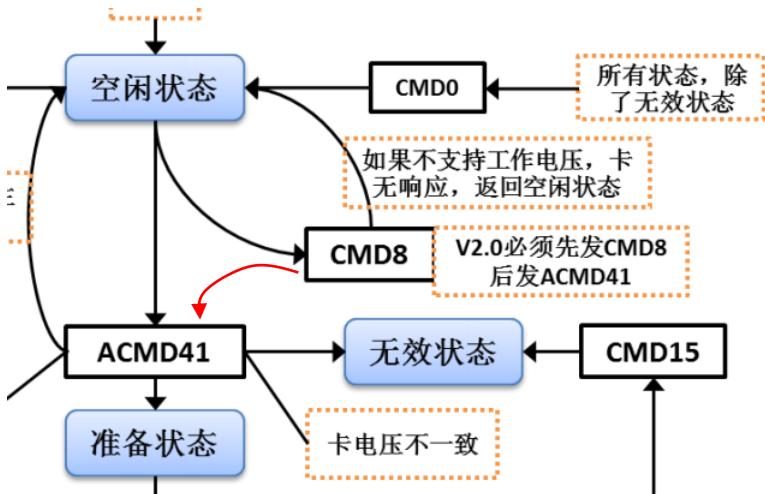
上电
空闲状态
所有状态, 除了无效状态
如果支持工作电压, 卡无响应, 返回空闲状态
CMD8
V2.0 必须先发 CMD8 后发 ACMD41
ACMD41
无效状态
CMD15

```
CardType = SDIO_STD_CAPACITY_SD_CARD_V2_0; /*!< SD Card 2.0 */
SDType = SD_HIGH_CAPACITY;
```

设置成高容量卡

```
/*!< CMD55 */
SDIO_CmdInitStructure.SDIO_Argument = 0x00;
SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);
errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
```

我下面要发送 ACMD 命令, 所以必须先发送 CMD55, 这样 CMD55 后面跟的 CMD 命令就是 ACMD 命令



```

SDIO_CmdInitStructure.SDIO_Argument = SD_VOLTAGE_WINDOW_SD | SDType;
SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SD_APP_OP_COND;
SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);

```

发送 ACMD41 命令

以上是 SD_PowerON 代码

```

errorstatus = SD_PowerON();
if (errorstatus != SD_OK)
{
    /*!< CMD Response TimeOut (wait for CMDSENT flag) */
    return(errorstatus);
}

errorstatus = SD_InitializeCards();
if (errorstatus != SD_OK)
{
    /*!< CMD Response TimeOut (wait for CMDSENT flag) */
    return(errorstatus);
}

/*!< Configure the SDIO peripheral */
/*!< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_TRANSFER_CLK_DIV) */
/*!< on STM32F2xx devices, SDIOCLK is fixed to 48MHz */
SDIO_InitStructure.SDIO_ClockDiv = SDIO_TRANSFER_CLK_DIV;
SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;
SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;

```

这是 STM32F2 系列的
时钟分频参数，这个
参数要改，因为我是
STM32F1 系列

SDIO_InitStructure.SDIO_ClockDiv = SDIO_TRANSFER_CLK_DIV; //这个参数分频出来是 48M
#define SDIO_TRANSFER_CLK_DIV ((uint8_t)0x00)

因为 SDIO_TRANSFER_CLK_DIV 是 0x00，我们要将其改成 0x01

#define SDIO_TRANSFER_CLK_DIV ((uint8_t)0x01)

这样 $72M/3 = 24M$ ，这才符合 STM32F1 系列的 SDIO 时钟

这就把最先的 400K，SDIO 时钟频率改成了 24M 时钟频率，可以开始读写数据块了。

```

-
if (errorstatus == SD_OK)
{
    /*----- Read CSD/CID MSD registers -----*/
    errorstatus = SD_GetCardInfo(&SDCardInfo);
}

if (errorstatus == SD_OK)
{
    /*----- Select Card -----*/
    errorstatus = SD_SelectDeselect((uint32_t) (SDCardInfo.RCA << 16));
}

if (errorstatus == SD_OK)
{
    errorstatus = SD_EnableWideBusOperation(SDIO_BusWide_4b);
}

```

这是将 SD 卡容量大小，一些基本信息放在 SDCardInfo 结构体中，你可以去查询这个结构体来获取 SD 卡大小

初始化 SD 卡发送命令方式就是这种流程，后面的代码自己分析了。

下面分析下 SDIO 读写数据块的代码

下面介绍 stm32_eval_sdio_sd.c 驱动文件里面的一些我们常常使用的函数
然后用这些驱动函数去读写 SD 卡

写 SD 卡单个扇区

```

SD_Error SD_WriteBlock(uint8_t *writebuff, uint32_t WriteAddr, uint16_t BlockSize)
// writebuff 要写入 SD 卡的数据，传入你申请的数组
// WriteAddr 写入 SD 卡的起始地址
// BlockSize 写入 SD 卡多少个块，也就是数据大小

```

```

SD_Error SD_WaitReadOperation(void)

```

//写完 SD 卡扇区后要执行这个函数等待一下，SD 卡数据写入需要时间，所以这里等待 SD 卡写完

```

void SD_SingleBlockTest(void)
{
    /*----- Block Read/Write -----*/
    /* Fill the buffer to send */
    uint16_t index = 0;
    for (index = 0; index < 512; index++)
    {
        Buffer_Block_Tx[index] = 0x32; 我们放一些测试数据在 buffer 里面
    }

    if (Status == SD_OK)
    {
        /* Write block of 512 bytes on address 0 */
        Status = SD_WriteBlock(Buffer_Block_Tx, 0x00, BLOCK_SIZE);
        /* Check if the Transfer is finished */
        Status = SD_WaitWriteOperation(); 等待 SD 卡写完
        while (SD_GetStatus() != SD_TRANSFER_OK); 判断 SD 卡是否写入成功
    }

    if (Status == SD_OK)
    {
        /* Read block of 512 bytes from address 0 */
        Status = SD_ReadBlock(Buffer_Block_Rx, 0x00, BLOCK_SIZE);
        /* Check if the Transfer is finished */
        Status = SD_WaitReadOperation(); 等待 SD 卡读写完毕
        while (SD_GetStatus() != SD_TRANSFER_OK); 如果写入成功我们再把数据从 SD 卡读出来
    }

    /* Check the correctness of written data */
    index=512;
    while (index--) 检查 SD 卡单块读写是否成功
    {
        if (Buffer_Block_Tx[index] != Buffer_Block_Rx[index])
        {
            printf("SD卡单块读写失败\r\n");
        }
    }
    printf("SD卡单块读写成功\r\n");
}

```

下面我们先对 SD 卡的单块读写进行测试

记住必须加入 SDIO 中断服务程序，否则你只能初始化 SD 卡成功，读写不成功

```

void SDIO_IRQHandler(void)
{
    SD_ProcessIRQSrc();
}

```

在主函数文件加入中断服务程序，当然你也可以在其他文件中加入

```

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    delay_ms(5000);
    printf("SD test start\r\n");
    SD_Test();
    SD_SingleBlockTest();
    printf("SD test end\r\n");

    while(1);
    return 0;
}

```

SD test start
 SD init success
 SD single block write success
 SD test end

SD 卡测试成功

SD 卡多个扇区(多块)读写测试

```
SD_Error SD_WriteMultiBlocks(uint8_t *writebuff, uint32_t WriteAddr, uint16_t BlockSize, uint32_t NumberOfBlocks)
// writebuff 写入数据的缓冲区，也就是数组
// WriteAddr 写入数据的起始地址
// BlockSize 写入数据的块大小，也就是一块多少个字节，我这里 SD 卡是 1 块=512 字节
// NumberOfBlocks 要写多少个块
void SD_MultiBlockTest(void)
{
    /*----- Multiple Block Read/Write -----*/
    /* Fill the buffer to send */

    uint16_t index = 512*32;//512是SD卡一个块(扇区)多少字节，32是要写多少个块

    for (index = 0; index < (512*32); index++)
    {
        Buffer_MultiBlock_Tx[index] = 0x32;//Buffer_MultiBlock_Tx是上面定义的512*32大小的数组
    }
    if (Status == SD_OK)
    {
        /* Write multiple block of many bytes on address 0 */
        Status = SD_WriteMultiBlocks(Buffer_MultiBlock_Tx, 0x00, 512, 32);
        /* Check if the Transfer is finished */
        Status = SD_WaitWriteOperation();
        while(SD_GetStatus() != SD_TRANSFER_OK);
    }

    if (Status == SD_OK)
    {
        /* Read block of many bytes from address 0 */
        Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00, 512, 32);
        /* Check if the Transfer is finished */
        Status = SD_WaitReadOperation();
        while(SD_GetStatus() != SD_TRANSFER_OK);
    }

    /* Check the correctness of written data */
    if (Status == SD_OK)
    {
        index=512*32;
        while (index--)
        {
            if (Buffer_MultiBlock_Tx[index] != Buffer_MultiBlock_Rx[index])
            {
                printf("SD MultiBlock write failed \r\n");
            }
        }
        printf("SD MultiBlock write success\r\n");
    }
}
int main(void)
{
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口

    delay_ms(5000);
    printf("SD test start\r\n");
    SD_Test();
    SD_MultiBlockTest(); //SD卡多块写入测试
    printf("SD test end\r\n");

    while(1);

    return 0;
}
```

把自定义的数据写入缓存区

向 SD 卡写数据

等待 SD 卡延时操作和上面单扇区测试一样

读多块函数和写多块函数是一样的形参操作方式

判段 SD 卡操作是否正确

```
SD test start
SD init success
SD MultiBlock write success
SD test end
```

SD 卡多块写入成功

其实 SD 卡单扇区测试成功的话，多块测试一般都没有问题。

擦除 SD 卡扇区

```
SD_Erase(uint32_t startaddr, uint32_t endaddr)
//startaddr 是擦除扇区的起始地址, endaddr 是擦除扇区的结束地址
void SD_EraseTest(void)
{
    /*----- Block Erase -----*/
    uint16_t index = (512*32);
    if (Status == SD_OK)
    {
        /* Erase NumberOfBlocks Blocks of WRITE_BL_LEN(512 Bytes) */
        Status = SD_Erase(0x00, (512 * 32)); //擦除SD卡32个块空间
    }

    if (Status == SD_OK)
    {
        Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00, 512, 32);

        /* Check if the Transfer is finished */
        Status = SD_WaitReadOperation();

        /* Wait until end of DMA transfer */
        while(SD_GetStatus() != SD_TRANSFER_OK);
    }

    /* Check the correctness of erased blocks */
    if (Status == SD_OK)
    {
        while (index--)
        {
            /* In some SD Cards the erased state is 0xFF, in others it's 0x00 */
            if ((Buffer_MultiBlock_Rx[index] != 0xFF) && (Buffer_MultiBlock_Rx[index] != 0x00))
            {
                printf("Erase failed\r\n");
            }
        }
        printf("Erase success\r\n");
    }
}

int main(void)
{
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口

    delay_ms(5000);
    printf("SD test start\r\n");
    SD_Test();
    SD_EraseTest(); //擦除SD卡32个扇区, 返回0xff或者0x00都是擦除成功
    printf("SD test end\r\n");

    while(1);

    return 0;
}
```

```
SD test start
SD init success
Erase success
SD test end
```

SD 卡指定块数量擦除测试成功

SD 卡 FATFS 文件系统移植(这里涉及到两个存储设备)

我们建立个 C 文件将上一章调试好的 SD 卡底层驱动测试函数汇总到一起

我建立了一个 Sdcode 源文件来汇总 SD 卡底层驱动函数

```
#include "sdcode.h"

/* Private macro ----- */
/* Private variables ----- */
uint8_t Buffer_Block_Tx[512], Buffer_Block_Rx[512];
uint8_t Buffer_MultiBlock_Tx[512*32], Buffer_MultiBlock_Rx[512*32];
static SD_Error Status = SD_OK;//定义一个变量检查SD卡初始化状态
void SDIO_NVIC_Configuration(void)//在firmware的c文件唯独没有初始化SDIO中断优先级，所以这里先初始化
{
    NVIC_InitTypeDef NVIC_InitStructure;
    /* Configure the NVIC Preemption Priority Bits */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

    NVIC_InitStructure.NVIC_IRQChannel = SDIO_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
void SD_init()
{
    SDIO_NVIC_Configuration();
    if((Status = SD_Init()) != SD_OK)
    {
        printf("SD init failed \r\n");
    }
    else
    {
        printf("SD init success \r\n");
    }
}
void SD_EraseTest(void) //擦除函数不用移植
{
    /*----- Block Erase -----*/
    uint16_t index = (512*32);
    if (Status == SD_OK)
    {
        /* Erase NumberOfBlocks Blocks of WRITE_BL LEN(512 Bytes) */
        Status = SD_Erase(0x00, (512 * 32)); //擦除SD卡32个块空间
    }

    if (Status == SD_OK)
    {
        Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00, 512, 32);

        /* Check if the Transfer is finished */
        Status = SD_WaitReadOperation();

        /* Wait until end of DMA transfer */
        while(SD_GetStatus() != SD_TRANSFER_OK);
    }
}

void SD_MultiBlockTest(void) //单块读写测试函数不移植
{
    /*----- Multiple Block Read/Write -----*/
    /* Fill the buffer to send */

    uint16_t index = 512*32;//512是SD卡一个块(扇区)多少字节，32是要写多少个块

    for (index = 0; index < (512*32); index++)
    {
        Buffer_MultiBlock_Tx[index] = 0x32; //Buffer_MultiBlock_Tx是上面定义的512*32大小的数组
    }

    if (Status == SD_OK)
    {
        /* Write multiple block of many bytes on address 0 */
        Status = SD_WriteMultiBlocks(Buffer_MultiBlock_Tx, 0x00, 512, 32);
        /* Check if the Transfer is finished */
        Status = SD_WaitWriteOperation();
        while(SD_GetStatus() != SD_TRANSFER_OK);
    }

    if (Status == SD_OK)
    {
        /* Read block of many bytes from address 0 */
        Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00, 512, 32);
    }
}
```

