零死角玩转STM32

与野火同行 乐意惬无边

1 原创教程，完全开源。

2 通俗易懂，详尽解读。

3 由浅入深，结合实操。

4 配套板子，全面玩转。

5 强强联合，不断更新。

野火团队 Wild Fire Team

0、 友情提示

《零死角玩转 STM32》系列教程由初级篇、中级篇、高级篇、系统篇、四个部分组成，根据野火 STM32 开发板旧版教程升级而来，且经过重新深入编写，重新排版，更适合初学者，步步为营，从入门到精通，从裸奔到系统，让您零死角玩转 STM32。M3 的世界，于野火同行，乐意惬无边。

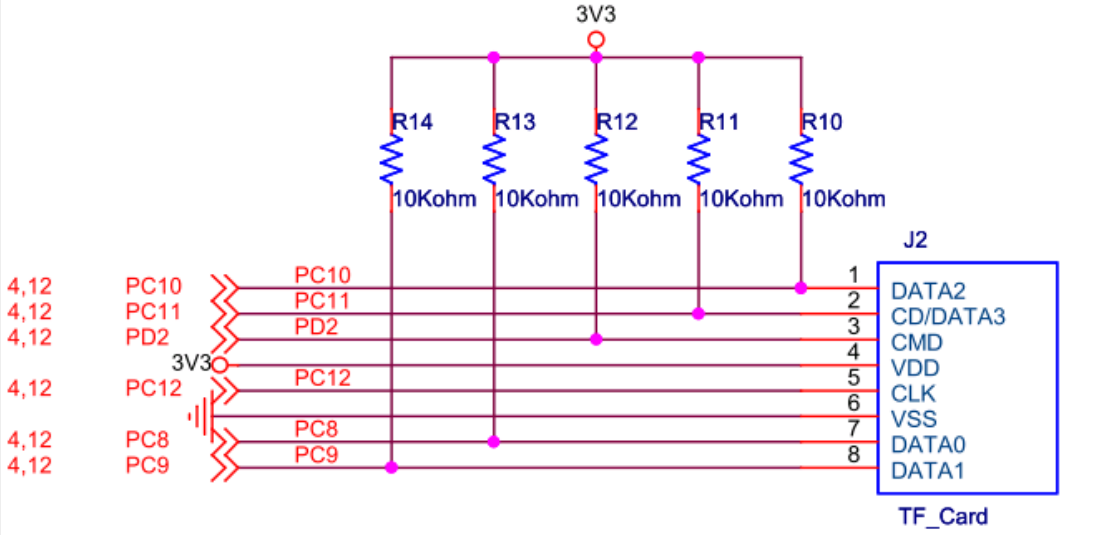
另外，野火团队历时一年精心打造的《STM32 库开发实战指南》将于今年 10 月份由机械工业出版社出版，该书的排版更适于纸质书本阅读以及更有利于查阅资料。内容上会给你带来更多的惊喜。是一本学习 STM32 必备的工具书。敬请期待！

1、SDIO（4bit + DMA、支持 SDHC）

1.1 实验描述及工程文件清单

实验描述	MicroSD 卡(SDIO 模式)测试实验，采用 4bit 数据线模式。没有跑文件系统，只是单纯地读 block 并将测试信息通过串口 1 在电脑的超级终端上 打印出来。
硬件连接	PC12-SDIO-CLK：CLK PC10-SDIO-D2 ：DATA2 PC11-SDIO-D3：CD/DATA3 PD2-SDIO-CMD ：CMD PC8-SDIO-D0：DATA0 PC9-SDIO-D1：DATA1
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/ stm32f10x_sdio.c FWlib/ stm32f10x_dma.c FWlib/ misc.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/usart1.c USER/ sdio_sdcard.c

野火 STM32 开发板 MicroSD 卡硬件原理图：



1.2 SDIO 简介

野火 STM32 开发板的 CPU (STM32F103VET6)具有一个 SDIO 接口。SD/SDIO/MMC 主机接口可以支持 MMC 卡系统规范 4.2 版中的 3 个不同的数据总线模式：1 位(默认)、4 位和 8 位。在 8 位模式下，该接口可以使数据传输速率达到 48MHz，该接口兼容 SD 存储卡规范 2.0 版。SDIO 存储卡规范 2.0 版支持两种数据总线模式：1 位(默认)和 4 位。

目前的芯片版本只能一次支持一个 SD/SDIO/MMC 4.2 版的卡，但可以同时支持多个 MMC 4.1 版或之前版本的卡。除了 SD/SDIO/MMC，这个接口完全与 CE-ATA 数字协议版本 1.1 兼容。

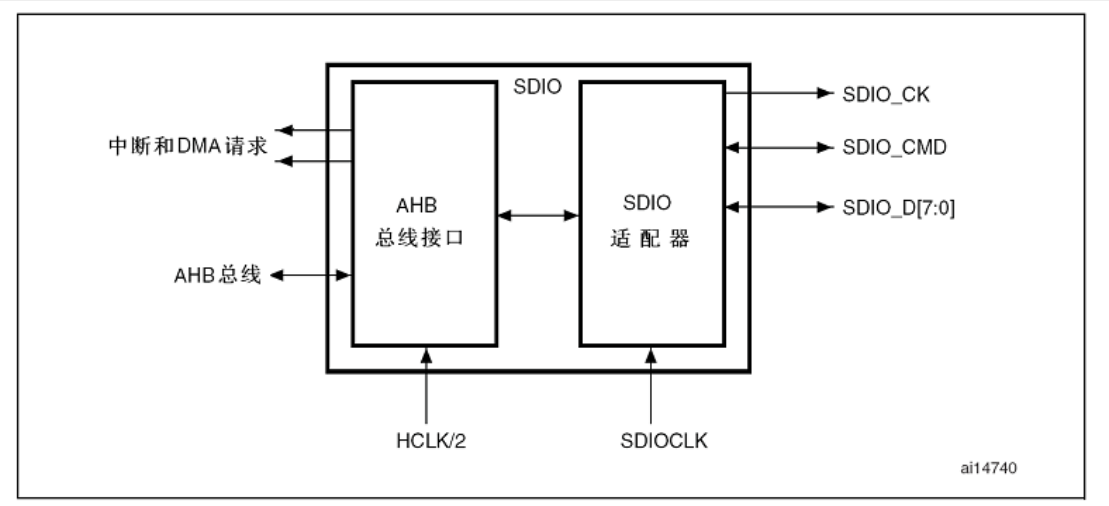
1.3 SD 协议

大多数人原来没有了解过 SD 协议，又看到 SDIO 的驱动有 2000 多行，感觉无从下手。所以野火重新写了这个文档进行详细的解释，帮助大家更快地跨过这道槛。

附资料：《Simplified_Physical_Layer_Spec.pdf》，这个资料包含了 SDIO 协议中 SD 存储卡的部分。

下面野火结合 STM32 的 SDIO，分析 SD 协议，让大家对它先有个大概了解，更具体的说明在代码中展开。

SDIO 接口图



一．从 SDIO 的时钟说起。

SDIO_CK 时钟是通过 PC12 引脚连接到 SD 卡的，是 SDIO 接口与 SD 卡用于同步的时钟。

SDIO 选配器挂载到 AHB 总线上，通过 HCLK 二分频输入到适配器得到 SDIO_CK 的时钟，这时 SDIO_CK = HCLK/（2+CLKDIV）。其中 CLKDIV 是 SDIO_CLK(寄存器)中的 CLKDIV 位。

另外，SDIO_CK 也可以由 SDIOCLK 通过设置 bypass 模式直接得到，这时 SDIO_CK = SDIOCLK=HCLK。

通过下面的库函数来配置时钟：

```
1.  SDIO_Init(&SDIO_InitStructure);
```

对 SD 卡的操作一般是大吞吐量的数据传输，所以采用 DMA 来提高效率，SDIO 采用的是 DMA2 中的通道 4。在数据传输的时候 SDIO 可向 DMA 发出请求。

二．讲解 SDIO 的命令、数据传输方式。

SDIO 的所有命令及命令响应，都是通过 SDIO-CMD 引脚来传输的。

命令只能由 host 即 STM32 的 SDIO 控制器发出。SDIO 协议把命令分成了 11 种，包括基本命令，读写命令还有 ACMD 系列命令等。其中，在发送 ACMD 命令前，要先向卡发送编号为 CMD55 的命令。

参照下面的命令格式图，其中的 start bit，transmission bit ，crc7，endbit，都是由 STM32 中的 SDIO 硬件完成，我们在软件上配置的时候只需要设置 command index 和命令参数 argument。Command index 就是命令索引（编号），如 CMD0，CMD1...被编号成 0，1...。有的命令会包含参数，读命令的地址参数等，这个参数被存放在 argument 段。

SD 卡命令格式

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	start bit	transmission bit	command index	argument	CRC7	end bit

Table 4-16: Command Format

可以通过下面的函数来配置、发送命令：

```
1.  SDIO_SendCommand(&SDIO_CmdInitStructure);    //发送命令
```

SD 卡对 host 的各种命令的回复称为响应，除了 CMD0 命令外，SD 卡在接收到命令都会返回一个响应。对于不同的命令，会有不同的响应格式，共 7 种，分为长响应型（136bit）和短响应型（48bit）。以下图，响应 6（R6）为例：

SD 卡命令响应格式（R6）

Bit position	47	46	[45:40]	[39:8] Argument field		[7:1]	0
Width (bits)	1	1	6	16	16	7	1
Value	'0'	'0'	x	x	x	x	'1'
Description	start bit	transmission bit	command index ('000011')	New published RCA [31:16] of the card	[15:0] card status bits: 23,22,19,12:0 (see Table 4-35)	CRC7	end bit

Table 4-32: Response R6

SDIO 通过 CMD 接收到响应后，硬件去除头尾的信息，把 command index 保存到 SDIO_RESPCMD 寄存器，把 argument field 内容保存存储到 SDIO_RESPx 寄存器中。这两个值可以分别通过下面的库函数得到。

```
1.  SDIO_GetCommandResponse();  //卡返回接收到的命令
2.  SDIO_GetResponse(SDIO_RESP1); //卡返回的 argument field 内容
```

数据写入，读取。请看下面的写数据时序图，在软件上，我们要处理的只是读忙。另外，我们的实验中用的是 Micro SD 卡，有 4 条数据线，默认的时候 SDIO 采用 1 条数据线的传输方式，更改为 4 条数据线模式要通过向卡发送命令来更改。