这些变量都是测试 SD 卡底层用的，我们这里不移植

1.这个 SDIO 优先级函数我们要移植到 FATFS 文件系统初始化函数部分

2.这个 SD_init()里面有 SDIO 管脚和一些寄存器初始化，我们也移植到 FATFS 文件系统初始化函数部分

擦除函数不用移植

3.多块测试函数里面的写连续块函数要移植

4.多块测试函数里面的读连续块函数要移植

```

#ifndef __SDCODE_H
#define __SDCODE_H
#include "uartprintf.h"
#include "stdio.h"
#include "stm32_eval_sdio_sd.h"

#define BLOCK_SIZE      512 /* Block Size in Bytes */

#define NUMBER_OF_BLOCKS    32 /* For Multi Blocks operation
#define MULTI_BUFFER_SIZE (BLOCK_SIZE * NUMBER_OF_BLOCKS)

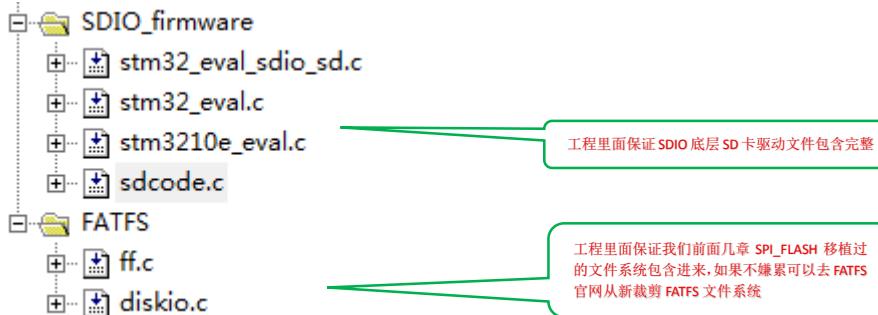
void SD_init(); //SD卡初始化
void SD_EraseTest(void);
void SD_SingleBlockTest(void);
void SD_MultiBlockTest(void);
void SDIO_NVIC_Configuration(void); //在firmware的C文件唯独没有初

#endif

```

这是驱动头文件

记得把 SDIO 中断服务函数也写在 sdcode.c 文件里面



主要修改 diskio.c

```

#include "stm32f10x.h"
#include "diskio.h" /* FatFs lower layer API */
#include "ff.h"
#include "spi_flash.h"
#include "norflash.h"
#include "stm32_eval_sdio_sd.h"
#include "sdcode.h"
#include <string.h>
/* Definitions of physical drive number for each drive */
// #define ATA     0 /* Example: Map ATA harddisk to physical drive 0 */
// #define MMC     1 /* Example: Map MMC/SD card to physical drive 1 */
// #define USB     2 /* Example: Map USB MSD to physical drive 2 */

#define SPI_NORFLASH 0
#define SDCard 1
extern SD_CardInfo SDCardInfo;
DSTATUS disk_status (
    BYTE pdrv /* Physical drive number to identify the drive */
)
{
    DSTATUS stat;
    SD_Error Status = STA_NOINIT; // 定义一个变量检查SD卡初始化状态

    switch (pdrv) {
        case 0 :
        //    //result = ATA_disk_status();
        //    // translate the reslut code here
        //    break;

        case SPI_NORFLASH :
            if(SPI_Flash_Read_ID() == 0xef4017)
            {
                //norflash通信正常
                stat = 0;
            }
            else
            {
                //norflash通信不正常
                stat = STA_NOINIT;
            }
            //stat |= STA_PPROTECT //flash通信正常, 只是flash硬件设置了写保护
            break;

        case SDCard : //1号节点就是SD卡, 0号节点是norflash
            stat &= ~STA_NOINIT; //不管SD卡初始化是否成功我们都返回正确
            break;

        default:
            stat = STA_NOINIT;
    }
    return stat;
}

```

为什么不把 SD 卡放在 2 号, SPI_NORFLASH 放在 1 号, 0 号预留存储设备呢??
设置 SPI_NORFLASH 为 0 号, SDCard 为 1 号也实属无奈才这样做

定义一个获取 SD 卡容量和扇区大小的变量

如果我让 0 号位置预留其它将来可能用到的存储器, 把 norflash 和 SD 卡设置到 1 号和 2 号, 初始话函数是不会出问题, 文件系统挂载是不会出问题, 但是给 SD 卡存储设备创建文件, 读写文件就会出现不可写权限问题

所以存储设备不能有编号空闲, 必须有几个存储设备就要写几个编号

状态函数里面就这样写, 不需要移植 SD 卡底层驱动

```

DSTATUS disk_initialize (
    BYTE pdrv           /* Physical drive number to identify the drive */
)
{
    DSTATUS stat = STA_NOINIT;
    SD_Error Status = SD_OK;

    switch (pdrv) {
// case 0 :
//     //result = ATA_disk_initialize();

//     // translate the result code here

//     return stat;

    case SPI_NORFLASH :
        spi_FUN_init(); //SPI初始化
        return disk_status(SPI_NORFLASH); //传入SPI_NORFLASH节点，表示初始化的是NORFLASH
        break;

    case SDcard : //1号节点就是SD卡，0号节点是norflash
        SDIO_NVIC_Configuration(); //SDIO中断优先级初始化不要忘了
        if((Status = SD_Init()) != SD_OK)
        {
            stat = STA_NOINIT; //SD卡初始化失败
        }
        else
        {
            stat &= ~STA_NOINIT; //SD卡初始化成功
        }
        break;

    default:
        stat = STA_NOINIT;
    }
    return stat;
}

DRESULT disk_read (
    BYTE pdrv,          /* Physical drive number to identify the drive */
    BYTE *buff,         /* Data buffer to store read data */
    DWORD sector,       /* Sector address in LBA */
    UINT count          /* Number of sectors to read */
)
{
    DRESULT res;
    SD_Error ret_status = SD_OK;

    switch (pdrv) {
// case 0 :
//     // translate the arguments here

//     //result = ATA_disk_read(buff, sector, count);

//     // translate the result code here

//     return res;

    case SPI_NORFLASH :
        sector = sector+512; //偏移2M
        READ_norflash_sector(sector*4096,buff,count*4096); //norflash每个扇区数4096字节
        res = RES_OK; //默认flash读取数据正常
        break;

    case SDcard :
        if((DWORD)buff&3)
        {
            while(count--)
            {
                align(4) uint8_t tempbuff[512]; //让数组按照4字节对齐
                res = disk_read(SDcard,(BYTE *)tempbuff,sector++,1);
                if(res == RES_OK)
                {
                    memcpy(buff,tempbuff,SDCardInfo.CardBlockSize);
                    buff+=SDCardInfo.CardBlockSize;
                }
            }
            return res;
        }
        else
        {
            ret_status = SD_ReadMultiBlocks(buff, sector*SDCardInfo.CardBlockSize, SDCard);
            ret_status = SD_WaitReadOperation();
            while(SD_GetStatus() != SD_TRANSFER_OK);
            if(ret_status == SD_OK)
            {
                res = RES_OK;
            }
            else
            {
                res = RES_ERROR;
            }
        }
        break;

    default:
        res = RES_PARERR;
    }
    return res;
}

```

初始化函数必须移植 SD 卡的
SDIO 中断优先级和 SD 卡的初始
化底层驱动函数

这是因为 STM32 DMA2 外设字节
对齐的问题，所以要进行处理

移植 SD 卡连续读块数据函数到文
件系统 disk_read 函数里面

这些我们驱动测试的等待 SD 卡读
取完毕的函数也不能少

```

DRESULT disk_write (
    BYTE pdrv,           /* Physical drive number to identify the drive */
    const BYTE *buff,     /* Data to be written */
    DWORD sector,         /* Sector address in LBA */
    UINT count            /* Number of sectors to write */
)
{
    DRESULT res = RES_PARERR;
    SD_Error ret_status = SD_OK;

    if (!count) {
        return RES_PARERR;      /* Check parameter */
    }

    switch (pdrv) {
// case 0 :
//     // translate the arguments here
//     //result = ATA_disk_write(buff, sector, count);
//     // translate the result code here
//     break;

    case SPI_NORFLASH :
        sector = sector+512;//偏移2M字节 就是2097152字节/4096(一个扇区4096字节) = 512扇区
        WRITE norflash_multisector(sector*4096, (uint8_t *)buff, count*4096);
        //这里和disk_read函数一样，只是buff要强转一下，其实uint8_t和BYTE都是8位的意思
        res = RES_OK;//默认flash写数据正常
        return res;

    case SDcard :
        if((DWORD)buff&3)
        {
            while(count--)
            {
                __align(4) uint8_t tempbuff[512];//让数组按照4字节对齐
                memcpy(tempbuff,buff,SDCardInfo.CardBlockSize);
                res = disk_write(SDcard, (BYTE *)tempbuff,sector++,1);
                if(res == RES_OK)
                {
                    buff+=SDCardInfo.CardBlockSize;;
                }
            }
            return res;
        }
        else
        {
            ret_status = SD_WriteMultiBlocks((uint8_t *)buff, sector*SDCardInfo.CardBloc
            if(ret_status == SD_OK)
            {
                ret_status = SD_WaitWriteOperation();
                while(SD_GetStatus() != SD_TRANSFER_OK);
            }
            if(ret_status == SD_OK)
            {
                res = RES_OK;
            }
            else
            {
                res = RES_ERROR;
            }
        }
        break;

    default:
        res = RES_PARERR;
    }

    return res;
}
#endif

```

就像我前面说的因为读写函数不能有空闲的存储设备号，所以要屏蔽空闲号

这里也是 STM32 DMA2
造成的必须 4 字节对齐
处理方法

SD 卡连续写块的底层驱动
函数移植到文件系统
disk_write 函数里面

给 SD 卡写数据也需要等待 SD
卡写完状态出现，所以移植底
层函数到这里

```

#if _USE_IOCTL
DRESULT disk_ioctl (
    BYTE pdrv,           /* Physical drive nmuber (0..) */
    BYTE cmd,             /* Control code */
    void *buff            /* Buffer to send/receive control data */
)
{
    DRESULT res;

    switch (pdrv) {
// case 0 :
//     return res;

    case SPI_NORFLASH :

        switch (cmd)
        {
            case GET_SECTOR_COUNT:
                *(DWORD *)buff = 1536; //1536个扇区 6M空间
                break;
            case GET_SECTOR_SIZE :
                *(WORD *)buff = 4096;
                break;
            case GET_BLOCK_SIZE :
                *(DWORD *)buff = 1;
                break;
        }

    case SDcard :
        switch (cmd)
        {
            case GET_SECTOR_COUNT:
                *(DWORD *)buff = SDCardInfo.CardCapacity/SDCardInfo.CardBlockSize;
                //SD卡总大小除以每个扇区的字节就是扇区个数
                break;
            case GET_SECTOR_SIZE :
                *(WORD *)buff = SDCardInfo.CardBlockSize;//每个扇区大小是512字节
                break;
            case GET_BLOCK_SIZE :
                *(DWORD *)buff = 1;//SD卡也是一次只擦除一个扇区
                break;
        }
        res = RES_OK;//默认IOCTL操作正常
        return res;
    }

    return RES_PARERR;
}

```

disk_ioctl 函数就只需要实现 SD 卡的扇区大小，扇区个数，每次擦除 SD 卡多少个扇区，这些参数传进来就可以了

ffconf.h 里面要设置存储设备数量

```
#define _VOLUMES      2
/* Number of volumes (logical drives) to be used. */
```

ffconf.h 除了修改存储设备数量，还要修改其它的配置选项，这些选项在 norflash 移植文件系统章节讲了的

现在 SD 卡和以前的 norflash 存储设备初步移植完成，现在板子上是两个存储设备了

我们下面在 SD 卡里面创建个文件，写一些数据进去，测试下文件系统

```

int main(void)
{
    char INbuf[]="123456,abcdefg";//写入文件的数据
    FRESULT ret; //mount返回值, 返回(0)FR_OK表示挂载成功
    uint32_t bytenum = 0;

    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口

    delay_ms(5000);
    ret = f_mount(&fsp, "1:", 1); //挂载文件系统, 设置节点2将SD卡挂载进文件系统
    printf("res = %d\r\n", ret);

    if(ret == FR_NO_FILESYSTEM) //如果SD卡没有格式化, 那就格式化
    {
        f_mkfs("1:", 0, 0);
        ret = f_mount(NULL, "1:", 1); //格式化完成后要取消挂载
        ret = f_mount(&fsp, "1:", 1); //然后再重新挂载才能使用
        printf("res2 = %d\r\n", ret);
    }
    printf("SD card sector size = %d\r\n", SDCardInfo.CardBlockSize);

    ret = f_open(&fp, "1:SDcardFS.txt", FA_OPEN_ALWAYS|FA_WRITE|FA_READ); //打开文件
    printf("open ret = %d\r\n", ret);
    if(ret == FR_OK) //文件打开成功
    {
        printf("open file success\r\n");
        ret = f_write(&fp, INbuf, sizeof(INbuf), &bytenum);
        if(ret == FR_OK)
        {
            printf("write file success bytenum = %d\r\n", bytenum);
        }
        else
        {
            printf("write file failed\r\n");
        }
    }
    f_close(&fp); //这个一定不要忘了, 如果不调用close可能导致文件不被保存
    printf("file system test end\r\n");
    //while(1);
    return 0;
}

```

res = 0
SD card sector size = 512
open ret = 0
open file success
write file success bytenum = 15
file system test end

再去设置 ffconf.h

```

#define _USE_LFN
#define _MAX_LFN 1
#define _CODE_PAGE 936
/* This option specifies the
   FATFS
   ff.c
   diskio.c
   cc936.c