SD 卡的多块写入时序图

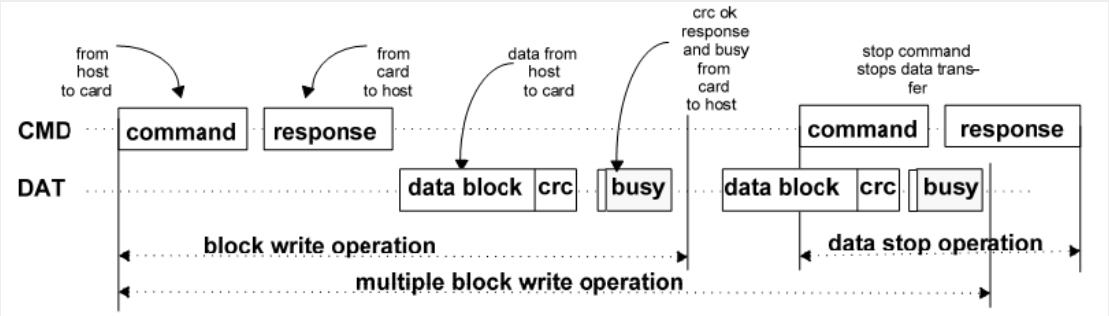


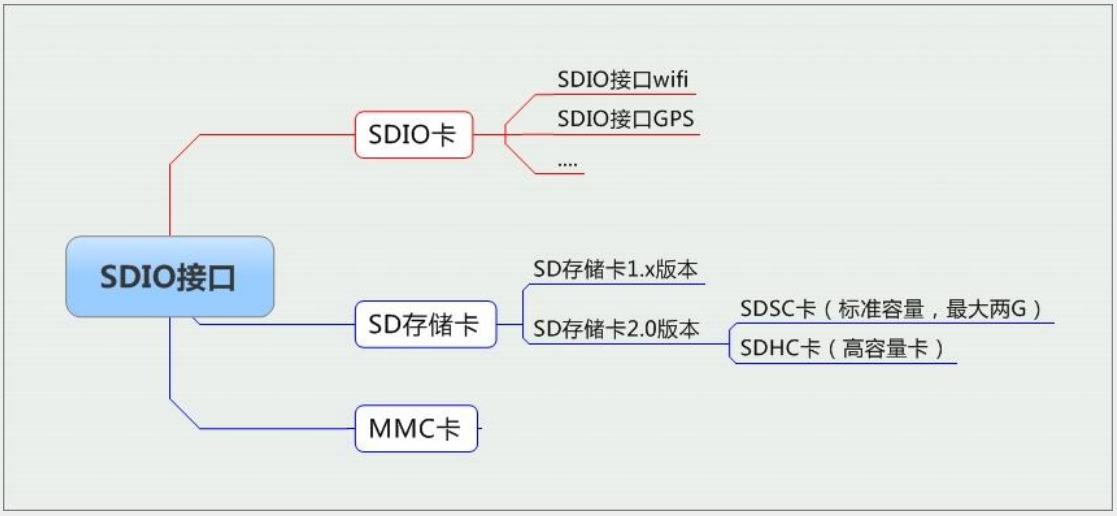
Figure 3-4: (Multiple) Block Write Operation

三．卡的种类。

STM32 的 SDIO 支持 SD 存储卡，SD I/O 卡 ，MMC 卡。

其中 SDI/O 卡与 SD 存储卡是有区别的，SDI/O 卡实际上就是利用 SDIO 接口的一些模块，插入 SD 的插槽中，扩展设备的功能，如：SDI/O wifi, SDI/O cmos 相机等。而 SD 存储卡就是我们平时常见的单纯用于存储数据的卡。

可使用 SDIO 接口类型的卡



本实验中使用的 Micro SD 卡属于 SDSC(标准容量，最大两 G)卡。介绍卡的种类是因为 SD 协议中的命令也支持这三种类型的卡，因此对 STM32 中的 SDIO 接口进行初始化后，上电后就要对接入的卡进行检测、分类，这个过程是通过向卡发送一系列不同的命令，根据卡不同的响应来进行分类。

下面进入代码展开具体讲解。

1.4 代码分析

首先要添加用的库文件，在工程文件夹下 Fwlib 下我们需添加以下库文件：

```
FWlib/stm32f10x_gpio.c

FWlib/stm32f10x_rcc.c

FWlib/stm32f10x_usart.c

FWlib/stm32f10x_sdio.c

FWlib/stm32f10x_dma.c

FWlib/misc.c
```

还要在 stm32f10x_conf.h 中把相应的头文件添加进来：

```
#include "stm32f10x_dma.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_sdio.h"
#include "stm32f10x_usart.h"
#include "misc.h"
```

保持良好的习惯，从 main 函数开始分析：

```
int main(void)
{

    /*进入到 main 函数前，启动文件 startup（startup_stm32f10x_xx.s）已经调用了在
         system_stm32f10x.c 中的 SystemInit()，配置好了系统时钟，在外部晶振 8M 的条件下，
         设置 HCLK = 72M  */

       /* Interrupt Config */
       NVIC_Configuration();

       /* USART1 config */
         USART1_Config();

    /*------------------------------ SD Init ---------------------------
--------- */
    Status = SD_Init();

    printf( "\r\n 这是一个 MicroSD 卡实验(没有跑文件系统).....\r\n " );


  if(Status == SD_OK) //检测初始化是否成功
    {
        printf( " \r\n SD_Init 初始化成功 \r\n " );
     }
  else
    {
        printf("\r\n SD_Init 初始化失败 \r\n" );
        printf("\r\n 返回的 Status 的值为:  %d \r\n",Status );
    }

    printf( " \r\n CardType is : %d ", SDCardInfo.CardType );
    printf( " \r\n CardCapacity is : %d ", SDCardInfo.CardCapacity );

    printf( " \r\n CardBlockSize is : %d ", SDCardInfo.CardBlockSize );
    printf( " \r\n RCA is : %d ", SDCardInfo.RCA);
    printf( " \r\n ManufacturerID is : %d \r\n", SDCardInfo.SD_cid.ManufacturerID );

    SD_EraseTest();     //擦除测试

    SD_SingleBlockTest();  //单块读写测试

    SD_MultiBlockTest();  //多块读写测试

  while (1)
  {}
}
```

main 函数的流程简单明了：

1. 用 NVIC_Configuration()初始化好 SDIO 的中断；
2. 用 USART1_Config()配置好用于返回调试信息的串口，SD_Init()开始进行 SDIO 的初始化；
3. 最后分别用 SD_EraseTest()、SD_SingleBlockTest()、SD_MultiBlockTest()进行擦除，单数据块读写，多数据块读写测试。

下面我们先进入 SDIO 驱动函数的大头——SD_Init()进行分析：

```
1.  /*
2.   * 函数名：SD_Init
3.   * 描述  ：初始化 SD 卡，使卡处于就绪状态(准备传输数据)
4.   * 输入  ：无
5.   * 输出  ：-SD_Error SD 卡错误代码
6.   *          成功时则为 SD_OK
7.   * 调用  ：外部调用
8.   */
9.  SD_Error SD_Init(void)
10. {
11.     /*重置 SD_Error 状态*/
12.   SD_Error errorstatus = SD_OK;
13.
14.   /* SDIO 外设底层引脚初始化 */
15.   GPIO_Configuration();
16.
17.   /*对 SDIO 的所有寄存器进行复位*/
18.   SDIO_DeInit();
19.
20.   /*上电并进行卡识别流程，确认卡的操作电压  */
21.   errorstatus = SD_PowerON();
22.
23.   /*如果上电，识别不成功，返回"响应超时"错误 */
24.   if (errorstatus != SD_OK)
25.   {
26.     /*!< CMD Response TimeOut (wait for CMDSENT flag) */
27.     return(errorstatus);
28.   }
29.
30.   /*卡识别成功，进行卡初始化     */
31.   errorstatus = SD_InitializeCards();
32.
33.   if (errorstatus != SD_OK)    //失败返回
34.   {
35.     /*!< CMD Response TimeOut (wait for CMDSENT flag) */
36.     return(errorstatus);
37.   }
38.
39.   /*!< Configure the SDIO peripheral
40.   上电识别，卡初始化都完成后，进入数据传输模式，提高读写速度
41.   速度若超过 24M 要进入 bypass 模式
42.   !< on STM32F2xx devices, SDIOCLK is fixed to 48MHz
43.   !< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_TRANSFER_CLK_DIV) */

44.   SDIO_InitStructure.SDIO_ClockDiv = SDIO_TRANSFER_CLK_DIV;
45.   SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;          //上
    升沿采集数据
```

```
46.   SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;  //时钟
    频率若超过 24M,要开启此模式
47.   SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;
      //若开启此功能，在总线空闲时关闭 sd_clk 时钟
48.   SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b;
       //1 位模式
49.   SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_
    Disable; //硬件流，若开启，在 FIFO 不能进行发送和接收数据时，数据传输暂停
50.   SDIO_Init(&SDIO_InitStructure);
51.
52.   if (errorstatus == SD_OK)
53.   {
54.     /*----------------- Read CSD/CID MSD registers ------------------
    */
55.     errorstatus = SD_GetCardInfo(&SDCardInfo);    //用来读取 csd/cid 寄存
    器
56.   }
57.
58.   if (errorstatus == SD_OK)
59.   {
60.     /*----------------- Select Card --------------------------------
    */
61.     errorstatus = SD_SelectDeselect((uint32_t) (SDCardInfo.RCA << 16));
     //通过 cmd7   ,rca 选择要操作的卡
62.   }
63.
64.   if (errorstatus == SD_OK)
65.   {
66.     errorstatus = SD_EnableWideBusOperation(SDIO_BusWide_4b);    //开启
    4bits 模式
67.   }
68.
69.   return(errorstatus);
70. }
```

先从整体上了解这个 SD_Init()函数：

1.用 GPIO_Configuration()进行 SDIO 的端口底层配置

2.分别调用了 SD_PowerON()和 SD_InitializeCards()函数，这两个函数共同实现了上面提到的卡检测、识别流程。

3.调用 SDIO_Init(&SDIO_InitStructure)库函数配置 SDIO 的时钟，数据线宽度，硬件流（在读写数据的时候，开启硬件流是和很必要的，可以减少出错）

4. 调用 SD_GetCardInfo(&SDCardInfo)获取 sd 卡的 CSD 寄存器中的内容，在 main 函数里输出到串口的数据就是这个时候从卡读取得到的。

5. 调用 SD_SelectDeselect()选定后面即将要操作的卡。

6.调用 SD_EnableWideBusOperation(SDIO_BusWide_4b)开启 4bit 数据线模式

如果 SD_Init()函数能够执行完整个流程，并且返回值是 SD_OK 的话则说明初始化成功，就可以开始进行擦除、读写的操作了。

下面进入 SD_PowerON()函数，分析完这个函数大家就能了解 SDIO 如何接收、发送命令了。

```
/*
 * 函数名：SD_PowerON
 * 描述  ：确保 SD 卡的工作电压和配置控制时钟
 * 输入  ：无
 * 输出  ：-SD_Error SD 卡错误代码
 *         成功时则为 SD_OK
 * 调用  ：在 SD_Init() 调用
 */
SD_Error SD_PowerON(void)
{
  SD_Error errorstatus = SD_OK;
  uint32_t response = 0, count = 0, validvoltage = 0;
  uint32_t SDType = SD_STD_CAPACITY;

  /*!< 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 */
  /*初始化时的时钟不能大于 400KHz*/
  SDIO_InitStructure.SDIO_ClockDiv = SDIO_INIT_CLK_DIV; /* HCLK = 72MHz, SDIOCLK = 72MHz, SDIO_CK = HCLK/(178 + 2) = 400 KHz */
  SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
  SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;  //不使用 bypass 模式，直接用 HCLK 进行分频得到 SDIO_CK
  SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable; // 空闲时不关闭时钟电源
  SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b;                    //1 位数据线
  SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_Disable;//硬件流
  SDIO_Init(&SDIO_InitStructure);

  /*!< Set Power State to ON */
  SDIO_SetPowerState(SDIO_PowerState_ON);

  /*!< Enable SDIO Clock */
  SDIO_ClockCmd(ENABLE);

   /*下面发送一系列命令,开始卡识别流程*/
  /*!< CMD0: GO_IDLE_STATE ---------------------------------------------------*/
  /*!< No CMD response required */
  SDIO_CmdInitStructure.SDIO_Argument = 0x0;
  SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_GO_IDLE_STATE; //cmd0
  SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_No;  //无响应
  SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
  SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;  //则 CPSM 在开始发送命令之前等待数据传输结束。
  SDIO_SendCommand(&SDIO_CmdInitStructure);          //写命令进命令寄存器

  errorstatus = CmdError();//检测是否正确接收到 cmd0

  if (errorstatus != SD_OK) //命令发送出错，返回
  {
    /*!< CMD Response TimeOut (wait for CMDSENT flag) */
    return(errorstatus);
  }

  /*!< 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)
```

```
57.                 - [7:0]: Check Pattern (recommended 0xAA) */
58.   /*!< CMD Response: R7 */
59.   SDIO_CmdInitStructure.SDIO_Argument = SD_CHECK_PATTERN;    //接收到命令 sd
    会返回这个参数
60.   SDIO_CmdInitStructure.SDIO_CmdIndex = SDIO_SEND_IF_COND;  //cmd8
61.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;      //r7
62.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;             //关闭等待中
    断
63.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
64.   SDIO_SendCommand(&SDIO_CmdInitStructure);
65.
66.   /*检查是否接收到命令*/
67.   errorstatus = CmdResp7Error();
68.
69.   if (errorstatus == SD_OK)      //有响应则 card 遵循 sd 协议 2.0 版本
70.   {
71.     CardType = SDIO_STD_CAPACITY_SD_CARD_V2_0; /*!< SD Card 2.0 ，先把它定
    义会 sdsc 类型的卡*/
72.     SDType = SD_HIGH_CAPACITY;  //这个变量用作 acmd41 的参数，用来询问是 sdsc 卡
    还是 sdhc 卡
73.   }
74.   else  //无响应，说明是 1.x 的或 mmc 的卡
75.   {
76.     /*!< CMD55 */
77.     SDIO_CmdInitStructure.SDIO_Argument = 0x00;
78.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
79.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
80.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
81.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
82.     SDIO_SendCommand(&SDIO_CmdInitStructure);
83.     errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
84.   }
85.   /*!< CMD55 */      //为什么在 else 里和 else 外面都要发送 CMD55?
86.   //发送 cmd55，用于检测是 sd 卡还是 mmc 卡，或是不支持的卡
87.   SDIO_CmdInitStructure.SDIO_Argument = 0x00;
88.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
89.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r1
90.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
91.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
92.   SDIO_SendCommand(&SDIO_CmdInitStructure);
93.   errorstatus = CmdResp1Error(SD_CMD_APP_CMD);  //是否响应，没响应的是 mmc 或
    不支持的卡
94.
95.   /*!< If errorstatus is Command TimeOut, it is a MMC card */
96.   /*!< If errorstatus is SD_OK it is a SD card: SD card 2.0 (voltage rang
    e mismatch)
97.      or SD card 1.x */
98.   if (errorstatus == SD_OK) //响应了 cmd55，是 sd 卡，可能为 1.x,可能为 2.0
99.   {
100.       /*下面开始循环地发送 sdio 支持的电压范围，循环一定次数*/
101.
102.       /*!< SD CARD */
103.       /*!< Send ACMD41 SD_APP_OP_COND with Argument 0x80100000 */
104.       while ((!validvoltage) && (count < SD_MAX_VOLT_TRIAL))
105.       {
106.       //因为下面要用到 ACMD41，是 ACMD 命令，在发送 ACMD 命令前都要先向卡发送
    CMD55
107.         /*!< SEND CMD55 APP_CMD with RCA as 0 */
108.         SDIO_CmdInitStructure.SDIO_Argument = 0x00;
109.         SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;    //CMD55

110.         SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
111.         SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
112.         SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
113.         SDIO_SendCommand(&SDIO_CmdInitStructure);
```

```
114.
115.          errorstatus = CmdResp1Error(SD_CMD_APP_CMD); //检测响应
116.
117.          if (errorstatus != SD_OK)
118.          {
119.            return(errorstatus);//没响应 CMD55，返回
120.          }
121.          //acmd41，命令参数由支持的电压范围及 HCS 位组成，HCS 位置一来区分卡是 SDSc
    还是 sdhc
122.          SDIO_CmdInitStructure.SDIO_Argument = SD_VOLTAGE_WINDOW_SD | SDT
    ype;     //参数为主机可供电压范围及 hcs 位
123.          SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SD_APP_OP_COND;
124.          SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;  //r3

125.          SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
126.          SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
127.          SDIO_SendCommand(&SDIO_CmdInitStructure);
128.
129.          errorstatus = CmdResp3Error();     //检测是否正确接收到数据
130.          if (errorstatus != SD_OK)
131.          {
132.            return(errorstatus);  //没正确接收到 acmd41，出错，返回
133.          }
134.           /*若卡需求电压在 SDIO 的供电电压范围内，会自动上电并标志 pwr_up 位*/
135.          response = SDIO_GetResponse(SDIO_RESP1);    //读取卡寄存器，卡状
    态
136.          validvoltage = (((response >> 31) == 1) ? 1 : 0); //读取卡的 ocr
    寄存器的 pwr_up 位，看是否已工作在正常电压
137.          count++;              //计算循环次数
138.        }
139.        if (count >= SD_MAX_VOLT_TRIAL) //循环检测超过一定次数还没上电
140.        {
141.          errorstatus = SD_INVALID_VOLTRANGE;       //SDIO 不支持 card 的供电电
    压
142.          return(errorstatus);
143.        }
144.                  /*检查卡返回信息中的 HCS 位*/
145.        if (response &= SD_HIGH_CAPACITY)   //判断 ocr 中的 ccs 位 ，如果是 sdsc
    卡则不执行下面的语句
146.        {
147.          CardType = SDIO_HIGH_CAPACITY_SD_CARD;  //把卡类型从初始化的 sdsc 型
    改为 sdhc 型
148.        }
149.
150.      }/*!< else MMC Card */
151.
152.      return(errorstatus);
153.    }
```

这个函数的流程就是卡的上电、识别操作，如下图：

卡的上电，识别流程：

截图来自《Simplified_Physical_Layer_Spec.pdf》 page27

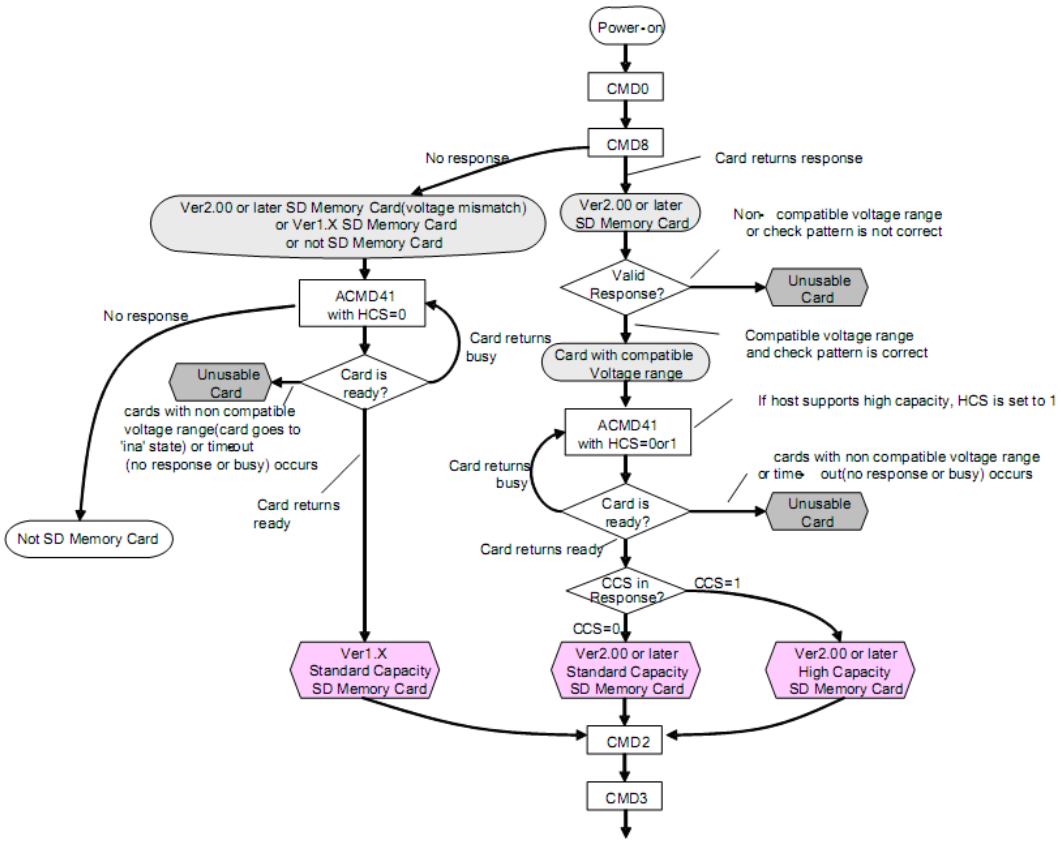


Figure 4-2: Card Initialization and Identification Flow (SD mode)

代码中所有的判断语句都是根据这个图的各个识别走向展开的，最终把卡分为 1.0 版的 SD 存储卡，2.0 版的 SDSC 卡和 2.0 版的 SDHC 卡。

在这个代码流程中有两点要注意一下：

1.初始化的时钟。SDIO_CK 的时钟分为两个阶段，在初始化阶段 SDIO_CK 的频率要小于 400KHz，初始化完成后可把 SDIO_CK 调整成高速模式，高速模式时超过 24M 要开启 bypass 模式，对于 SD 存储卡即使开启 bypass，最高频率不能超过 25MHz。

2.CMD8 命令。

CMD8 命令格式。

Table 4-15 shows the format of CMD8.