```

1
255
936

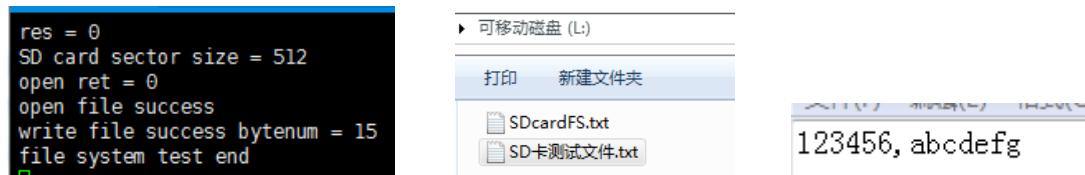
设置 1, 将长文件名放在全局区, 这时必须打开 936 编码支持
选择 936 中文编码

将 SD 卡以前的 txt 文件删除, 插入开发板
重新下载程序, 成功创建大小写 txt 文件

有了长文件名支持中文，测试下中文文件创建

```
ret = f_open(&fp, "1:SD卡测试文件.txt", FA_OPEN_ALWAYS | FA_WRITE | FA_READ); //打开文件  
printf("open ret = %d\r\n", ret);  
if(ret == FR_OK) //文件打开成功  
{  
    printf("open file success\r\n"); //向文件写入数据  
    ret = f_write(&fp, INbuf, sizeof(INbuf), &bytenum);  
    if(ret == FR_OK)  
    {  
        printf("write file success bytenum = %d\r\n", bytenum);  
    }  
    else  
    {  
        printf("write file failed\r\n");  
    }  
}
```

挂载文件套路一样，这里就创建个 SD 卡中文测试文件



成功写入，SD 卡多了中文文件，没有问题，中文文件里面内容也没有问题。

测试读取中文文件数据的功能

```
res = f_mount(&fsp, "0:", 1); //挂载文件系统, 设置节点2将SD卡挂载进文件系统  
printf("res = %d\r\n", ret);  
  
if(ret == FR_NO_FILESYSTEM) //如果SD卡没有格式化，那就格式化  
{  
    f_mkfs("0:", 0, 0);  
    ret = f_mount(NULL, "0:", 1); //格式化完成后要取消挂载  
    ret = f_mount(&fsp, "0:", 1); //然后再重新挂载才能使用  
    printf("res2 = %d\r\n", ret);  
}  
printf("SD card sector size = %d\r\n", SDCardInfo.CardBlockSize);  
  
ret = f_open(&fp, "0:SD卡测试文件.txt", FA_OPEN_ALWAYS | FA_WRITE | FA_READ); //打开文件  
  
printf("open ret = %d\r\n", ret);  
if(ret == FR_OK) //文件打开成功  
{  
    printf("open file success\r\n");  
    f_lseek(&fp, 0);  
    ret = f_read(&fp, readbuf, f_size(&fp), &bytenum); //读取文件数据  
    if(ret == FR_OK)  
    {  
        printf("read file success bytenum = %d\r\n", bytenum);  
    }  
    else  
    {  
        printf("read file failed\r\n");  
    }  
    printf("-----%s-----\r\n", readbuf);  
}  
f_close(&fp); //这个一定不要忘了，如果不调用close可能导致文件不被保存  
printf("file system test end\r\n");  
//while(1);  
return 0;  
}
```

```
res = 0  
SD card sector size = 512  
open ret = 0  
open file success  
read file success bytenum = 15  
-----123456,abcdefg-----  
file system test end
```

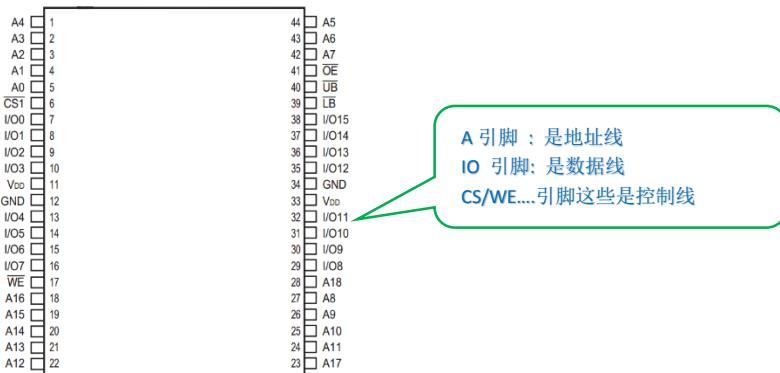
你也可以不用 f_size，用 sizeof 关键字
f_read(&fp, readbuf, sizeof(readbuf), &bytenum);
计算数组长度来获取文件内容也是可以的

读取中文文件内容测试成功，中文能成功英文文件名

也就可以成功。这里就不测试了

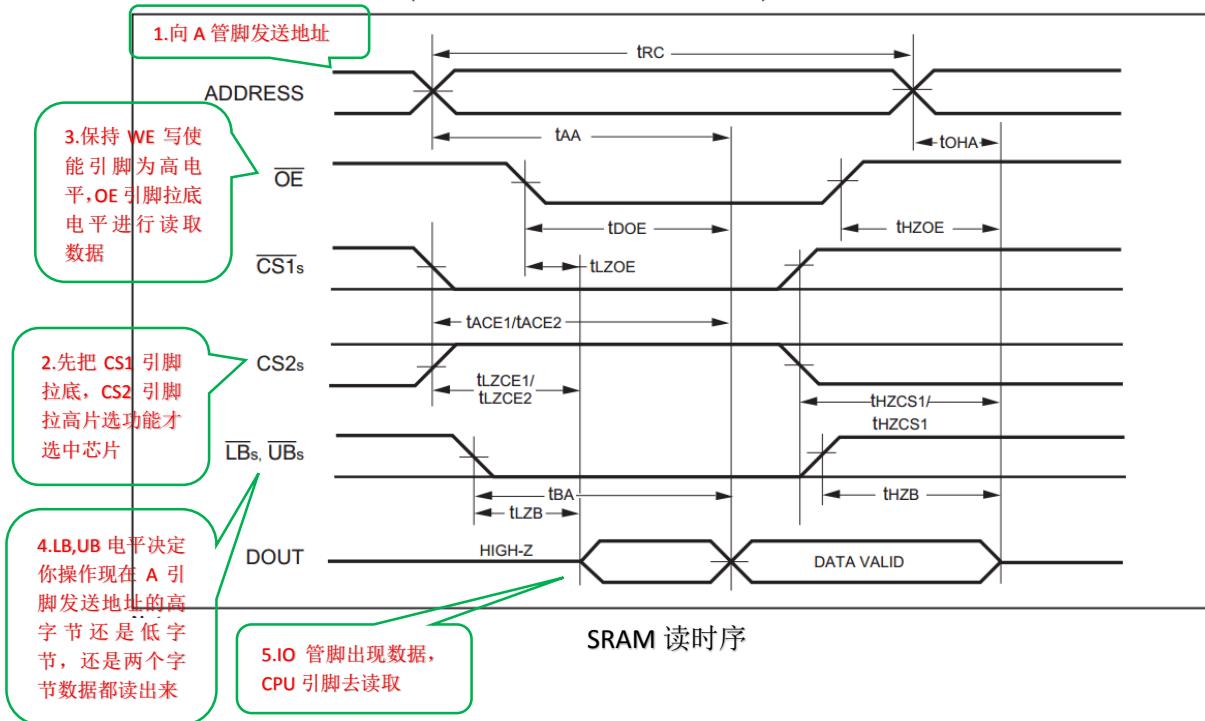
FSMC 驱动 SRAM 实现，方便移植 malloc 内存分配函数定义的变量存放到 SRAM

我用的 SRAM 型号为 IS62WV51216

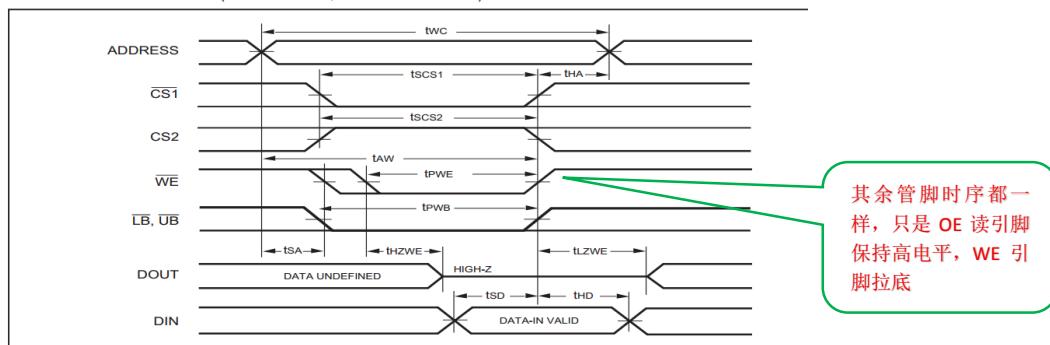


A0~A17 根数据线，就证明了能寻址 2^{18} 次方，就是 512K 的空间，但是 IO 管脚是 0~16 也就是 1 个地址两个字节，那么 512K 地址*2 字节就是 1024KB 的大小能存放 1M 字节数据

READ CYCLE NO. 2^(1,3) ($\overline{CS1}$, $CS2$, \overline{OE} , AND $\overline{UB}/\overline{LB}$ Controlled)



WRITE CYCLE NO. 1^(1,2) ($\overline{CS1}$ Controlled, $\overline{OE} = \text{HIGH or LOW}$)



STM32 FSMC 功能

FSMC 外设可以驱动 nor flash , nand flash, SRAM。不能驱动 SDRAM

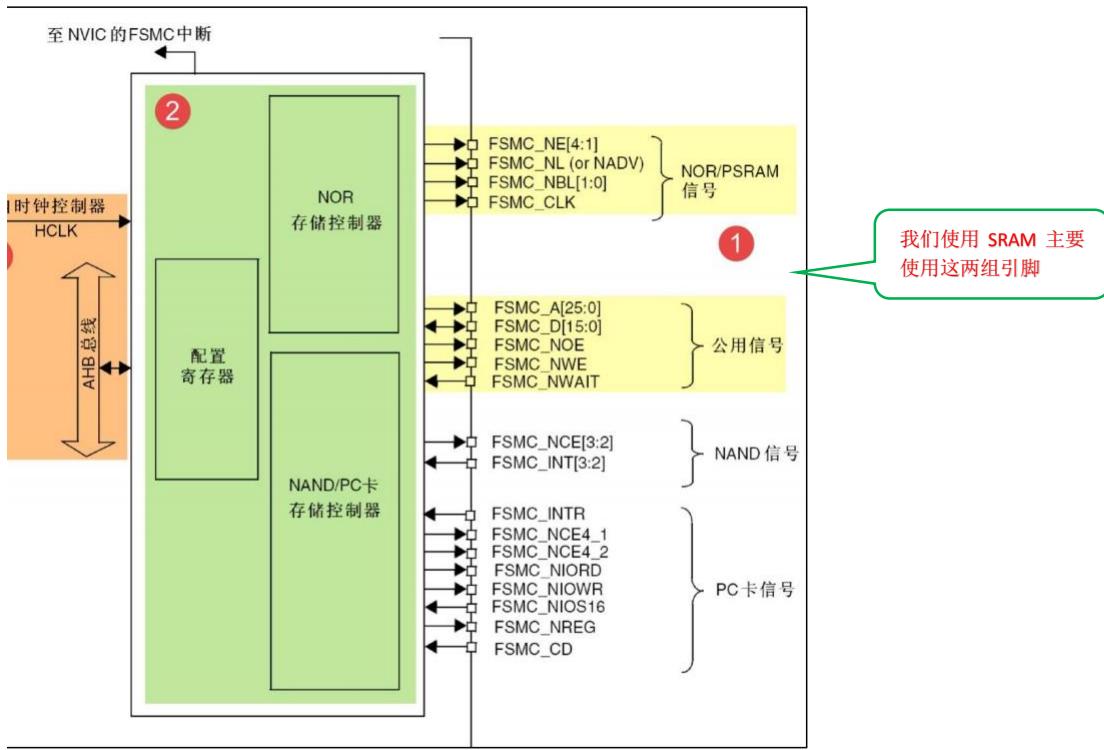


图 27-6 FSMC 控制器框图

表 27-3 FSMC 中的 SRAM 控制信号线

FSMC 引脚名称	对应 SRAM 引脚名	说明
FSMC_NBL[1:0]	LB#、UB#	数据掩码信号
FSMC_A[18:0]	A[18:0]	行地址线
FSMC_D[15:0]	I/O[15:0]	数据线
FSMC_NWE	WE#	写入使能
FSMC_NOE	OE#	输出使能(读使能)
FSMC_NE[1:4]	CE#	片选信号

其中比较特殊的 `FSMC_NE` 是用于控制 SRAM 芯片的片选控制信号线， STM32 具有 `FSMC_NE1/2/3/4` 号引脚，不同的引脚对应 STM32 内部不同的地址区域。

例如，当 STM32 访问 `0x68000000-0x6BFFFFFF` 地址空间时， `FSMC_NE3` 引脚会自动设置为低电平，由于它连接到 SRAM 的 `CE#` 引脚，所以 SRAM 的片选被使能，