Bit position	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
Width (bits)	1	1	6	20	4	8	7	1
Value	'0'	'1'	'001000'	'00000h'	x	x	x	'1'
Description	start bit	transmission bit	command index	reserved bits	voltage supplied (VHS)	check pattern	CRC7	end bit

CMD8 命令中的 VHS 是用来确认主机 SDIO 是否支持卡的工作电压的。Check pattern 部分可以是任何数值，若 SDIO 支持卡的工作电压，卡会把接收到的 check pattern 数值原样返回给主机。

CMD8 命令的响应格式 R7：

Bit position	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
Width (bits)	1	1	6	20	4	8	7	1
Value	'0'	'0'	'001000'	'00000h'	x	x	x	'1'
Description	start bit	transmission bit	command index	reserved bits	voltage accepted	echo-back of check pattern	CRC7	end bit

Table 4-33: Response R7

在驱动程序中调用了 CmdResp7Error()来检验卡接收命令后的响应。

3.ACMD41 命令。

这个命令也是用来进一步检查 SDIO 是否支持卡的工作电压的，协议要它在调用它之前必须先调用 CMD8，另外还可以通过它命令参数中的 HCS 位来区分卡是 SDHC 卡还是 SDSC 卡。

确认工作电压时循环地发送 ACMD41，发送后检查在 SD 卡上的 OCR 寄存器中的 pwr_up 位，若 pwr_up 位置为 1，表明 SDIO 支持卡的工作电压，卡开始正常工作。

同时把 ACMD41 中的命令参数 HCS 位置 1，卡正常工作的时候检测 OCR 寄存器中的 CCS 位，若 CCS 位为 1 则说明该卡为 SDHC 卡，为零则为 SDSC 卡。

因为 ACMD41 命令属于 ACMD 命令，在发送 ACMD 命令前都要先发送 CMD55.

ACMD41 命令格式

ACMD INDEX	type	argument	resp	abbreviation	command description
ACMD41	bcr	[31]reserved bit [30]HCS(OCR[30]) [29:24]reserved bits [23:0] V_{DD} Voltage Window(OCR[23:0])	R3	SD_SEND_OP_COND	Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response on the CMD line. HCS is effective when card receives SEND_IF_COND command. Reserved bit shall be set to '0'. CCS bit is assigned to OCR[30].

ACMD41 命令的响应（R3），返回的是 OCR 寄存器的值

4.9.4 R3 (OCR register)

Code length is 48 bits. The contents of the OCR register are sent as a response to ACMD41.

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'0'	'111111'	x	'1111111'	'1'
Description	start bit	transmission bit	reserved	OCR register	reserved	end bit

Table 4-31: Response R3

OCR 寄存器的内容

OCR bit position	OCR Fields Definition
0-3	reserved
4	reserved
5	reserved
6	reserved
7	Reserved for Low Voltage Range
8	reserved
9	reserved
10	reserved
11	reserved
12	reserved
13	reserved
14	reserved
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24-29	reserved
30	Card Capacity Status (CCS)[1]
31	Card power up status bit (busy)[2]

VDD Voltage Window

1) This bit is valid only when the card power up status bit is set.
2) This bit is set to LOW if the card has not finished the power up routine.

SD 卡上电确认成功后，进入 SD_InitializeCards()函数：

```
1.  /*
2.   * 函数名：SD_InitializeCards
3.   * 描述  ：初始化所有的卡或者单个卡进入就绪状态
4.   * 输入  ：无
5.   * 输出  ：-SD_Error SD 卡错误代码
6.   *          成功时则为 SD_OK
7.   * 调用  ：在 SD_Init() 调用，在调用 power_on（）上电卡识别完毕后，调用此函数进行卡初始化
8.   */
9.  SD_Error SD_InitializeCards(void)
10. {
11.   SD_Error errorstatus = SD_OK;
12.   uint16_t rca = 0x01;
13.
14.   if (SDIO_GetPowerState() == SDIO_PowerState_OFF)
15.   {
16.     errorstatus = SD_REQUEST_NOT_APPLICABLE;
17.     return(errorstatus);
18.   }
19.
20.   if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)//判断卡的类型
21.   {
22.     /*!< Send CMD2 ALL_SEND_CID */
23.     SDIO_CmdInitStructure.SDIO_Argument = 0x0;
24.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_ALL_SEND_CID;
25.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
26.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
27.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
28.     SDIO_SendCommand(&SDIO_CmdInitStructure);
29.
30.     errorstatus = CmdResp2Error();
31.
32.     if (SD_OK != errorstatus)
33.     {
34.       return(errorstatus);
35.     }
36.
37.     CID_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
38.     CID_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
39.     CID_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
40.     CID_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
41.   }
42.
43.     /*下面开始 SD 卡初始化流程*/
44.   if ((SDIO_STD_CAPACITY_SD_CARD_V1_1 == CardType) ||  (SDIO_STD_CAPACITY_SD_CARD_V2_0 == CardType) ||  (SDIO_SECURE_DIGITAL_IO_COMBO_CARD == CardType)
45.       ||  (SDIO_HIGH_CAPACITY_SD_CARD == CardType))  //使用的是 2.0 的卡
46.   {
47.     /*!< Send CMD3 SET_REL_ADDR with argument 0 */
48.     /*!< SD Card publishes its RCA. */
49.     SDIO_CmdInitStructure.SDIO_Argument = 0x00;
50.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_REL_ADDR;  //cmd3
51.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r6
52.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
53.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
54.     SDIO_SendCommand(&SDIO_CmdInitStructure);
55.
56.     errorstatus = CmdResp6Error(SD_CMD_SET_REL_ADDR, &rca); //把接收到的卡相对地址存起来。
57.
58.     if (SD_OK != errorstatus)
59.     {
60.       return(errorstatus);
```

```
61.     }
62.   }
63.
64.   if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)
65.   {
66.     RCA = rca;
67.
68.     /*!< Send CMD9 SEND_CSD with argument as card's RCA */
69.     SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)(rca << 16);
70.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SEND_CSD;
71.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
72.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
73.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
74.     SDIO_SendCommand(&SDIO_CmdInitStructure);
75.
76.     errorstatus = CmdResp2Error();
77.
78.     if (SD_OK != errorstatus)
79.     {
80.       return(errorstatus);
81.     }
82.
83.     CSD_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
84.     CSD_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
85.     CSD_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
86.     CSD_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
87.   }
88.
89.   errorstatus = SD_OK; /*!< All cards get intialized */
90.
91.   return(errorstatus);
92. }
```

这个函数向卡发送了 CMD2 和 CMD3 命令

1.CMD2

CMD2 命令是要求卡返回它的 CID 寄存器的内容。

命令的响应格式（R2）。

Bit position	135	134	[133:128]	[127:1]	0
Width (bits)	1	1	6	127	1
Value	'0'	'0'	'111111'	x	'1'
Description	start bit	transmission bit	reserved	CID or CSD register incl. internal CRC7	end bit

Table 4-30: Response R2

因为命令格式是 136 位的，属于长响应。软件接收的信息有 128 位。在长响应的时候通过 SDIO_GetResponse(SDIO_RESP4);中的不同参数来获取 CID 中的不同数

据段的数据。

表151 响应类型和SDIO_RESPx寄存器

寄存器	短响应	长响应
SDIO_RESP1	卡状态[31:0]	卡状态[127:96]
SDIO_RESP2	不用	卡状态[95:64]
SDIO_RESP3	不用	卡状态[63:32]
SDIO_RESP4	不用	卡状态[31:1]

总是先收到卡状态的最高位，SDIO_RESP3寄存器的最低位始终为0。

2.CMD3

CMD3 命令是要求卡向主机发送卡的相对地址。在接有多个卡的时候，主机要求接口上的卡重新发一个相对地址，这个地址跟卡的实际 ID 不一样。比如接口上接了 5 个卡，这 5 个卡的相对地址就分别为 1，2，3，4，5.以后主机 SDIO 对这几个卡寻址就直接使用相对地址。这个地址的作用就是为了寻址更加简单。

接下来我们回到 SD_Init()函数。分析到这里大家应该对 SDIO 的命令发送和响应比较清楚了。在 SD_InitializeCards()之后的 SD_GetCardInfo(&SDCardInfo)、SD_SelectDeselect()和 SD_EnableWideBusOperation(SDIO_BusWide_4b)的具体实现就不再详细分析了，实际就是发送相应的命令，对卡进行相应的操作。

接下来分析 main 函数中的 SD_MultiBlockTest()多块数据读写函数，让大家了解 SDIO 是怎样传输数据的。

```
1.  /*
2.   * 函数名: SD_MultiBlockTest
3.   * 描述  :      多数据块读写测试
4.   * 输入  : 无
5.   * 输出  : 无
6.   */
7.  void SD_MultiBlockTest(void)
8.  {
9.    /*--------------- Multiple Block Read/Write --------------------
    */
10.   /* Fill the buffer to send */
11.   Fill_Buffer(Buffer_MultiBlock_Tx, MULTI_BUFFER_SIZE, 0x0);
12.
13.   if (Status == SD_OK)
```

```
14.   {
15.     /* Write multiple block of many bytes on address 0 */
16.     Status = SD_WriteMultiBlocks(Buffer_MultiBlock_Tx, 0x00, BLOCK_SIZ
    E, NUMBER_OF_BLOCKS);
17.     /* Check if the Transfer is finished */
18.     Status = SD_WaitWriteOperation();
19.     while(SD_GetStatus() != SD_TRANSFER_OK);
20.   }
21.
22.   if (Status == SD_OK)
23.   {
24.     /* Read block of many bytes from address 0 */
25.     Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00, BLOCK_SIZE
    , NUMBER_OF_BLOCKS);
26.     /* Check if the Transfer is finished */
27.     Status = SD_WaitReadOperation();
28.     while(SD_GetStatus() != SD_TRANSFER_OK);
29.   }
30.
31.   /* Check the correctness of written data */
32.   if (Status == SD_OK)
33.   {
34.     TransferStatus2 = Buffercmp(Buffer_MultiBlock_Tx, Buffer_MultiBloc
    k_Rx, MULTI_BUFFER_SIZE);
35.   }
36.
37.   if(TransferStatus2 == PASSED)
38.     printf("\r\n 多块读写测试成功！ " );
39.
40.   else
41.     printf("\r\n 多块读写测试失败！ " );
42.
43. }
```

把这个函数拿出来分析最重要的一点就是让大家注意在调用了 SD_WriteMultiBlocks()这一类读写操作的函数后，一定要调用 SD_WaitWriteOperation()[在读数据时调用 SD_WaitReadOperation]和 SD_GetStatus()来确保数据传输已经结束再进行其它操作。其中的 SD_WaitWriteOperation()是用来等待 DMA 把缓冲的数据传输到 SDIO 的 FIFO 的；而 SD_GetStatus()是用来等待卡与 SDIO 之间传输数据完毕的。