而访问 `0x60000000-0x63FFFFFF`

地址时， `FSMC_NE1` 会输出低电平。当使用不同的 `FSMC_NE` 引脚连接外部存储器时，

STM32 访问 SRAM 的地址不一样，从而达到控制多块 SRAM 芯片的目的

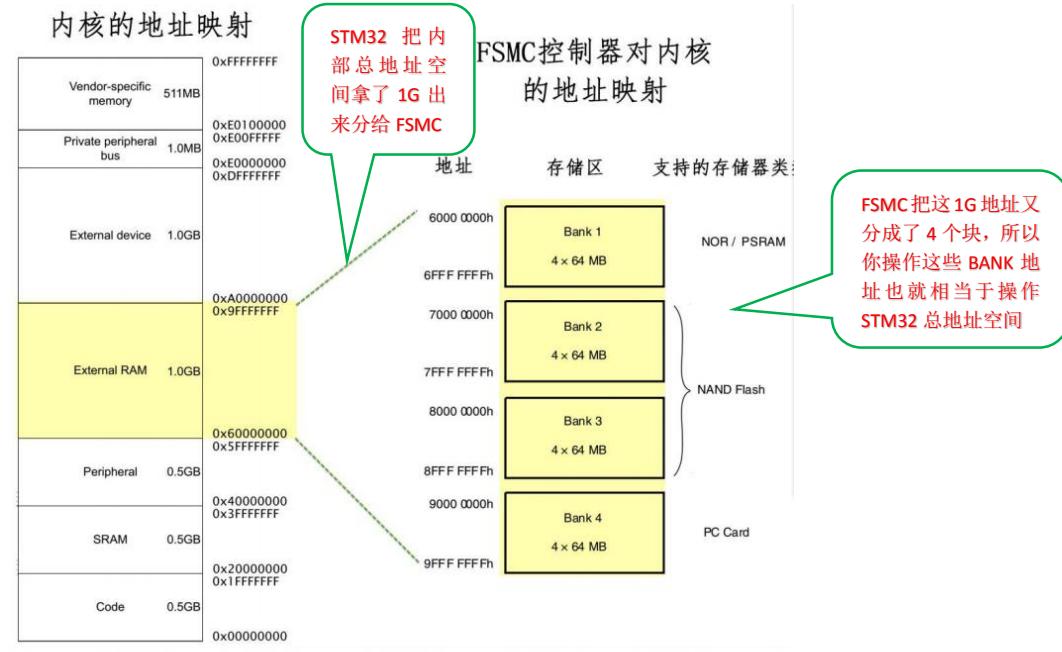


图 27-7 FSMC 的地址映射

我们 SRAM 只能接在 6000 000h~6FFF FFFh

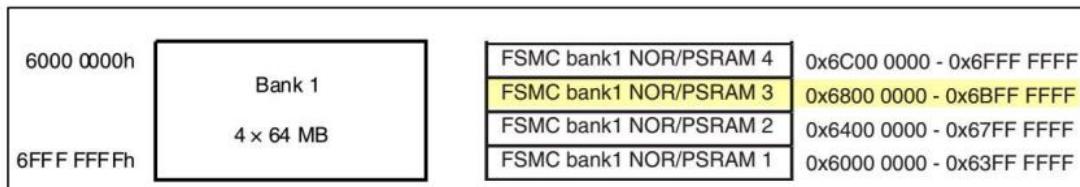
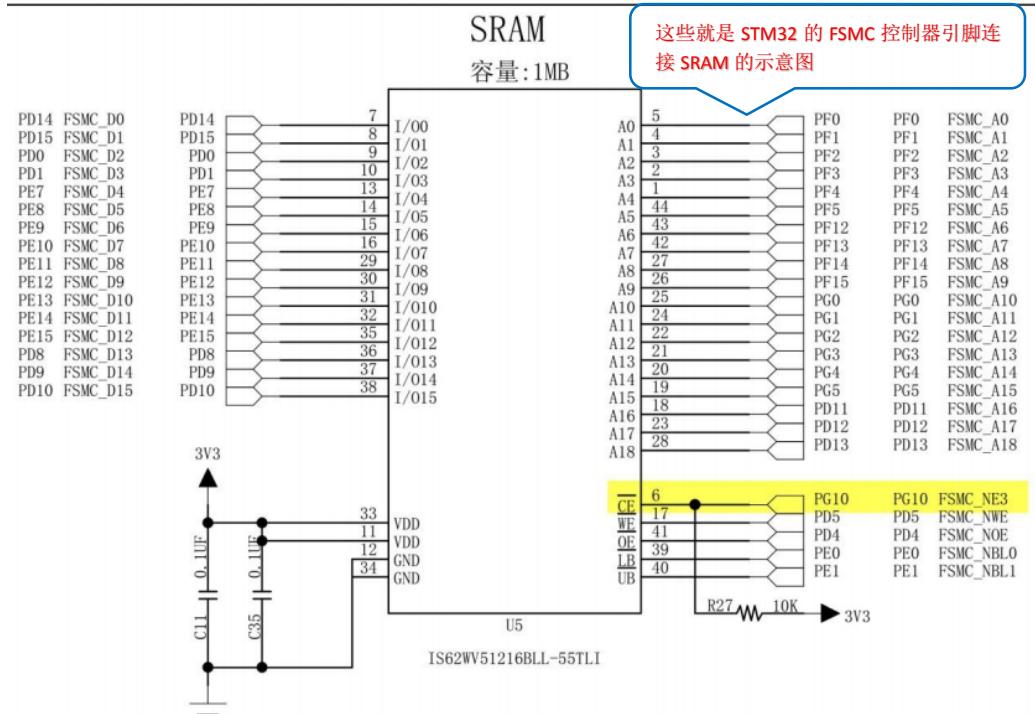


图 27-8 Bank1 内部的小块地址分配



片选选择的是 FSMC_NE3

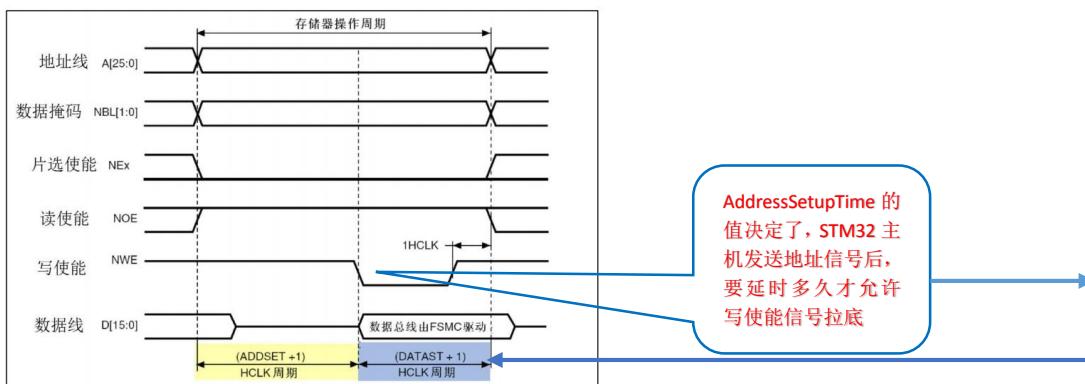
FSMC 初始化代码

初始化时序结构体

```

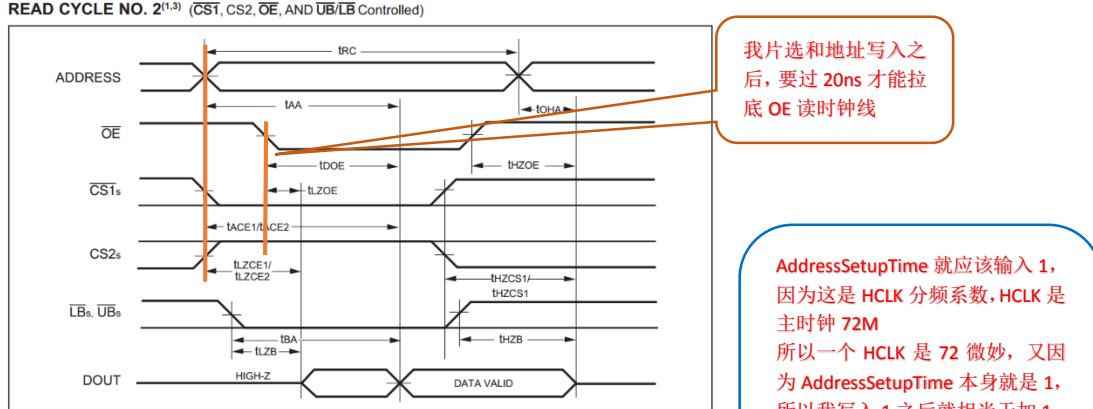
1 typedef struct
2 {
3     uint32_t FSMC_AddressSetupTime;          /*地址建立时间, 0-0xF 个 HCLK 周期*/
4     uint32_t FSMC_AddressHoldTime;           /*地址保持时间, 0-0xF 个 HCLK 周期*/
5     uint32_t FSMC_DataSetupTime;             /*地址建立时间, 0-0xF 个 HCLK 周期*/
6     uint32_t FSMC_BusTurnAroundDuration;    /*总线转换周期, 0-0xF 个 HCLK 周期, 在 NOR FLASH */
7     uint32_t FSMC_CLKDivision;              /*时钟分频因子, 1-0xF, 若控制异步存储器, 本参数无效 */
8     uint32_t FSMC_DataLatency;              /*数据延迟时间, 若控制异步存储器, 本参数无效 */
9     uint32_t FSMC_AccessMode;               /*设置访问模式 */
10 }FSMC_NORSRAMTimingInitTypeDef;

```



给 `FSMC_AddressSetupTime` 参数打个比方

READ CYCLE NO. 2^(1,3) (`CS1`, `CS2`, `OE`, AND `UB/LB` CONTROLLED)



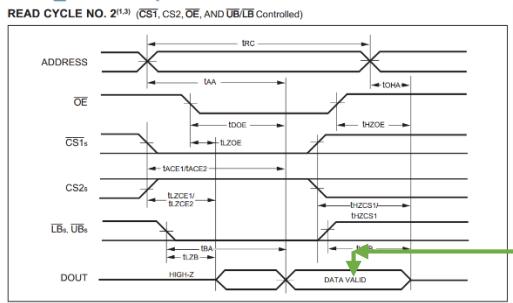
```

1 typedef struct
2 {
3     uint32_t FSMC_AddressSetupTime;          /*地址建立时间, 0-0xF 个 HCLK 周期*/
4     uint32_t FSMC_AddressHoldTime;           /*地址保持时间, 0-0xF 个 HCLK 周期*/
5     uint32_t FSMC_DataSetupTime;             /*地址建立时间, 0-0xF 个 HCLK 周期*/
6     uint32_t FSMC_BusTurnAroundDuration;    /*总线转换周期, 0-0xF 个 HCLK 周期, 在 NOR FLASH */
7     uint32_t FSMC_CLKDivision;              /*时钟分频因子, 1-0xF, 若控制异步存储器, 本参数无效 */
8     uint32_t FSMC_DataLatency;              /*数据延迟时间, 若控制异步存储器, 本参数无效 */
9     uint32_t FSMC_AccessMode;               /*设置访问模式 */
10 }FSMC_NORSRAMTimingInitTypeDef;

```

FSMC_AddressHoldTime 我们写 0，因为我们是驱动 SRAM，不是驱动 norflash 这个时序不需要

FSMC_DataSetupTime 控制数据保持时间



FSMC_BusTurnAroundDuration 这个是用于 norflash，我们 SRAM 不用，所以写 0

FSMC_CLKDivision 我们也不用，这个是做同步时钟的，我们写 0

FSMC_DataLatency 数据保存时间，用于 norflash，我们 SRAM 写 0

FSMC_AccessMode 我们选择模式 A

```
typedef struct
{
    uint32_t FSMC_Bank; /* 设置要控制的 Bank 区域 */
    uint32_t FSMC_DataAddressMux; /* 设置地址总线与数据总线是否复用 */
    uint32_t FSMC_MemoryType; /* 设置存储器的类型 */
    uint32_t FSMC_MemoryDataWidth; /* 设置存储器的数据宽度 */
    uint32_t FSMC_BurstAccessMode; /* 设置是否支持突发访问模式，只支持同步类型的存储器 */
    uint32_t FSMC_AynchronousWait; /* 设置是否使能在同步传输时的等待信号 */
    uint32_t FSMC_WaitSignalPolarity; /* 设置等待信号的极性 */
    uint32_t FSMC_WrapMode; /* 设置是否支持对齐的突发模式 */
    uint32_t FSMC_WaitSignalActive; /* 配置等待信号在等待前有效还是等待期间有效 */
    uint32_t FSMC_WriteOperation; /* 设置是否写使能 */
    uint32_t FSMC_WaitSignal; /* 设置是否使能等待状态插入 */
    uint32_t FSMC_ExtendedMode; /* 设置是否使能扩展模式 */
    uint32_t FSMC_WriteBurst; /* 设置是否使能写突发操作 */

    /* 当不使用扩展模式时，本参数用于配置读写时序，否则用于配置读时序 */
    FSMC_NORSRAMTimingInitTypeDef* FSMC_ReadWriteTimingStruct;
    /* 当使用扩展模式时，本参数用于配置写时序 */
    FSMC_NORSRAMTimingInitTypeDef* FSMC_WriteTimingStruct;
}FSMC_NORSRAMInitTypeDef;
```

因为我的 SRAM 接到的是
BANK1 的第 3 个 64M 空间，所以我写
FSMC_Bank1_NORSRAM3

SRAM 的地址引脚和
数据线引脚是分开的，
所以选择
**FSMC_DataAddressMu
x_Disable**

可以输入的宏	对应的地址区域
FSMC_Bank1_NORSRAM1	0x60000000-0x63FFFF
FSMC_Bank1_NORSRAM2	0x64000000-0x67FFFFFF
FSMC_Bank1_NORSRAM3	0x68000000-0x6BFxFFFF
FSMC_Bank1_NORSRAM4	0x6C000000-0x6FFFFFFFFFF