最后进入 SD_WriteMultiBlocks（）函数分析：

```
1.  /*
2.   * 函数名：SD_WriteMultiBlocks
3.   * 描述   ：从输入的起始地址开始，向卡写入多个数据块，
4.             只能在 DMA 模式下使用这个函数
5.      注意：调用这个函数后一定要调用
6.               SD_WaitWriteOperation（）来等待 DMA 传输结束
7.               和   SD_GetStatus() 检测卡与 SDIO 的 FIFO 间是否已经完成传输
8.   * 输入   ：
9.             * @param  WriteAddr: Address from where data are to be read.

10.            * @param  writebuff: pointer to the buffer that contain the
    data to be transferred.
11.            * @param  BlockSize: the SD card Data block size. The Block
    size should be 512.
```

```
12.           * @param  NumberOfBlocks: number of blocks to be written.
13.  * 输出  : SD 错误类型
14.  */
15. SD_Error SD_WriteMultiBlocks(uint8_t *writebuff, uint32_t WriteAddr, u
    int16_t BlockSize, uint32_t NumberOfBlocks)
16. {
17.   SD_Error errorstatus = SD_OK;
18.   __IO uint32_t count = 0;
19.
20.   TransferError = SD_OK;
21.   TransferEnd = 0;
22.   StopCondition = 1;
23.
24.   SDIO->DCTRL = 0x0;
25.
26.   if (CardType == SDIO_HIGH_CAPACITY_SD_CARD)
27.   {
28.     BlockSize = 512;
29.     WriteAddr /= 512;
30.   }
31.
32.     /*******************add，没有这一段容易卡死在 DMA 检测中
    ************************************/
33.     /*!< Set Block Size for Card，cmd16,若是 sdsc 卡，可以用来设置块大小，若
    是 sdhc 卡，块大小为 512 字节，不受 cmd16 影响 */
34.   SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) BlockSize;
35.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
36.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;    //r1
37.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
38.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
39.   SDIO_SendCommand(&SDIO_CmdInitStructure);
40.
41.   errorstatus = CmdResp1Error(SD_CMD_SET_BLOCKLEN);
42.
43.   if (SD_OK != errorstatus)
44.   {
45.     return(errorstatus);
46.   }
47.  /*************************************************************************
    *************/
48.
49.   /*!< To improve performance  */
50.   SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) (RCA << 16);
51.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD; // cmd55
52.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
53.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
54.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
55.   SDIO_SendCommand(&SDIO_CmdInitStructure);
56.
57.
58.   errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
59.
60.   if (errorstatus != SD_OK)
61.   {
62.     return(errorstatus);
63.   }
64.   /*!< To improve performance *///  pre-erased，在多块写入时可发送此命令进
    行预擦除
65.   SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)NumberOfBlocks;  //
    参数为将要写入的块数目
66.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCK_COUNT;  //cmd
    23
67.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
68.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
69.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
70.   SDIO_SendCommand(&SDIO_CmdInitStructure);
71.
```

```
72.   errorstatus = CmdResp1Error(SD_CMD_SET_BLOCK_COUNT);
73.
74.   if (errorstatus != SD_OK)
75.   {
76.     return(errorstatus);
77.   }
78.
79.
80.   /*!< Send CMD25 WRITE_MULT_BLOCK with argument data address */
81.   SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)WriteAddr;
82.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_WRITE_MULT_BLOCK;
83.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
84.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
85.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
86.   SDIO_SendCommand(&SDIO_CmdInitStructure);
87.
88.   errorstatus = CmdResp1Error(SD_CMD_WRITE_MULT_BLOCK);
89.
90.   if (SD_OK != errorstatus)
91.   {
92.     return(errorstatus);
93.   }
94.
95.   SDIO_DataInitStructure.SDIO_DataTimeOut = SD_DATATIMEOUT;
96.   SDIO_DataInitStructure.SDIO_DataLength = NumberOfBlocks * BlockSize;

97.   SDIO_DataInitStructure.SDIO_DataBlockSize = (uint32_t) 9 << 4;
98.   SDIO_DataInitStructure.SDIO_TransferDir = SDIO_TransferDir_ToCard;

99.   SDIO_DataInitStructure.SDIO_TransferMode = SDIO_TransferMode_Block;

100.        SDIO_DataInitStructure.SDIO_DPSM = SDIO_DPSM_Enable;
101.        SDIO_DataConfig(&SDIO_DataInitStructure);
102.
103.        SDIO_ITConfig(SDIO_IT_DATAEND, ENABLE);
104.        SDIO_DMACmd(ENABLE);
105.        SD_DMA_TxConfig((uint32_t *)writebuff, (NumberOfBlocks * Bloc
    kSize));
106.
107.        return(errorstatus);
108.      }
```

写操作在发送正式的多块写入命令 CMD25 前调用了 CMD23 进行预写，这样有利于提高写入的速度。在代码的最后调用了 SDIO_ITConfig()，SDIO 的数据传输结束中断就是这个时候开启的，数据传输结束时，就进入到 stm32f10x_it.c 文件中的中断服务函数 SDIO_IRQHandler()中处理了，中断服务函数主要就是负责清中断。

最后讲一下官方原版的驱动中的一个 bug。

在官方原版的 SDIO 驱动的 SD_ReadBlock()、SD_ReadMultiBlocks()、SD_WriteBlock()和 SD_WriteMultiBlocks()这几个函数中，发送读写命令前，漏掉了发送一个 CMD16 命令，这个命令用于设置读写 SD 卡的块大小。缺少这个命令很容易导致程序运行时卡死在循环检测 DMA 传输结束的代码中，网上很多

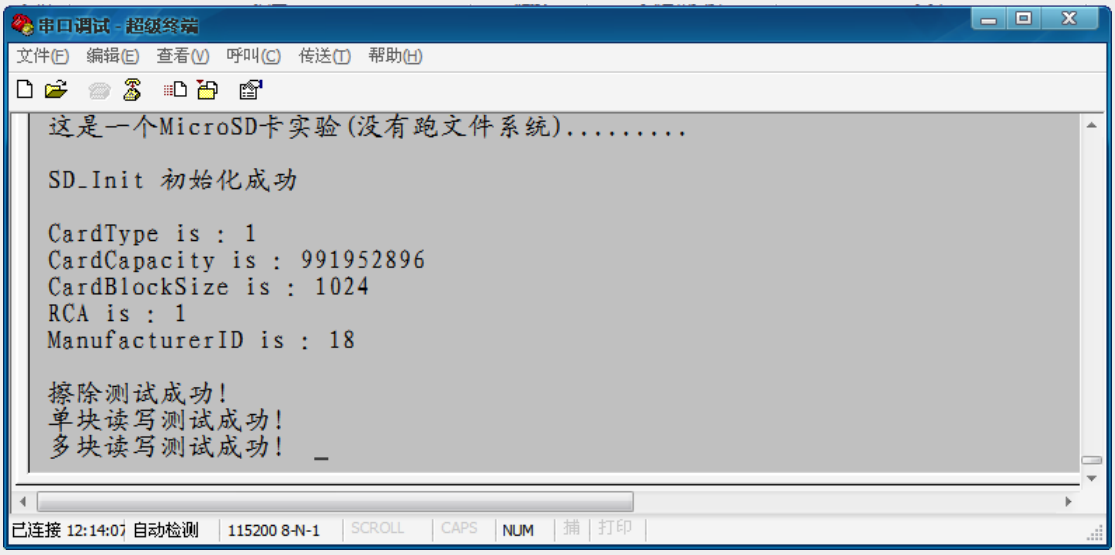
人直接移植 ST 官方例程时，用 3.5 版库函数和这个 4.5 版的 SDIO 驱动移植失败，就是缺少了这段用 CMD16 设置块大小的代码。

到这里，终于讲解完毕啦！这个讲解如果能让你从对 SDIO 一无所知到大概了解的话，我的目标就达到啦，想要更深入了解还是要好好地配合这个例程中我在代码中的注释和附带资料 SD2.0 协议《Simplified_Physical_Layer_Spec.pdf》好好研究一番! ^_^

注意：这个例程是没有跑文件系统的，而是直接就去读卡的 block，这样的话就会破坏卡的分区，在实验完成之后，你再把卡插到电脑上时，电脑会提示你要重新初始化卡，这是正常想象，并不是本实验把你的卡弄坏了，如果卡原来有资料的请先把数据备份了再进行测试。但跑文件系统时就不会出现这种问题，有关文件系统的操作将在下一讲的教程中讲解。

1.5 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，插上 MicroSD 卡(我用的是 1G ，经测试，本驱动也适用于 2G 以上的卡（sdhc 卡）)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：



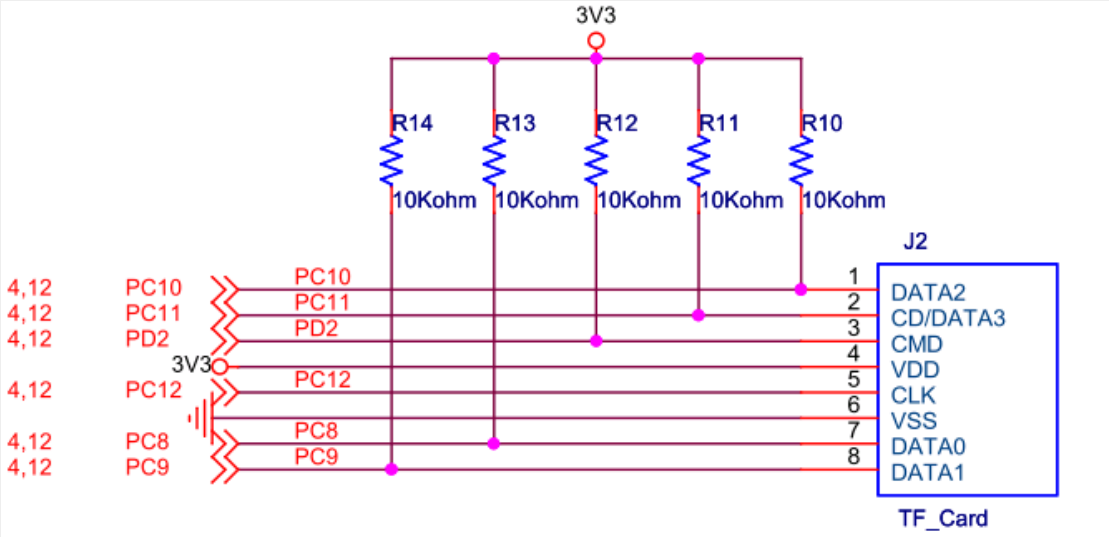
2、FatFs (Rev-R0.09)

2.1 实验描述及工程文件清单

实验描述	MicroSD 卡文件系统 FATFS R0.07C 测试实验。在 MicroSD 卡里面创建一个 DEMO.TXT 文本文件，在文件里面写入字符串“感谢您选用 野火 STM32 开发板 ！ ^_^”，然后通过串口将这些内容打印在电脑的超级终端上。这个更新版本的代码增加了简体中文和长文件名的支持，采用 SDIO 的 4bit+DMA 模式。并图解了文件系统移植的全部过程。
硬件连接	PC12-SDIO-CLK：CLK PC10-SDIO-D2 ：DATA2 PC11-SDIO-D3：CD/DATA3 PD2-SDIO-CMD ： CMD PC8-SDIO-D0：DATA0 PC9-SDIO-D1：DATA1
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/ stm32f10x_sdio.c FWlib/ stm32f10x_dma.c FWlib/ misc.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c

	USER/usart1.c USER/ sdio_sdcard.c
文件系统文件	ff9/diskio.c ff9/ff.c ff9/cc936.c

野火 STM32 开发板 MicroSD 卡硬件原理图：



2.2 实验简介

本实验是在上一讲《SDIO（4bit + DMA）》的基础上讲解的，只有上一讲的实验成功了，文件系统才能跑起来。有关卡的底层的初始化，这里不再详述，这里着重讲解文件系统的移植和文件系统的应用。这个文档是更新版本，采用的文件系统的版本是目前最新的 R0.09。 文件系统的源码可以从 fatfs 官网下载 http://elm-chan.org/fsw/ff/00index_e.html

2.3 FatFS 文件系统简介

FAFFS 是面向小型嵌入式系统的一种通用的 FAT 文件系统。FATFS 完全是由 AISI C 语言编写并且完全独立于底层的 I/O 介质。因此它可以很容易地不加修改地移植到其他的处理器当中，如 8051、PIC、AVR、SH、Z80、H8、ARM

等。FATFS 支持 FAT12、FAT16、FAT32 等格式，所以我们利用前面写好的 SDIO 驱动，把 FATFS 文件系统代码移植到工程之中，就可以利用文件系统的各种函数，对已格式化的 SD 卡进行读写文件了。

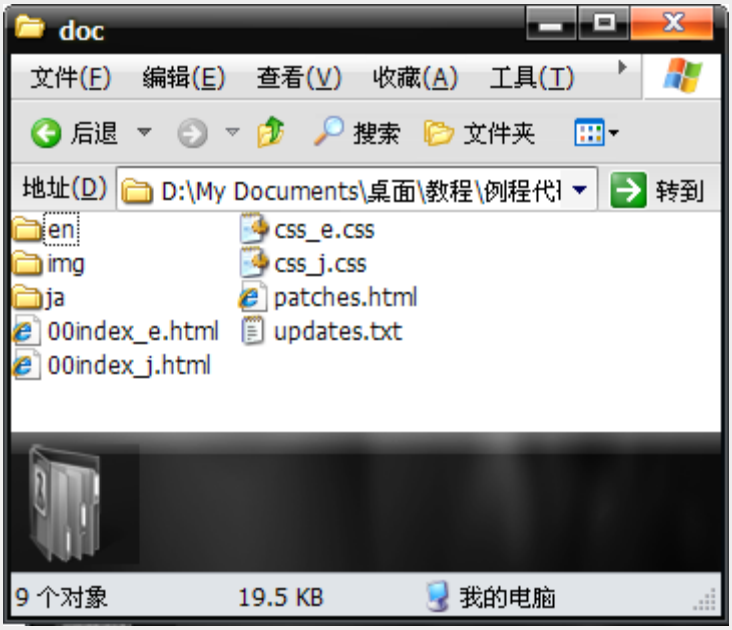
本实验是将 FATFS 移植到野火 STM32 开发板中，CPU 为 STM32F103VET6，是采用 ARM 公司最新内核 ARMV7 的一款单片机。

2.4 移植前的工作

2.4.1 分析 FATFS 的目录结构

在移植 FATFS 文件系统之前，我们先要到 FAT 的官网获取源码，版本为 R0.07C。解压之后可看到里面有 doc 和 src 这两个文件夹。doc 文件夹里面是一些使用文档， src 里面是文件系统的源码。

打开 doc 文件夹，可看到如下文件目录：



其中 en 和 ja 这两个文件夹里面是编译好的 html 文档，讲的是 FATFS 里面各个函数的使用方法，这些函数就如 LINUX 下的系统调用，是封装得非常好的函数，利用这些函数我们就可以操作我们的 MicroSD 卡了。有关具体的函数我们在用到的时候再讲解。这两个文件夹的唯一区别就是 en 文件夹下的文档是英文的，ja 文件夹下的是日文的。偏偏就是没中文的，真狗血呀。00index_e.html 是一些关于 FATFS 的英文简介，updates.txt 是 FATFS 的更新信息，至于其他几个文件可以不看。

打开 src 文件夹，可看到如下目录：

option 文件夹下是一些可选的外部 c 文件，包含了多语言支持需要用到的文件和转换函数。

00readme.txt 说明了当前目录下 diskio.c 、diskio.h、ff.c、ff.h、integer.h 的功用、涉及了 FATFS 的版权问题(是自由软件)，还讲到了 FATFS 的版本更新信息。

integer.h：是一些数值类型定义

diskio.c ：底层磁盘的操作函数，这些函数需要用户自己实现

ff.c ：独立于底层介质操作文件的函数，完全由 ANSI C 编写

cc936.c :简体中文支持所需要添加的文件，包含了简体中文的 GBK 和转换函数。

只要添加进来就行，这个文件不需要修改。

ffconf.h:这个头文件包含了对文件系统的各种配置，如需要支持简体中文要把_CODE_PAGE 的宏改成 936 并把上面的 cc936.c 文件加入到工程之中

建议阅读这些源码的顺序为：integer.h -> diskio.c -> ff.c 。

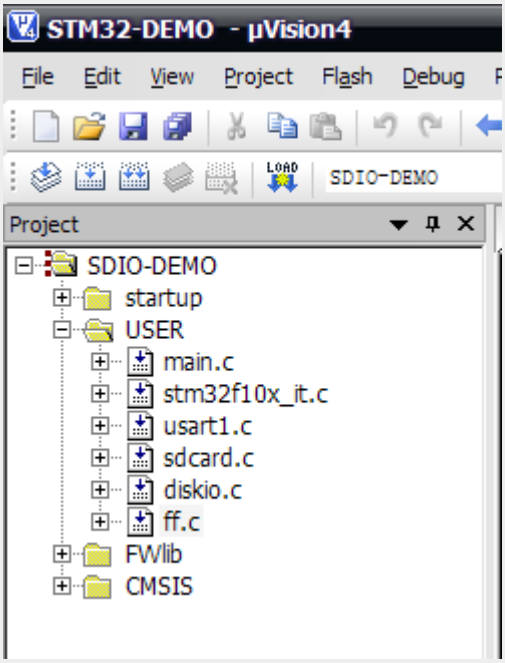
关于具体源码的分析不是我能力所及呀，大家就自己研究吧。我的主要工作是带领大家把这个文件系统移植到我们的开发板上，让这个文件系统先跑起来，这样才是硬道理呀。文件系统工作起来了的话，源码的分析那自然是大家的活啦。

2.5 开始移植

首先我们要获取一个完全没有修改过的文件系统源码，然后在 10-MicroSD 卡这个文件夹下的实验代码下移植，这个实验代码实现的是卡的底层的块操作。注意，我们在移植这个文件系统的过程中会尽量保持文件系统源码的纯净，尽量做到在修改最少量的源码的情况下移植成功。

首先将 integer.h、diskio.h、diskio.c、ff.h、ff.c 添加到工程目录下的 USER 文件夹下，如下截图：

然后并回到 MDK 界面下将 diskio.c 、ff.c 这两个文件添加到 USER 目录下，如下截图：



因为我们要用到这两个 c 文件，所以我们在 main.c 中将这两个 c 文件对应的头文件 diskio.h 、ff.h 包含进来，如下截图：

```
/********************* (C) COPYRIGHT 2011 野
 * 文件名  : main.c
 * 描述    : MicroSD卡文件系统 FATFS R0.07
 * 实验平台: 野火STM32开发板
 * 库版本  : ST3.0.0
 *
 * 作者    : fire  QQ: 313303034
 * 博客    : firestm32.blog.chinaunix.net
*****************************************/
#include "stm32f10x.h"
#include "usart1.h"
#include "sdcard.h"
#include "ff.h"
#include "diskio.h"

#include <string.h>
```

好嘞，下面我们开始编译，这时会出现如下错误：

```
Build Output
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
compiling diskio.c...
..\CMSIS\stm32f10x.h(237): error:  #101: "FALSE" has already been declared in the current scope
..\CMSIS\stm32f10x.h(237): error:  #101: "TRUE" has already been declared in the current scope
compiling ff.c...
Target not created
```

意思是说 FALSE 跟 TRUE 这两个变量已经定义过了，为什么会出现这个错误呢？因为在 integer.h 和我们的 M3 库头文件 stm32f10x.h 中都定义了这两个变量，所以就产生了重复定义：

integer.h

```
31  /* Boolean type */
32  typedef enum { FALSE = 0, TRUE } BOOL;
33
```

stm32f10x.h

```
0236
0237  typedef enum {FALSE = 0, TRUE = !FALSE} bool;
0238
```

怎么解决呢，很简单，只要搞掉一个即可，但要去掉哪个呢？我们考虑到外面的 M3 库很多文件都包含了 stm32f10x.h 这个头文件，假如是修改

stm32f10x.h 的话工作量会非常大，鉴于此就只能委屈 integer.h 了，修改如下，将它注释掉：

```
31 /* Boolean type */
32 //typedef enum { FALSE = 0, TRUE } BOOL; /* add by fire */
33
```

好嘞，修改之后，我们再编译下，接着又出现如下错误：

```
diskio.h(29): error:  #20: identifier "BOOL" is undefined
compiling ff.c...
diskio.h(29): error:  #20: identifier "BOOL" is undefined
ff.c(584): error:  #20: identifier "BOOL" is undefined
ff.c(861): error:  #20: identifier "FALSE" is undefined
ff.c(915): error:  #20: identifier "FALSE" is undefined
ff.c(1008): error:  #20: identifier "TRUE" is undefined
ff.c(2254): error:  #20: identifier "FALSE" is undefined
Target not created
```

意思是说在 diskio.h、ff.c 中 BOOL 、FALSE、TRUE 没定义。刚刚才把人家注释掉了，不报错才怪。

解决方法如下：

1、将 integer.h 中有关 BOOL 的那句注释掉，注释掉也没太大关系，因为注释掉的不会怎么用到：

```
28
29 //BOOL assign_drives (int argc, char *argv[]); /* add by fire */
30
```

2、在 ff.c 文件的开头重新定义一个布尔变量，取名为 bool，与 stm32f10x.h 中的名字一样：

```
0076 //#include "stm32f10x.h"  /* add by fire */
0077 typedef enum {FALSE = 0, TRUE = !FALSE} bool; /* add by fire */
0078
```