FSMC_MemoryType 我们存储器类型是 SRAM，所以选择 **FSMC_MemoryType_SRAM**

FSMC_MemoryDataWidth 我们的 SRAM 数据引脚是 16 位，所以选择 **FSMC_MemoryDataWidth_16b**

FSMC_BurstAccessMode 我们 SRAM 不需要突发模式，选择 **FSMC_BurstAccessMode_Disable**

FSMC_AynchronousWait 我们 SRAM 是异步的没有时钟同步，所以选择 **FSMC_AynchronousWait_Disable**

FSMC_WaitSignalPolarity 设置等待信号极性，我们 SRAM 不需要，我设置 **FSMC_WaitSignalPolarity_High**

FSMC_WrapMode 是否支持对齐突发模式，我们 SRAM 不需要，设置 **FSMC_WrapMode_Disable**

FSMC_WaitSignalActive 是一个周期之前等待有效还是之后有效，SRAM 不用配置

FSMC_WaitSignalActive_BeforeWaitState

FSMC_WriteOperation 设置是否写使能，SRAM 要设置写使能 **FSMC_WriteOperation_Enable**

FSMC_ExtendedMode 设置是否支持扩展模式，

使能就是支持读时序用 **ReadWriteTimingStruct** 指针来做，写时序用 **WriteTimingStruct** 指针来做

```
/*当不使用扩展模式时，本参数用于配置读写时序，否则用于配置读时序*/
FSMC_NORSRAMTimingInitTypeDef* FSMC_ReadWriteTimingStruct;
/*当使用扩展模式时，本参数用于配置写时序*/
FSMC_NORSRAMTimingInitTypeDef* FSMC_WriteTimingStruct;
```

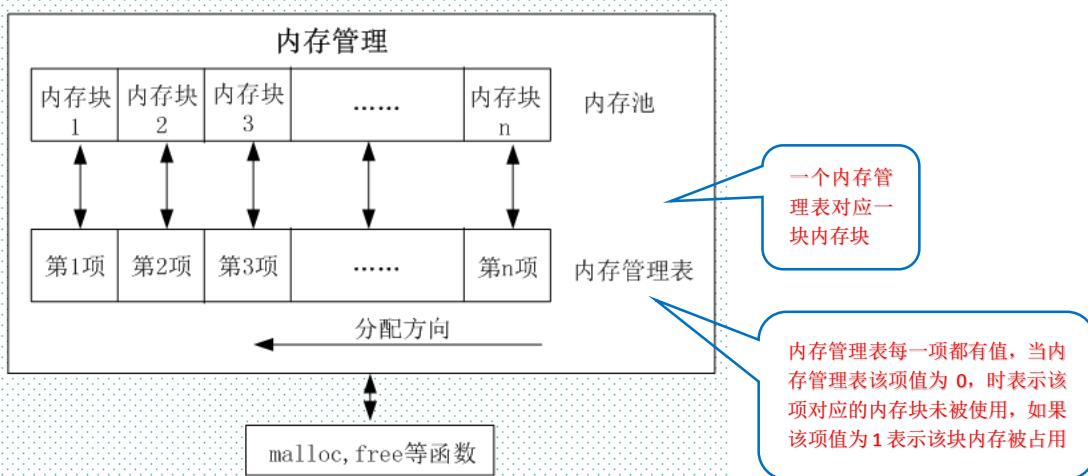
失能就是读写时序都用 **ReadWriteTimingStruct** 指针来做

```
10 int main(void)
11 {
12     uint8_t *pSRAM;
13
14     RCC_Configuration(); // 初始化时钟
15     USART_Config(115200); // 初始化串口
16
17     init_FSMC_GPIO(); // 初始化FSMC连接SRAM的引脚
18     FSMC_ModeConfig(); // 配置FSMC工作模式
19
20     pSRAM = SRAM_BASE_ADDR;
21
22     while(1)
23     {
24         printf("XXXXXXXXXX\r\n");
25
26         delay_ms(1000);
    }
```

Build Output

```
pSRAM = SRAM_BASE_ADDR;
main\main.c(28): warning: #111-D: statement is unreachable
    return 0;
main\main.c(12): warning: #550-D: variable "pSRAM" was set but never used
    uint8_t *pSRAM;
main\main.c: 2 warnings, 1 error
".\out\SRAM_driver.axf" - 1 Error(s), 2 Warning(s).
Target not created
```

STM32 内存管理 malloc 底层实现



如果有 10 项标志位都置 1 了，表示分配了 10 块内存给外部指针变量

内存初始化函数就是把内存里面的所有项标准全部置 0

比如我的 malloc 要分配 100 个字节，假设 1 个内存块是 32 字节，那么就要分配 4 个内存块

malloc 分配方法就是先判断外部定义的指针 p 要分配多少个字节，比如我要分配 100 个字节，然后计算出要多少块内存，然后从内存表第 n 项开始向下查找，直到找到符合 100 个字节的 4 个连续内存块，将这 4 个内存块的内存管理表项置 1。

现在我们来设计内存管理代码 malloc 函数实现

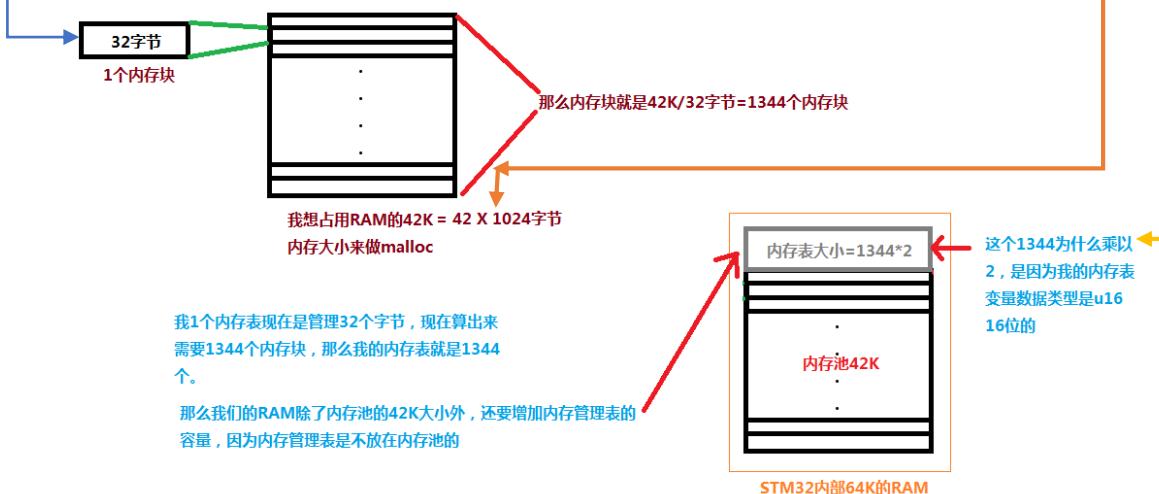
我们现在先设计用 STM32 内部 RAM 空间划分一块出来做 malloc 的动态内存，用 1 个 C 文件实现

```
// 定义分配内存的参数
#define MEM_BLOCK_SIZE 32 // 内存块大小32字节
#define MEM_MAX_SIZE 42*1024 // 最大内存42K
#define MEM_ALLOC_TABLE_SIZE MEM_MAX_SIZE/MEM_BLOCK_SIZE // 内存表大小

// 内部SRAM (RAM) 内存池4字节对齐
__align(4) u8 membase[MEM_MAX_SIZE]; // SRAM内存池

u16 memmapbase[MEM_ALLOC_TABLE_SIZE]; // 定义内存管理表大小

const u32 memtblsize=MEM_ALLOC_TABLE_SIZE; // 内存表大小
const u32 memblksize=MEM_BLOCK_SIZE; // 内存分块大小
const u32 memsize=MEM_MAX_SIZE; // 内存总大小
```



其实我们发现 malloc 分配的内存没有在堆空间，而是在全局区，只是反复使用 RAM 全局区里面分配的内存区域而已

```

// 定义内存控制器结构
struct _m_mallco_dev
{
    void (*init) (void);           // 接受初始化内存函数
    u8 (*perused) (void);         // 接受内存使用率函数
    u8 *membase;                  // 内存池(内部SRAM)
    u16 *memmap;                  // 内存管理状态表
    u8 memrdy;                   // 内存管理是否就绪
};

// 定义一个内存控制器变量
struct _m_mallco_dev mallco_dev; // 定义一个内存控制器变量

// 给内存控制器赋默认值
struct _m_mallco_dev mallco_dev=
{
    mem_init,                      // 内存初始化函数 mem_init 放进来
    mem_perused,                   // 内存使用率函数放进来
    membase,                       // 放 SRAM 内存池进结构体
    memmapbase,                    // 放 SRAM 内存管理表进结构体
    0,                            // 在没有执行内存初始化函数的时候, 默认写 0
};

/* 设置内存 */
// *s 内存首地址
// c 给每个内存地址写的值
// count 需要清 0 的内存地址数(字节为单位)
void mymemset(void *s, u8 c, u32 count)
{
    u8 *xs = s;
    while(count--) *xs+++=c;
}

/* 内存管理初始化 */
void mem_init(void)
{
    mymemset(mallco_dev.memmap, 0, memtblsize*2); // 内存状态表数据清 0
    mymemset(mallco_dev.membase, 0, memsize);      // 内存池(SRAM) 所以数据清 0
    mallco_dev.memrdy=1;                          // 内存初始化函数执行, 这里写 1
}

// 内存分配实现
u32 mem_malloc(u32 size)
{
    signed long offset=0;
    u16 nmemb;
    u16 cmemb=0;
    u32 i;
    if(!mallco_dev.memrdy) mallco_dev.init();
    if(size==0) return 0xFFFFFFFF;
    nmemb=size/memtblsize;
    if(size%memtblsize) nmemb++;
    for(offset=memtblsize-1;offset>=0;offset--)
    {
        if(!mallco_dev.memmap[offset]) cmemb++;
        else cmemb=0;
        if(cmemb==nmemb)
        {
            for(i=0;i<nmemb;i++)
            {
                mallco_dev.memmap[offset+i]=nmemb;
            }
            return (offset*memtblsize);
        }
    }
    return 0xFFFFFFFF;
}

```

memtblsize 是内存表大小 1344

membase[42k] 数组, 传地址进来

memmapbase[1344] 1344 个内存管理表, 传地址进来

把 42K 内存池和内存表数组清 0

先判断内存池开机的时候是否先初始化了, 初始化了的话继续执行下面的程序, 没有初始化就先初始化

你要申请的内存字节大小/内存块大小(32), 得到你要多少个内存块, 最小分配 32 字节, 所以你申请 1 个字节也要占用 1 块(32 字节), 申请 33 字节就要占用 2 块

有些人要申请 100 个字节, 那么 $100/32=3.125$, 除不尽那么把 3 拿出来, 先分配 3 块, 后面 $100 \% 32$ 看看余数是不是 0, 是 0 就不分配了, 不是 0 再分配 1 块, 就是分配 4 块内存块

从内存表数组最后一个元素相前面循环寻找

接续上面的 mem_malloc 代码继续分析

//内存分配实现

```
u32 mem_malloc(u32 size)
{
    signed long offset=0;
    u16 nmemb;
    u16 cmemb=0;
    u32 i;
    if(!mallco_dev.memrdy) mallco_dev.init();
    if(size==0) return 0xFFFFFFFF;
    nmemb=size/memblksize;
    if(size%memblksize) nmemb++;
    for(offset=memtblsize-1;offset>=0;offset--)
    {
        if(!mallco_dev.memmap[offset]) cmemb++;
        else cmemb=0;
        if(cmemb==nmemb)
        {
            for(i=0;i<nmemb;i++)
            {
                mallco_dev.memmap[offset+i]=nmemb;
            }
            return (offset*memblksize);
        }
    }
    return 0xFFFFFFFF;
}
```

如果内存表数组 memmapbase[1344] 其中某项元素等于 0，表示内存表对应的内存块没有被使用，cmemb++表示找到 1 个空的内存块

如果我要分配 100 个字节，需要 4 块内存。如果我循环两次找到了两个空内存，在循环第 3 次的时候发现内存有数据，那么我就把前面两次的内存块也清理掉，因为内存不连续，无法申请 4 块连续内存

如果经过 4 次循环没有出现有数据的内存，那么我这里的 cmemb 就等于了 nmemb (nmemb 是我刚才假设 100 字节，4 块 32 字节内存)，那么给内存表的 4 块内存写上非 0 的数字，标记上。

返回 offset 循环到的可以分配 4 块内存的首地址，然后乘以 32，也就是 1 块内存块的大小

```
void *mymalloc(u32 size)
{
    u32 offset;
    offset=mem_malloc(size);
    if(offset==0xFFFFFFFF) return 0; //如果mem_malloc返回0xFFFFFFFF表示内存申请失败，返回0
    else return ((void*)((u32)mallco_dev.membase+offset)); //内存申请成功返回申请成功内存的首地址
}
```

内存池 membbase[42k] 数组的首地址 + 偏移地址，就是你要使用的首地址，因为内存是向下增长，所以你使用的时候记住不要写出超过你分配的内存大小的数据量就是了

从这里我们看出来了，我们整个分配内存的过程主要是配置内存表，内存池数组反而没有动。只是给了一个内存池里面可以用的内存地址指针而已。

//释放内存函数

//ptr，要释放内存的首地址

```
void myfree(void *ptr)
```

```
{
    u32 offset;
    if(ptr==0) return;
    offset=(u32)ptr-(u32)mallco_dev.membase;
    mem_free(offset);
}
```

ptr 传入现在使用内存的地址 - 内存池数组基址，得到这个内存数组里面偏移的第几个地址

把这个得到的数组地址传入进去

```

//释放内存
//offset要释放内存池里面某块内存的偏移地址
//返回0表示释放内存成功， 返回1表示释放内存失败
u8 mem_free(u32 offset)
{
    int i;
    if(!mallco_dev.memrdy)
    {
        mallco_dev.init(); 判断内存是否开机初始化
        return 1;
    }
    if(offset<memsize) 要清除内存的这个地址在内存范围内才执行清除内存操作
    {
        int index=offset/membblksize;
        int nmemb=mallco_dev.memmap[index];
        for(i=0;i<nmemb;i++)
        {
            mallco_dev.memmap[index+i]=0;
        }
        return 0;
    }else return 2;
}

```

所以在 STM32 实现的整个 free 释放内存其实并没有把释放的内存块全部写 0，而是把内存表清 0 了，到时候再次分配内存直接覆盖内存池数组。这点和 linux 的 malloc 不一样，linux 的 malloc 是在堆里面的，所以操作了将动态内存写 0 的过程。

所以 STM32 的动态内存没有在堆中，而是在全局区的 RAM 中定义了一个很大的数组做动态内存。

下面我们测试下动态分配的内存

```

#include "stdio.h"
#include "malloc.h" //动态分配函数malloc  free 包含该头文件

#define NULL 0

int main(void)
{
    char *p;

    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    mem_init(); //初始化动态内存
    delay_ms(5000);

    p=mm_malloc(2048); //动态分配2K内存空间给指针
    if(p != NULL) //证明内存分配成功可以写数据
    {
        sprintf((char *)p,"malloc request success!!");
    }

    printf("xxxxxxxx\r\n");
    printf("-----%s-----\r\n",p);
    delay_ms(1000);
    printf("-----%s-----\r\n",p);
    delay_ms(1000);
    printf("malloc memuser = %d /100\r\n",mem_perused());
    myfree(p);
    printf("free memuser = %d /100\r\n",mem_perused());
}

```

分配内存成功

测量内存占用率 4%，因为
2048/43008(42K) = 0.04 也就是 malloc 分配的 2048 字节
占用了 42K 内存的 4%

```

xxxxxxxx
-----malloc request success!!
-----malloc request success!!
malloc memuser = 4 /100
free memuser = 0 /100

```

如果这样测试不是很准确那么我们再来看看其它测试方法

```
int main(void)
{
    char *p;

    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    mem_Init(); // 初始化动态内存
    delay_ms(5000);

    // p = mymalloc(2048); // 动态分配2K内存空间给指针p
    if (p != NULL) // 证明内存分配成功可以写数据
    {
        sprintf((char *)p, "malloc request success!!!");
    }

    printf("xxxxxxxx\r\n");
    printf("-----%s-----\r\n", p);
    delay_ms(1000);
}

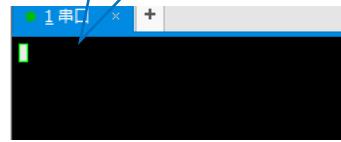
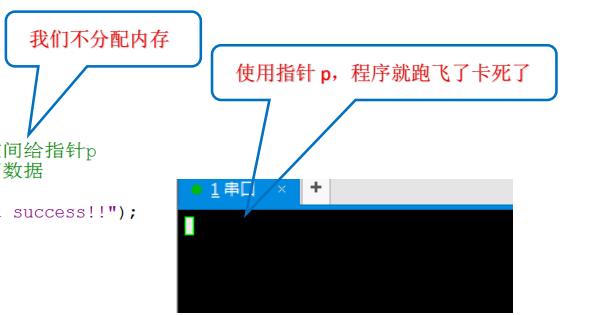
int main(void)
{
    char *p;

    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    mem_Init(); // 初始化动态内存
    delay_ms(5000);

    p = mymalloc(2048); // 动态分配2K内存空间给指针p
    if (p != NULL) // 证明内存分配成功可以写数据
    {
        sprintf((char *)p, "malloc request success!!!");
    }

    printf("xxxxxxxx\r\n");
    printf("-----%s-----\r\n", p);
    delay_ms(1000);
    printf("-----%s-----\r\n", p);

    Target 1
in.c...
: Code=6824 RO-data=336 RW-data=64 ZI-data=46720
sting hex file...
xmalloc.elf" - O Error(s), 0 Warning(s).
```



如果我不想使用 STM32 内部 RAM 做动态内存，想把动态内存的内存池放在其它存储器上怎么办？这时候我们就要用外部内存了，外接一个 SRAM

外部 SRAM 做动态内存

1. 保证上面章节 SRAM 驱动成功
2. 保证上面章节 STM32 内部 RAM 做动态内存的内存代码实现成功

只需要修改 malloc 函数

```
// 定义分配内存的参数
#define MEM_BLOCK_SIZE 32 // 内存块大小32字节
#define MEM_MAX_SIZE 42*1024 // 最大内存42K
#define MEM_ALLOC_TABLE_SIZE MEM_MAX_SIZE/MEM_BLOCK_SIZE // 内存表大小

#define MEM2_BLOCK_SIZE 32 // 内存块大小32字节
#define MEM2_MAX_SIZE 960*1024 // 外部SRAM内存池申请960K, 要留一部分给内存表
#define MEM2_ALLOC_TABLE_SIZE MEM2_MAX_SIZE/MEM2_BLOCK_SIZE // 内存表大小

// 外部SRAM内存池4字节对齐
#define align(4) u8 membase[MEM_MAX_SIZE]; // RAM内部内存存池
#define _align(32) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0x68000000))); // 32字节对齐
// 这是用attribute绝对定位，把数组定义到外部内存的首地址0x68000000，为什么这样定义？
// 这是因为我们SRAM章节说了，使用0x68000000才能驱动FSMC去读取1M的外部SRAM
#define memmapbase[MEM2_ALLOC_TABLE_SIZE] __attribute__((at(0x68000000+MEM2_MAX_SIZE)));
// 定义外部SRAM内存管理表大小

const u32 memtblsize=MEM2_ALLOC_TABLE_SIZE; // 外部内存表大小
const u32 memblksize=MEM2_BLOCK_SIZE; // 外部内存分块大小
const u32 memsize=MEM2_MAX_SIZE; // 外部内存总大小
```

增加外部内存参数

修改内存池定位到外部 SRAM

内存表数组也定义到外部内存

修改内存基本参数

```

3 //给内存控制器赋默认值
4 struct _m_mallco_dev mallco_dev=
5 {
6     mem_init,           //内存初始化函数mem_init放进来
7     mem_perused,        //内存使用率函数放进来
8     //membase,           //放SRAM内存池进结构体
9     mem2base,           //放外部SRAM内存池进结构体
10    memmapbase,         //放SRAM内存管理表进结构体
11    0,                  //在没有执行内存初始化函数的时候,默认写0
12 };

```

内存控制器增加外部内存池

其余的 malloc 代码不变

```

4 #include "stdio.h"
5 #include "malloc.h" //动态分配函数malloc free 包含该头文件
6
7 #define NULL 0
8
9 int main(void)
10 {
11     char *p;
12
13     RCC_configuration(); //初始化时钟
14     USART_config(115200); //初始化串口
15     mem_init(); //初始化动态内存
16     delay_ms(5000);
17
18     p=mymalloc(2048); //动态分配2K内存空间给指针p
19     if(p != NULL) //证明内存分配成功可以写数据
20     {
21         sprintf((char *)p,"malloc request success!!");
22     }
23
24     printf("xxxxxxxx\r\n");

```

```

rget 'Target 1'
g malloc.c...
..
Size: Code=6824 RO-data=336 RW-data=64 ZI-data=1045504
creating hex file...
RAM_xmalloc.axf" - 0 Error(s), 0 Warning(s).

```

你看编译出来的 RAM 空间超过 1M 了,
里面很大一部分是外部 RAM

我没有测试过，你可以测试下这个外部内存程序

我还附录了一份正点原子的内部 RAM 和外部 SRAM 同用 malloc 程序放在 zmalloc 目录下

WAV 文件创建代码设计

wav 文件在 STM32 的 SD 卡创建需要上面的 SD 卡 FATFS 文件系统

wav 文件需要用内存管理,保证上面章节的 malloc 实现成功

```
//初始化wav头
```

```
void recoder_wav_init(__WaveHeader* wavhead) //初始化WAV头
{
    wavhead->riff.ChunkID=0X46464952; // "RIFF"
    wavhead->riff.ChunkSize=0; //还未确定需要最后计算, 先填0
    wavhead->riff.Format=0X45564157; // "WAVE"
    wavhead->fmt.ChunkID=0X20746D66; // "fmt "
    wavhead->fmt.ChunkSize=16; //大小为16字节
    wavhead->fmt.AudioFormat=0X01; //0X01, 表示线性PCM
    wavhead->fmt.NumOfChannels=1; //1表示单声道
    wavhead->fmt.SampleRate=8000; //8Khz 采样率
    wavhead->fmt.ByteRate=wavhead->fmt.SampleRate*2;//16位, 既2字节
    wavhead->fmt.BlockAlign=2; //块大小:2字节为1块
    wavhead->fmt.BitsPerSample=16; //16位PCM
    wavhead->data.ChunkID=0X61746164; // "data"
    wavhead->data.ChunkSize=0; //数据大小还需计算, 先写0
}
```

WAV 文件格式实例分析:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000H	52	49	46	46	0A	06	01	00	57	41	56	45	66	6D	74	20
00000010H	12	00	00	00	01	00	02	00	44	AC	00	00	10	B1	02	00
00000020H	04	00	10	00	64	61	74	61	E6	05	01	00	00	00	00	00

```

wavhead->riff.ChunkID      //写入 0x46464952 这是固定数据表示 RIFF
wavhead->riff.ChunkSize     //整个 wav 文件的大小, (就是 wav 头数据+后面的音频数据)-8,
                            //这个-8 就是减去 riff.ChunkID 和 riff.ChunkSize 的字节
wavhead->riff.Format       //写入 0x45564157 这是固定数据表示 wave
wavhead->fmt.ChunkID        //写入 0x20746d66 这是固定数据表示 fmt
wavhead->fmt.ChunkSize      //子集合大小 16 或者 18, 线性 PCM 只选择 16
                            //子集合就是 fmt.AudioFormat~fmt.BitsPerSample 这几个变量加起来的大小
wavhead->fmt.AudioFormat    //一般为 0x00001, 表示线性 PCM
wavhead->fmt.NumOfChannels   //通道数 1 表示单声道, 通道数 2 表示双声道
wavhead->fmt.SampleRate      //采用率如果设置 0x1f40, 就是 8K 采样率, 可以选择其它
wavhead->fmt.ByteRate        //字节速率
                            //字节速率就是采样率 X 通道数 X (ADC 位数/8)
                            //举例: ADC 采样 8000(8K), ADC 单声道话筒, ADC 精度 16 位
                            //字节速率=8000 X 1 X (16/8), 也就是你 ADC 每秒获取 16000 个字节给 CPU
wavhead->fmt.BlockAlign      //块对齐(字节对齐), 块对齐=通道数*(ADC 位数/8)
wavhead->fmt.BitsPerSample    //单个采样位数, 我们是 16PCM(16 位 ADC), 设置 16
wavhead->data.ChunkID        //这里固定写 0x61746164
wavhead->data.ChunkSize       //这里是音频数据大小(不包含前面的 wav 头数据)

```

我发现原子的 mymalloc 函数在遇到 FATFS 文件系统时就会出现 mymalloc 函数申请的 RAM 空间和 FATFS 申请的 RAM 空间地址冲突, 所以我还是使用的 C 标准库的 malloc 函数

```

2 FATFS fsp;//文件系统使用要先有这个句柄，这个句柄很大所以定义在全局变量
3 extern SD_CardInfo SDCardInfo;
4
5 int main(void)
6 {
7     FIL fp;//读写文件要创建文件句柄
8
9     FRESULT ret; //mount返回值，返回(0)FR_OK表示挂载成功
10    uint32_t bytelen = 0,i = 0;
11
12    __WaveHeader *wavhead=0;
13    u32 wavsize = 0;//WAV文件数据大小，但是不包含wav头数据大小
14
15    uint8_t buff[12] = {0xff,1,2,3,4,5,6,7,8,9,10,11};//产生的音频数据
16
17    RCC_configuration(); //初始化时钟
18    USART_config(115200); //初始化串口
19
20    delay_ms(5000);
21
22    ret = f_mount(&fsp,"0:",1); //挂载文件系统，设置节点2将SD卡挂载进文件系统
23    printf("res = %d\r\n",ret);
24    if(ret == FR_NO_FILESYSTEM)//如果SD卡没有格式化，那就格式化
25    {
26        f_mkfs("0:",0,0);
27        ret = f_mount(NULL,"0:",1); //格式化完成后要取消挂载
28        ret = f_mount(&fsp,"0:",1); //然后再重新挂载才能使用
29        printf("res2 = %d\r\n",ret);
30    }
31
32    wavhead = (__WaveHeader *)malloc(sizeof(__WaveHeader)); //创建wav头数据
33    recorder_wav_init(wavhead); //给wav头写头数据
34
35    ret = f_open(&fp,"0:recorder.wav",FA_OPEN_ALWAYS|FA_WRITE|FA_READ); //打开创建wav文件
36    printf("open ret = %d\r\n",ret);
37    if(ret == FR_OK) //文件打开成功
38        printf("wav open file success\r\n"); //wav文件创建成功
39    else
40        printf("wav open file failed\r\n");
41
42    ret = f_write(&fp,(const void*)wavhead,sizeof(__WaveHeader),&bytelen); //将wav头写入wav文件
43    if(ret == FR_OK)
44        printf("wav write RIFF success\r\n");
45    else
46        printf("wav write RIFF failed\r\n");
47
48    ret = f_write(&fp,(const void*)buff,sizeof(buff),&bytelen); //将数据头写入wav文件
49    if(ret == FR_OK)
50        printf("wav write data success\r\n");
51    else
52        printf("wav write data failed\r\n");
53
54    ret = f_write(&fp,(const void*)wavhead,sizeof(__WaveHeader),&bytelen); //将wav头写入wav文件
55    if(ret == FR_OK)
56        printf("wav write RIFF success\r\n");
57    else
58        printf("wav write RIFF failed\r\n");
59
60    ret = f_write(&fp,(const void*)buff,sizeof(buff),&bytelen); //将数据头写入wav文件
61    if(ret == FR_OK)
62        printf("wav write data success\r\n");
63    else
64        printf("wav write data failed\r\n");
65
66    wavsize = sizeof(buff); //计算音频数据长度
67
68    wavhead->riff.ChunkSize = wavsize+36; //将wav头和后面的音频数据加起来的长度
69    wavhead->data.ChunkSize = wavsize; //将计算的音频数据长度写在wav头的数据长度变量里
70
71    f_lseek(&fp,0); //偏移到wav文件头，把计算的wav文件数据长度覆盖进前面写入文件的wav头数据
72    ret = f_write(&fp,(const void*)wavhead,sizeof(__WaveHeader),&bytelen); //将数据计算结果写入wav文件
73    if(ret == FR_OK)
74        printf("wav write data size success\r\n");
75    else
76        printf("wav write data size failed\r\n");
77    f_close(&fp);