同时在 ff.c 的第 585 行做如下修改：

```
0581 static
0582 FRESULT dir_next (  /* FR_OK:Succeeded, FR_NO
0583   DIR *dj,     /* Pointer to directory object
0584   //BOOL streach  /* FALSE: Do not streach tal
0585   bool streach    /* add by fire */
0586 )
```

现在我们再编译下，发现既没警告也没错误：

```
Build Output
Build target 'SDIO-DEMO'
compiling main.c...
compiling diskio.c...
compiling ff.c...
linking...
Program Size: Code=22822 RO-data=342 RW-data=636 ZI-data=3428
FromELF: creating hex file...
"..\Output\STM32-DEMO.axf" - 0 Error(s), 0 Warning(s).
```

到这里我们算是把文件系统移植成功了，接下来的任务就是调用文件系统的函数来操作我们的卡了。其实这里的移植是非常非常简单的，要是你学过 LINUX 的话，那里面的 UBOOT 移植，系统移植，那才叫人头疼，就光是目录里面的文件夹都几千个，更别说是找到要修改的源代码了，刚接触的话绝对叫你吐血，就连下载个交叉编译器都涉及到移植。

2.6 实验代码分析

FATFS 是独立于底层介质的应用函数库，对底层介质的操作都要交给用户去实现，其仅仅是提供了一个函数接口而已，函数为空，要用户添加代码。

这几个函数的原型如下，在 diskio.c 中定义：

```
1.  /* Inidialize a Drive */
2.  DSTATUS disk_initialize (
3.      BYTE drv        /* Physical drive nmuber (0..) */
4.  )
```

```
1.  /* Return Disk Status  */
2.  DSTATUS disk_status (
3.      BYTE drv    /* Physical drive nmuber (0..) */
4.  )
```

```
1.  /* Read Sector(s) */
2.  DRESULT disk_read (
```

```
    BYTE drv,    /* Physical drive nmuber (0..) */
    BYTE *buff, /* Data buffer to store read data */
    DWORD sector,    /* Sector address (LBA) */
    BYTE count  /* Number of sectors to read (1..255) */
)
```

```
/* Write Sector(s) */
#if _READONLY == 0
DRESULT disk_write (
    BYTE drv,             /* Physical drive nmuber (0..) */
    const BYTE *buff,    /* Data to be written */
    DWORD sector,        /* Sector address (LBA) */
    BYTE count       /* Number of sectors to write (1..255) */
)
```

```
/* Miscellaneous Functions                                          */
DRESULT disk_ioctl (
    BYTE drv,    /* Physical drive nmuber (0..) */
    BYTE ctrl,  /* Control code */
    void *buff  /* Buffer to send/receive control data */
)
```

这些函数都是操作底层介质的函数，都需要用户自己实现，然后 FATFS 的应用函数就可以调用这些函数来操作我们的卡了。关于这些底层介质函数是如何实现的，请参考源码，这里就不贴出来了。

在 diskio.c 的最后我们还得提供了获取时间的函数，因为 ff.c 中调用了这个函数，而 FATFS 库又没有给出这个函数的原型，所以需要用户实现，不然会编译出错，函数体为空即可（也可以为它加载 STM32 的 RTC 驱动）：

```
/* 得到文件Calendar格式的建立日期,是DWORD get_fattime (void) 逆变换 */
/*-----------------------------------------------------------------------*/
/* User defined function to give a current time to fatfs module          */
/* 31-25: Year(0-127 org.1980), 24-21: Month(1-12), 20-16: Day(1-31) */
/* 15-11: Hour(0-23), 10-5: Minute(0-59), 4-0: Second(0-29 *2) */
DWORD get_fattime (void)
{
  return 0;
}
```

实现好底层介质的操作函数之后，我们就可以回到应用层了，下面我们从 main 函数开始看起。有关系统初始化和串口初始化这部分请参考前面的教程，这里不再详述。

首先我们调用函数 `disk_initialize( 0 );` 将我们的底层硬件初始化好，这一步非常重要，如果不成功的话，接下来什么都干不了。

`f_open( &fsrc , "0:/Demo.TXT" , FA_CREATE_NEW | FA_WRITE);` 将在刚刚开辟的工作区的盘符 0 下打开一个名为 Demo.TXT 的文件，以只写的方式打开，如果文件不存在的话则创建这个文件。并将 Demo.TXT 这个文件关联到 fsrc 这个结构指针，以后我们操作文件就是通过这个结构指针来完成的。

`f_write(&fsrc, textFileBuffer, sizeof(textFileBuffer), &br);` 将缓冲区的数据写到刚刚打开的 Demo.TXT 文件中。写完之后调用 `f_close(&fsrc);`。关闭文件，

`f_open(&fsrc, "0:/Demo.TXT", FA_OPEN_EXISTING | FA_READ);`以只读的方式打开刚刚的文件。

`f_read( &fsrc, buffer, sizeof(buffer), &br );` 将文件的内容读到缓冲区，然后调用 `printf("\r\n %s ", buffer);`将数据打印到电脑的超级终端。

最后调用 `f_close(&fsrc);`关闭文件。当被打开的文件操作完成之后都要调用 `f_close();`将它关闭，就像一块动态分配的内存在用完之后都要调用 `free()`来将它释放。

这里涉及到了 FATFS 文件系统库函数的操作，如果你学过 LINUX 系统调用的话，操作这些函数将是非常简单，没有学过的话也没太大的关系，因为 FATFS 源码目录 doc 这个文件夹中提供了每个应用函数的用法，如 `f_mount()`:

f_mount

The f_mount fucntion registers/unregisters a work area to the FatFs module.

```
FRESULT f_mount (
  BYTE   Drive,               /* Logical drive number */
  FATFS*   FileSystemObject   /* Pointer to the work area */
);
```

Parameters

Drive

Logical drive number (0-9) to register/unregister the work area.

FileSystemObject

Pointer to the work area (file system object) to be registered.

Return Values

FR_OK (0)

The function succeeded.

FR_INVALID_DRIVE

The drive number is invalid.

Description

The f_mount function registers/unregisters a work area to the FatFs modu
function. To unregister a work area, specify a NULL to the **FileSystemO**

This function only initializes the given work area and registers its addres
process is performed on first file access after f_mount or media change.

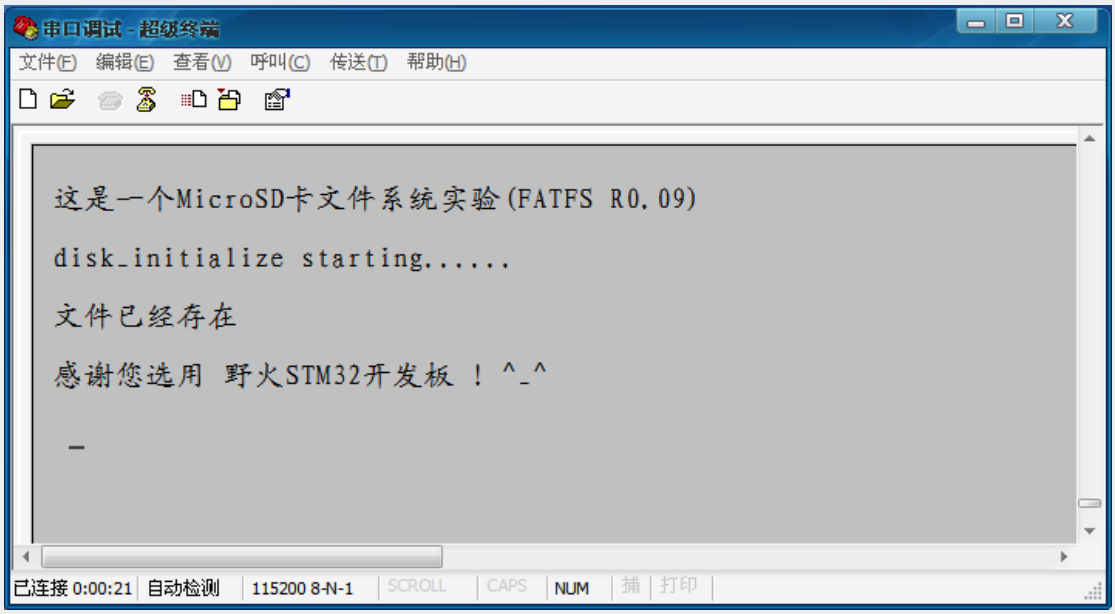
References

FATFS

Return

2.7 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，插上 MicroSD 卡(野火用的是 1G，4G 的也已经测试通过)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：


串口调试 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
这是一个MicroSD卡文件系统实验(FATFS R0.09)
disk_initialize starting.....
文件已经存在
感谢您选用 野火STM32开发板 ！ ^_^
已连接 0:00:21 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

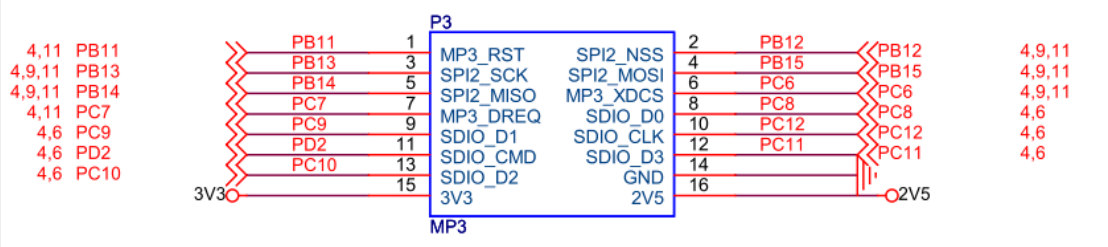
3、MP3（支持中英文、长短文件名）

3.1 实验描述及工程文件清单

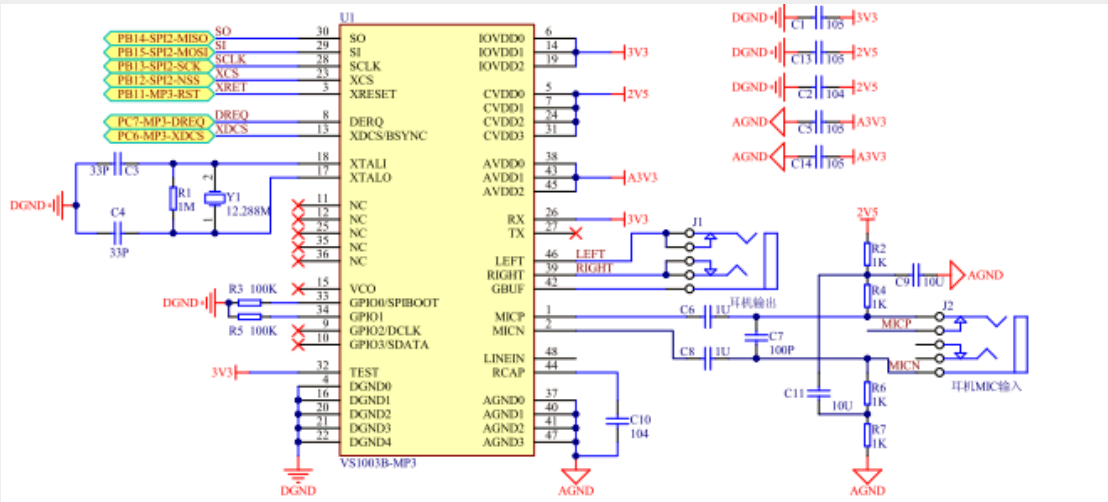
实验描述	将 MicroSD 卡(以文件系统 FATFS 访问)里面的 mp3 文件通过 VS1003B 解码，然后将解码后的数据送到功放 TDA1308 后通过耳机播放出来。注意：野火 M3-V1 的 MP3 模块是加了功放，野火 M3-V3 里面去掉了功放 TDA1308，因为从 VS1003 出来的模拟信号足够驱动耳机。 （这个文档是更新版本的，配套的例程已经可以支持长中文文件名、4G 的 sd 卡，可以播放 mp3，wma，mid 和部分的 wav 格式的音频文件）
硬件连接	PB13-SPI2_SCK ：VS1003B-SCLK PB14-SPI2_MISO：VS1003B-SO PB15-SPI2_MOSI：VS1003B-SI PB12-SPI2_NSS ：VS1003B-XCS PB11 ：VS1003B-XRET PC6 ：VS1003B-XDCS PC7 ：VS1003B-DREQ
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/stm32f10x_sdio.c FWlib/stm32f10x_dma.c