```



经过以上代码测试，可以生成一个播放器识别的 wav 文件

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	52	49	46	46	30	00	00	00	57	41	56	45	66	6D	74	20
00000010	10	00	00	00	01	00	01	00	40	1F	00	00	80	3E	00	00
00000020	02	00	10	00	64	61	74	61	0C	00	00	00	FF	01	02	03
00000030	04	05	06	07	08	09	0A	0B	FF	01	02	03	04	05	06	07
00000040	08	09	0A	0B	FF	01	02	03	04	05	06	07	08	09	0A	0B

这是最后计算出来的整个文件数据长度

```
//初始化wav头
void recoder_wav_init(__WaveHeader* wavhead) //初始化WAV头
{
    wavhead->riff.ChunkID=0X46464952;           // "RIFF"
    wavhead->riff.ChunkSize=0;                   // 还未确定需要最后计算, 先填0
    wavhead->riff.Format=0X45564157;           // "WAVE"
    wavhead->fmt.ChunkID=0X20746D66;           // "fmt "
    wavhead->fmt.ChunkSize=16;                  // 大小为16字节
    wavhead->fmt.AudioFormat=0X01;              // 0X01, 表示线性PCM
    wavhead->fmt.NumOfChannels=1;                // 1表示单声道
    wavhead->fmt.SampleRate=8000;               // 8Khz 采样率
    wavhead->fmt.ByteRate=wavhead->fmt.SampleRate*2;//16位, 既2字节
    wavhead->fmt.BlockAlign=2;                  // 块大小:2字节为1块
    wavhead->fmt.BitsPerSample=16;              // 16位PCM
    wavhead->data.ChunkID=0X61746164;          // "data"
    wavhead->data.ChunkSize=0;                  // 数据大小还需计算, 先写0
}
```

有了 wav 文件的基本测试, 我们下面来测试 I2S 总线

请注意, 这里面程序用的 malloc 是有问题的, keil 系统并没有 malloc, 它是去自动寻找了我的 mymalloc, 所以注意系统的 malloc 是没有实现的

I2S 总线获取音频数据

STM32 I2S 引脚

I2S_LRCK	PB12	73	PB12/SPI2_NSS/TIM1_BKIN/U3_CK/CAN2_RX/I2C2_SMBA/OTG_HS_ULPI_D5/OTG_HS_ID/ETH_MII_TXD0/ETH_RMII_TXD0/I2S2_WS
I2S_SCLK	PB13	74	PB13/SPI2_SCK/TIM1_CHIN/U3_CTS/CAN2_TX/OTG_HS_ULPI_D6/OTG_HS_VBUS/ETH_MII_TXD1/ETH_RMII_TXD1/I2S2_CLK

I2S_WS : WS 引脚就是左右声道切换时钟, WS 信号频率等于音频信号采样率

I2S_CK : 串行时钟, CK 频率 = WS(采样率) X 通道数 X ADC 位数

例如:ADC 是 192K 采样率, 2 个话筒采集信号, ADC 精度为 16 位

CK = 192000 X 2 X16

I2S_SDOUT	PC2	28	PC2/SPI2_MISO/OTG_HS_ULPI_DIR/ETH_MII_TXD2/I2S2ext_SD/ADC123_IN12
I2S_SDIN	PC3	29	PC3/SPI2_MOSI/OTG_HS_ULPI_NXT/ETH_MII_TX_CLK/I2S2_SD/ADC123_IN13

I2S_SD : 在 STM32F1 系列下面就只有这一个引脚, 录音时用来接受音频芯片 ADC 发出的数据, 播放时用来发送数据给音频芯片 DAC, 这是个录音和播放复用的数据引脚, 是半双工模式。

I2S2ext_SD : 这个引脚在 STM32F4 系列添加了的, 为了弥补 I2S_SD 的半双工, 不能全双工的缺陷。这样在 STM32F4 就可以支持全双工模式, I2S_SD 接受数据, I2S2_SD 发送数据

I2S_MCLK	DCMI_D0	PC6	96	PC6/TIM1_CH1/TIM8_CH1/U6_TX/SDIO_D6/DCMI_D0/I2S2_MCK
DCMI_D1	DC7	07		

MCLK: 主时钟输出引脚, 有些 codec 芯片节省成本, 没有外接晶振给 codec 的 MCLK 时钟, 所以交由 STM32 主机输出 MCLK 时钟给 codec, 该 MCLK 时钟频率=256 X fs(采样率), 这是固定频率。

I2S_LRCK	7	LRC
I2S_SCLK	8	BCLK
I2S_SDOUT	9	ADCDAT
I2S_SDIN	10	DACDAT
I2S_MCLK	11	MCLK

这就是 codec 编解码芯片和主机的 I2S 连接方法

单声道	取样 1	取样 2	取样 3	取样 4
8位量化	声道 0	声道 0	声道 0	声道 0
双声道	取样 1		取样 2	
8位量化	声道 0(左)	声道 1(右)	声道 0(左)	声道 1(右)
单声道	取样 1		取样 2	
16位量化	声道 0(低字节)	声道 0(高字节)	声道 0(低字节)	声道 0(高字节)
双声道	取样 1			
16位量化	声道 0 (左, 低字节)	声道 0 (左, 高字节)	声道 1 (右, 低字节)	声道 1 (右, 高字节)

wav 文件格式是录音声道低字节在前，高字节在后，但是 STM32 的 I2S 输出的是 MSB 高字节在前，低字节在后。

8位量化	声道 0					
双声道	取样 1		取样 2		取样 3	
8位量化	声道 0(左)	声道 1(右)	声道 0(左)	声道 1(右)	声道 0(左)	声道 1(右)
单声道	取样 1		取样 2		取样 3	
16位量化	声道 0 (低字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (高字节)
双声道	取样 1				取样 2	
16位量化	声道 0 (低字节)	声道 0 (高字节)	声道 1 (低字节)	声道 1 (高字节)	声道 0 (低字节)	声道 0 (高字节)
单声道	取样 1			取样 2		
24位量化	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)
双声道	取样 1					
24位量化	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)	声道 1 (低字节)	声道 1 (中字节)	声道 1 (高字节)

表 48.1.1.1 WAVE 文件数据采样格式

1.首先是要 I2C 驱动 WM8978 芯片寄存器操作正常

因为 STM32 的硬件 I2C 有 BUG , 所以我们用模拟 I2C。

1.1 写 I2C 驱动

```
myiic.c
//IO方向设置
void SDA_IN()
{
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;//PB11输入模式
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化
}
void SDA_OUT()
{
    //GPIOB11输出
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;//PB11输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP ; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化
}
```

STM32 模拟 I2C 驱动网上资料很

多，这里就不一一概述了。

1.2 用 I2C 驱动操作 WM8978 寄存器

```
//注意:WM8978的寄存器值是9位的,所以要用u16来存储.
static u16 WM8978_REGVAL_TBL[58]=
{
    0X0000,0X0000,0X0000,0X0000,0X0050,0X0000,0X0140,0X0000,
    0X0000,0X0000,0X0000,0X00FF,0X00FF,0X0000,0X0100,0X00FF,
    0X00FF,0X0000,0X012C,0X002C,0X002C,0X002C,0X0000,
    0X0032,0X0000,0X0000,0X0000,0X0000,0X0000,0X0000,0X0000,
    0X0038,0X000B,0X0032,0X0000,0X0008,0X000C,0X0093,0X00E9,
    0X0000,0X0000,0X0000,0X0000,0X0003,0X0010,0X0010,0X0100,
    0X0100,0X0002,0X0001,0X0001,0X0039,0X0039,0X0039,0X0039,
    0X0001,0X0001
};

//WM8978初始化
//返回值:0,初始化正常
// 其他,错误代码
u8 WM8978_Init(void)
{
    u8 res;

    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable GPIOB, GPIOC and AFIO clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOB |
        IIC_Init(); //初始化I2C接口
    res=WM8978_Write_Reg(0,0); //软复位WM8978
    if(res) return 1; //发送指令失败,WM8978异常
    //以下为通用设置
    WM8978_Write_Reg(1,0X1B); //R1,MICEN设置为1(MIC使能),BIASEN设置为
    WM8978_Write_Reg(2,0X1B0); //R2,ROUT1,LOUT1输出使能(耳机可以工作)
    WM8978_Write_Reg(3,0X6C); //R3,LOUT2,ROUT2输出使能(喇叭工作),RMI
    WM8978_Write_Reg(6,0); //R6,MCLK由外部提供
    WM8978_Write_Reg(43,1<<4); //R43,INVROUT2反向,驱动喇叭
    WM8978_Write_Reg(47,1<<8); //R47设置,PGABOOSTL,左通道MIC获得20倍增益
    WM8978_Write_Reg(48,1<<8); //R48设置,PGABOOSTR,右通道MIC获得20倍增益
    WM8978_Write_Reg(49,1<<1); //R49,TSDEN,开启过热保护
    WM8978_Write_Reg(10,1<<3); //R10,SOFTMUTE关闭,128x采样,最佳SNR
    WM8978_Write_Reg(14,1<<3); //R14,ADC 128x采样率
    return 0;
}

/*打开wm8978录音功能, 打开主机IIS录音功能*/
void recoder_mode(void)
{
    WM8978_Init(); //codec初始化
    WM8978_ADDA_Cfg(0,1); //开启ADC
    WM8978_Input_Cfg(1,1,0); //开启输入通道(MIC&LINE IN)
    WM8978_Output_Cfg(0,1); //开启BYPASS输出
    WM8978_MIC_Gain(46); //MIC增益设置

    WM8978_I2S_Cfg(2,0); //飞利浦I2S标准,16数据长度

    I2S3_Init(); //I2S接口初始化
    I2S3_NVIC_Configuration(); //I2S中断优先级设置
}
```

这是初始化 WM8978 的程序，这个程序把耳机和喇叭直通打开。这样就能在耳机上确认 I2C 总线驱动代码和 WM8978 寄存器操作顺序是否正常

将 `recoder_mode` 函数放在主函数 `main` 去运行，应该能听到耳机声音，如果听不到那么就可能有硬件问题，软件问题，自己排查。

2.模拟 I2C 驱动 WM8978 完成，下面将读写 SD 卡 FATFS 文件系统放进工程

```
FATFS fsp; //文件系统使用要先有这个句柄，这个句柄很大所以定义在全局变量
extern SD_CardInfo SDCardInfo;

uint16_t I2S2_Buffer_Rx[32];
uint16_t I2SFLAG = 0;

int main(void)
{
    FIL fp; //读写文件要创建文件句柄

    FRESULT ret; //mount返回值，返回(0)FR_OK表示挂载成功
    uint32_t bytenum = 0;
    u32 wavsize = 0; //WAV文件数据大小，但是不包含wav头数据大小
    __WaveHeader wavhead;

    recoder_mode(); //打开录音

    ret = f_open(&fp, "0:recorder.wav", FA_OPEN_ALWAYS | FA_WRITE | FA
printf("open ret = %d\r\n", ret);
if(ret == FR_OK) //文件打开成功
    printf("wav open file success\r\n"); //wav文件创建成功
else
    printf("wav open file failed\r\n");

    ret = f_write(&fp, (const void*)&wavhead, sizeof(__WaveHeader));
if(ret == FR_OK)
    printf("wav write RIFF success\r\n");
else
    printf("wav write RIFF failed\r\n");

    XXXXXXXX
    res = 0
    open ret = 10
    wav open file failed
    wav write RIFF failed
```

有了文件系统我们再次尝试操作 WM8978 寄存器，发现 WM8978 寄存器操作成功，文件系统挂载成功，但是读写文件系统出问题了

你看读写文件系统失败。

经过一系列排查发现是内存全局区的空间被占满了

```
static u16 WM8978_REGVAL_TBL[58] =
{
    0x0000, 0x0000, 0x0000, 0x0000, 0x0050, 0x0000, 0x0140, 0x0000,
    0x0000, 0x0000, 0x0000, 0x000F, 0x00FF, 0x0000, 0x0100, 0x00FF,
    0x00FF, 0x0000, 0x012C, 0x002C, 0x002C, 0x002C, 0x002C, 0x0000,
    0x0032, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
    0x0038, 0x000B, 0x0032, 0x0000, 0x0008, 0x000C, 0x0093, 0x00E9,
    0x0000, 0x0000, 0x0000, 0x0000, 0x0003, 0x0010, 0x0010, 0x0010,
    0x0010, 0x0002, 0x0001, 0x0001, 0x0039, 0x0039, 0x0039, 0x0039,
    0x0001, 0x0001
};

wm8978 数组占用了全局区
```

文件系统 FATFS 文件句柄占用了全局区

发现 STM32F103VC 这个系列 RAM 只有 48K，但是如果把 WM8978 的全局数组放在函数里面放在栈区，会导致寄存器无法正常工作。那就只有把主函数的 FATFS 文件系统放进栈区。

2.1 修改栈区大小

```
Stack_Size      EQU      0x000001000
                AREA     STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE    Stack_Size
__initial_sp

; <h> Heap Configuration
; <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Heap_Size       EQU      0x00000400

int main(void)
{
    FATFS fsp; //文件系统使用要先有这个句柄，这个句柄很大所以定义在全局变量
    FIL fp; //读写文件要创建文件句柄

    FRESULT ret; //mount返回值，返回(0)FR_OK表示挂载成功
    uint32_t bytenum = 0;
    u32 wavsize = 0; //WAV文件数据大小，但是不包含wav头数据大小
    __WaveHeader wavhead;
```

把 STM32 启动文件栈区改到 1000

把文件系统句柄放到主函数栈区上

```
XXXXXXXX
res = 0
open ret = 0
wav open file success
wav write RIFF success
```

打开文件系统，写文件系统成功，而且还能听到 WM8978 耳机声音。

还有很多录音内容没讲完，请参考我 STM32 外设操作文档对录音功能补充...

DMA 驱动实验

在用 I2S 传输音频数据之前，我们测试下 DMA

STM32F103 有两个 DMA 通道，DMA1,DMA2

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC1	ADC1						
SPI/I ² S		SPI1_RX	SPI1_TX	SPI/I2S2_RX	SPI/I2S2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I ² C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

图 22-2 DMA1 各个通道的请求映像

DMA1 通道支持的外设

外设	通道1	通道2	通道3	通道4	通道5
ADC3 ⁽¹⁾					ADC3
SPI/I2S3	SPI/I2S3_RX	SPI/I2S3_TX			
UART4			UART4_RX		UART4_TX
SDIO ⁽¹⁾				SDIO	
TIM5	TIM5_CH4 TIM5_TRIG	TIM5_CH3 TIM5_UP		TIM5_CH2	TIM5_CH1
TIM6/ DAC 通道1			TIM6_UP/ DAC 通道1		
TIM7/ DAC 通道2				TIM7_UP/ DAC 通道2	
TIM8 ⁽¹⁾	TIM8_CH3 TIM8_UP	TIM8_CH4 TIM8_TRIG TIM8_COM	TIM8_CH1		TIM8_CH2

图 22-3 DMA2 各个通道的请求映像

DMA2 通道支持的外设

包含 STM32F10x_dma.c/h 文件

```
typedef struct
{
    uint32_t DMA_PeripheralBaseAddr; // 外设地址
    uint32_t DMA_MemoryBaseAddr; // 存储器地址
    uint32_t DMA_DIR; // 传输方向
    uint32_t DMA_BufferSize; // 传输数目
    uint32_t DMA_PeripheralInc; // 外设地址增量模式
    uint32_t DMA_MemoryInc; // 存储器地址增量模式
    uint32_t DMA_PeripheralDataSize; // 外设数据宽度
    uint32_t DMA_MemoryDataSize; // 存储器数据宽度
    uint32_t DMA_Mode; // 模式选择
    uint32_t DMA_Priority; // 通道优先级
    uint32_t DMA_M2M; // 存储器到存储器模式
} DMA_InitTypeDef;
```

一次传输多少个数据，比如你数组是 `uint8_t buff[32]` 那么这里就填 32，传输的数据和数据类型无关，只关心数据个数，如果你是 `uint16_t buff[32]`，那么也是 32 个数据。但是 F103 最大只能填入 65535

这里填入外设地址，比如串口/ADC 的数据寄存器地址

就是接受的串口数据放在内存什么地址上

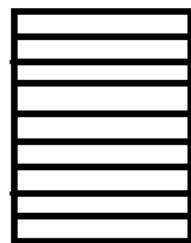
DMA_DIR_PeripheralDST
该选项是内存发数据到外设比如串口/ADC
DMA_DIR_PeripheralSRC
外设发数据到内存，比如串口数据/ADC 数据

上面没讲完的 DMA 结构体，下面继续讲

```
typedef struct
{
    uint32_t DMA_PeripheralBaseAddr;      // 外设地址
    uint32_t DMA_MemoryBaseAddr;          // 存储器地址
    uint32_t DMA_DIR;                    // 传输方向
    uint32_t DMA_BufferSize;             // 传输数目
    uint32_t DMA_PeripheralInc;          // 外设地址增量模式
    uint32_t DMA_MemoryInc;              // 存储器地址增量模式
    uint32_t DMA_PeripheralDataSize;     // 外设数据宽度
    uint32_t DMA_MemoryDataSize;         // 存储器数据宽度
    uint32_t DMA_Mode;                  // 模式选择
    uint32_t DMA_Priority;              // 通道优先级
    uint32_t DMA_M2M;                   // 存储器到存储器模式
} DMA_InitTypeDef;
```

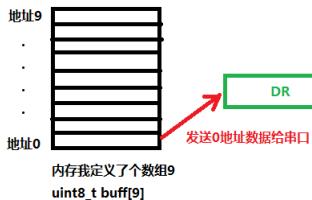
前面我们说了，数据传输的数量和数据类型没关系，只关心数据个数。那么这里就是要关心每一个数据的数据类型是什么，填入每个数据位数宽度。

DMA_MemoryDataSize_Byte
一个数据 8 位(uint_8)
DMA_MemoryDataSize_HalfWord
一个数据 16 位(uint_16)
DMA_MemoryDataSize_Word
一个数据 32 位(uint_32)

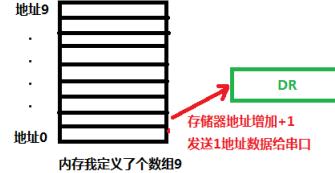


很明显串口每次只能发
1个字节数据，你的
buff数组数据没法一
次发送完

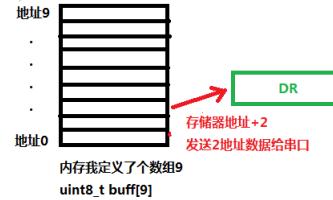
内存我定义了个数组9
uint8_t buff[9]



内存我定义了个数组9
uint8_t buff[9]



内存我定义了个数组9
uint8_t buff[9]



内存我定义了个数组9
uint8_t buff[9]

DR
串口只有一个8位的数
据寄存器

这就是存储器递增(增量
模式)，你把数据全部写
入数组 buff[9]，然后交
给 DMA 自己去让数组地
址+1+2+3。然后把数据
发送给串口

外设地址增量模式和存储器地址增量模式一样，其实外设地址增量模式都是用在内存拷贝内存情况下。因为真正的外设数据寄存器没有这么多地址来做增量。外设数据寄存器地址都是固定的。除非个别比如外部 SRAM，外部 flash。

DMA_Mode_Circular :选择循环传输，就是开启 DMA，buff 数组数据不停的重复发送给外设

DMA_Mode_Normal: 一次性传输，就是开启 DMA，buff 数组循环传完一次给外设就不再传输了。

DMA_InitStruct.DMA_M2M: 这是内存到内存传输功能

填入 DMA_M2M_Enable 使能内存到内存

填入 DMA_M2M_Disable 关闭内存到内存功能

用 DMA 做内存到内存拷贝数据实验

```
#include "stm32f10x_dma.h"
#include "dma.h"
#include "uartprintf.h"
#include "sysclock.h"

const uint32_t flash_buffer[32] = {0,1,2,3,4,5,6,7,8,9,10,
                                  11,12,13,14,15,16,17,18,
                                  19,20,21,22,23,24,25,26,27,
                                  28,29,30,31}; //加入const 这段数组存放在flash

uint32_t readbuff[32];//没有const的全局数组是放在RAM里面
void DMA1_init(void)
{
    DMA_InitTypeDef DMA_InitStruct;
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //开启AHB总线DMA1时钟
    DMA_InitStruct.DMA_PeripheralBaseAddr = (uint32_t)flash_buffer; //传入flash数组的地址
    DMA_InitStruct.DMA_MemoryBaseAddr = (uint32_t)readbuff; //这是要接受数据的地址
    DMA_InitStruct.DMA_DIR = DMA_DIR_PeripheralSRC;
    DMA_InitStruct.DMA_BufferSize = 32; //我们一次传输32个数据，因为数组是32?
    DMA_InitStruct.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
    DMA_InitStruct.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word; //Word表示1个数据32位

    DMA_InitStruct.DMA_MemoryInc = DMA_MemoryInc_Enable;
    DMA_InitStruct.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;

    DMA_InitStruct.DMA_Mode = DMA_Mode_Normal; //flash数组数据只向RAM数组发送一次32个
    DMA_InitStruct.DMA_Priority = DMA_Priority_High; //我们只使用了一个DMA通道，我就选最高优先级
    DMA_InitStruct.DMA_M2M = DMA_M2M_Enable; //因为是flash数组发到RAM数组，所以这里要打开内存到内存模式
    DMA_Init(DMA1_Channel16, &DMA_InitStruct); //选择DMA1 通道 6

    DMA_Cmd(DMA1_Channel16, ENABLE); //打开DMA1 通道16
}

void dma_print(void)
{
    int i = 0;
    for(i=0;i<32;i++)
    {
        printf(" %d ", readbuff[i]);
        delay_ms(100);
    }
}

#include "dma.h"

int main(void)
{
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    delay_ms(5000);
    printf("DMA before\r\n");
    dma_print();

    DMA1_Init(); //DMA初始化
    printf("\r\n");
    printf("DMA later\r\n");
    dma_print();

    while(1);
}
```

DMA before
0
DMA later
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

根据结果显示没有问题，内存 DMA 传输给另一块内存成功。

我们实验内存数据发送到串口外设

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC1	ADC1						
SPI/I ² S		SPI1_RX	SPI1_TX	SPI/I2S2_RX	SPI/I2S2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I ² C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

图 22-2 DMA1 各个通道的请求映像

先确定了 USART1_TX 发送数据的串口在 DM1 通道 4

```
const uint8_t usart_buffer[32] = "abcdefg123456"; //加入const 这段数组存放在flash
/*DMA串口实验*/
void DMA1_UART_init(void)
{
    DMA_InitTypeDef DMA_InitStruct;
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //开启AHB总线DMA1时钟
    DMA_InitStruct.DMA_PeripheralBaseAddr = (uint32_t)(USART1_BASE+0x04); //这是串口外设地址+偏移地址(串口数据寄存器地址)
    DMA_InitStruct.DMA_MemoryBaseAddr = (uint32_t)usart_buffer; //这是发送给串口数据的数组
    DMA_InitStruct.DMA_DIR = DMA_DIR_PeripheralDST; //内存发送数据到串口外设
    DMA_InitStruct.DMA_BufferSize = 32; //我们一次传输32个数据，因为数组是32?
    DMA_InitStruct.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //因为串口只有一个数据寄存器，所以不需要内存递增
    DMA_InitStruct.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte; //Byte表示1个数据8位
    DMA_InitStruct.DMA_MemoryInc = DMA_MemoryInc_Enable; //内存递增要打开，因为数组是递增的
    DMA_InitStruct.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte; //这里的内存数据定义的是8位
    DMA_InitStruct.DMA_Mode = DMA_Mode_Normal; //flash数组数据只向USART发送一次32个
    DMA_InitStruct.DMA_Priority = DMA_Priority_High; //我们只使用了一个DMA通道，我就选最高优先级
    DMA_InitStruct.DMA_M2M = DMA_M2M_Disable; //因为是flash数组发到串口，所以这里要关闭内存到内存模式
    DMA_Init(DMA1_Channel4, &DMA_InitStruct); //根据数据手册串口1 选择DMA1 通道 4
    DMA_ClearFlag(DMA1_FLAG_TC4); //清除发送完成标志位
    DMA_Cmd(DMA1_Channel4, ENABLE); //打开DMA1 通道4
}
```

既然是 DMA 发送数据给外设，那么就要写入外设地址，外设地址都是 32 位的，所以要强转

外设改成目的地

打开 DMA 通道前要清除发送标志位，这个参数看 clearFlag 原型

```
#include "dma.h"
int main(void)
{
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    delay_ms(5000);
    printf("DMA before\r\n");
    DMA1_UART_init(); //开启usart_buffer发送数组数据到串口初始化
    USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE); //这里就是串口请求DMA发送数据到串口，真正的开始发送
    while(1);
    return 0;
}
```

初始化串口 DMA

串口要求 DMA 发送数据一次

DMA before
abcdefg123456

DMA 发送了一次实验成功。

```
DMA_InitStruct.DMA_Mode = DMA_Mode_Circular; //flash数组数据无限循环发送32个  
DMA_InitStruct.DMA_Priority = DMA_Priority_High; //我们只使用了一个DMA通道,
```

我将 DMA 初始化函数改成无限循
环发生数据给串口

```
#include "dma.h"  
  
int main(void)  
{  
    RCC_Configuration(); //初始化时钟  
    USART_Config(115200); //初始化串口  
    delay_ms(5000);  
    printf("DMA before\r\n");  
  
    DMA1_USART_Init(); //开启uart_buffer发送数组数据到串口初始化  
    USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE); //这里就是串口请求DMA发送数据到串口, 真正的开始发送  
  
    while(1);  
    return 0;  
}
```

主函数代码不变

```
DMA before  
abcdefg123456abcdefg123456abcdefg123456abcdefg123  
123456abcdefg123456abcdefg123456abcdefg123456abcdef  
abcdefg123456abcdefg123456abcdefg123456abcdefg1234  
3456abcdefg123456abcdefg123456abcdefg123456abcdefg12  
3456abcdefg123456abcdefg123456abcdefg123456abcdefg1  
3456abcdefg123456abcdefg123456abcdefg123456abcdefg1  
defg123456abcdefg123456abcdefg123456abcdefg123456ab  
cdefg123456abcdefg123456abcdefg123456abcdefg123456a  
bcdefg123456abcdefg123456abcdefg123456abcdefg123456  
56abcdefg123456abcdefg123456abcdefg123456abcdefg123456ab  
cdefg1
```

DMA 就会自己去无限读取 usart_buffer 里面的数据来发送