	FWlib/stm32f10x_spi.c FWlib/misc.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/sdio_sdcard.c USER/ff.c USER/usart1.c USER/mp3play.c USER/vs1003.c USER/SysTick.c
文件系统文件	ff9/diskio.c ff9/ff.c ff9/cc936.c

野火 STM32 开发板中 MP3 硬件接口图



MP3 模块原理图（野火 M3-V3 没有板载 MP3，需要另外选购）



解码部分采用 VS1003-MP3/WMA 音频解码器，然后将解码后的数据送

TDA1308 放大后由音频接口外播出来。注意：野火 M3-V1 的 MP3 模块是加了功放，野火 M3-V3 里面去掉了功放 TDA1308，因为从 VS1003 出来的模拟信号足够驱动耳机。

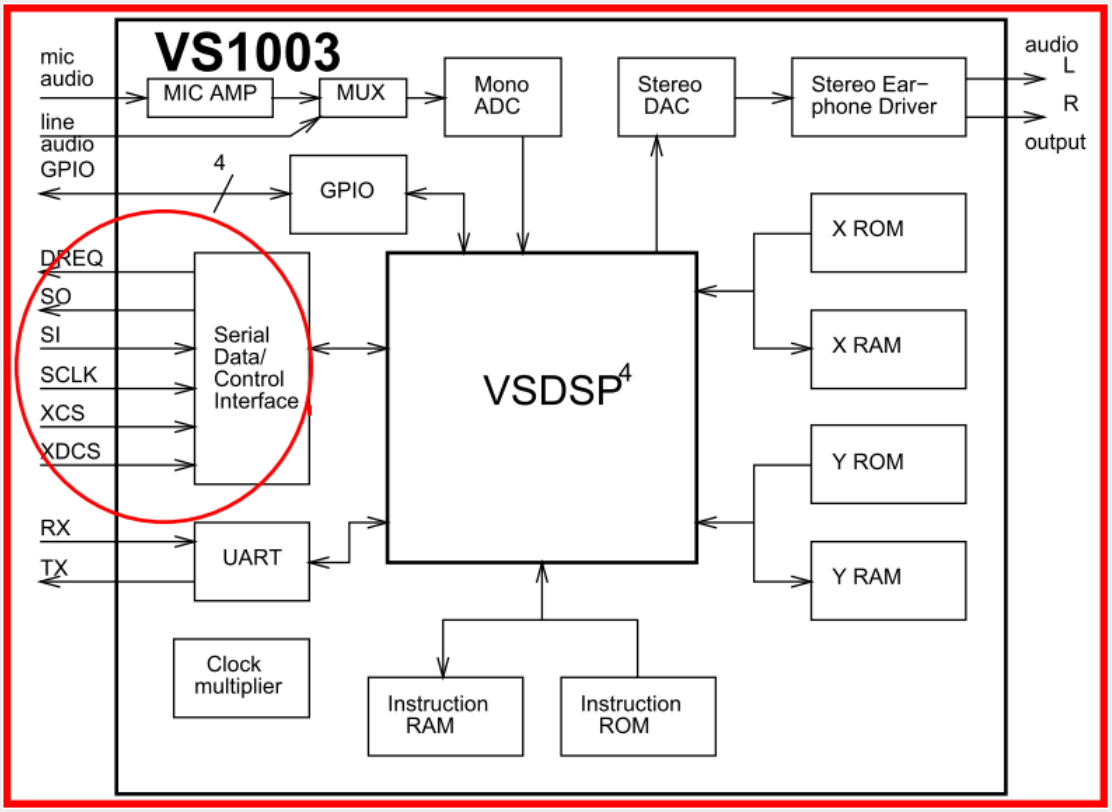
3.2 VS1003 & TDA1308 简介

3.2.1 VS1003

VS1003 是一个单片 MP3/WMA/MIDI 音频解码器和 ADPCM 编码器。它包含一个高性能，自主产权的低功耗 DSP 处理器核 VS_DSP 4 ,工作数据存储器，为用户应用提供 5KB 的指令 RAM 和 0.5KB 的数据 RAM。串行的控制和数据接口，4 个常规用途的 I/O 口，一个 UART，也有一个高品质可变采样率的 ADC 和立体声 DAC，还有一个耳机放大器和地线缓冲器。

VS1003 通过一个串行接口来接收输入的比特流，它可以作为一个系统的从机。输入的比特流被解码，然后通过一个数字音量控制器到达一个 18 位过采样多位 ε - Δ DAC。通过串行总线控制解码器。除了基本的解码，在用户 RAM 中它还可以做其他特殊应用，例如 DSP 音效处理。

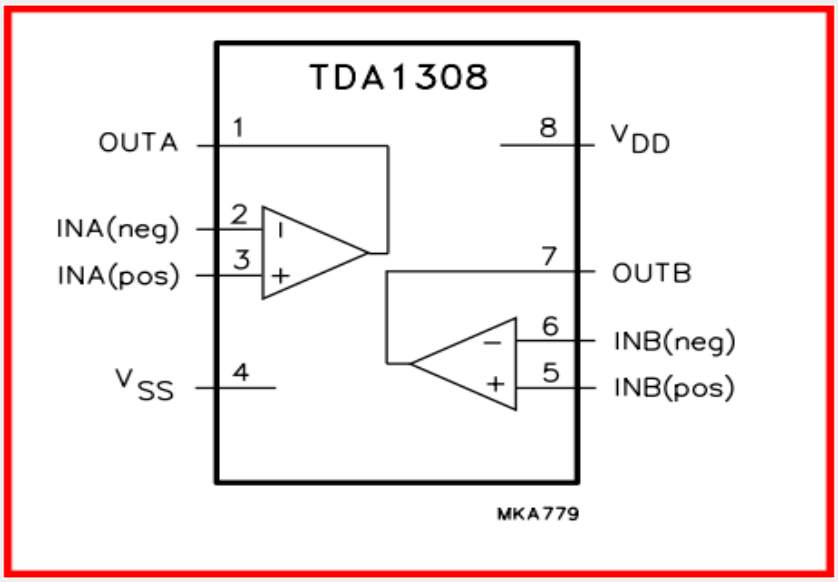
VS1003 原理框图：



本实验中我们只用了红色圆圈中的那几个数据口，这些数据口是串行模式的，我们用到了开发板中的 SPI2 来控制。其中数据经 SI 接口进去，经解码后由 L、R 这两个左右声道引脚出来，因为 VS1003 内部集成了一个 DA，所以出来的数据是模拟的，可直接驱动耳机，一般不需要另外加耳机功放。

3.2.2 TDA1308

TDA1308 是一款双通道的立体耳机驱动器，是一款专门用于耳机驱动的功放。其原理框图如右：



有关 VS1003B 和 TDA1308 的详细应用，大家可参考官方的 datasheet，野火就不在这里罗嗦。野火只是在这里介绍 TDA1308 下，让大家知道有这回事。野火 M3-V3 里面的 MP3 模块中采用的是 VS1003 的官方应用电路，没有加耳机功放，而是直接驱动耳机。

3.3 实验讲解

本实验是在 **《2、FatFS(Rev-R0.09)》**这个实验基础上进行的。 没做过这个实验的话可参考前面的教程，否则有些代码会让您犯糊涂。

首先需要将需要用到的库文件添加进来，有关库的配置可参考前面的教程，这里不再详述。在配置好库的环境之后我们从 main 函数开始分析：

```
1.  int main(void)
2.  {
3.      SysTick_Init();          /* 配置 SysTick 为 10us 中断一次 */
4.      USART1_Config();         /* 配置串口 1 115200 8-N-1 */
5.
6.  /* Interrupt Config,配置 sdio 的中断优先级,  */
7.    NVIC_Configuration();
8.
9.      printf(" \r\n 这是一个 MP3 测试例程 !\r\n " );
10.
11.      VS1003_SPI_Init();      /* MP3 硬件 I/O 初始化 */
```

```
12.
13.         MP3_Start();                /* MP3 就绪，准备播放，在 vs1003.c 实
    现 */
14.
15.          MP3_Play();                /* 播放 SD 卡(FATFS)里面的音频文
    件 */
16.
17.   /* Infinite loop */
18.   while (1)
19.   {
20.   }
21. }
```

这里没有调用库函数 SystemInit();是因为在 3.5 的固件库中，在 **3.5 版本**的库中 SystemInit()函数在启动文件 startup_stm32f10x_hd.d 中已用汇编语句调用了，设置的时钟为默认的 72M。所以在 main 函数就不需要再调用啦，当然，再调用一次也是没问题的。

如果你使用的是其它版本的库，在所有工作之前首先要做的就是先设置系统时钟，这可千万别忘了。在 ST3.0.0 版本之后的库中，这部分工作都放在了启动文件中了，由汇编实现，只要用户代码一进入 main 函数就表示已经初始化好系统时钟了，完全不用用户考虑，用户不知道这点的话还以为不需要初始化系统时钟呢。至于 ST3.0.0 和之后高版本的库有什么区别，我想说的是没什么大的区别，代码的目录结构基本没有改变，只是在代码的功能增多了，支持更完善的外设。

SysTick 为 10us 中断一次用于 SysTick 为 10us 中断一次，用于后面的延时函数。

USART1_Config(); 配置串口 1 波特率为 115200，8 个数据位，1 个停止位，无硬件流控制。

NVIC_Configuration(); 用于配置 MicroSD 卡的中断优先级。

VS1003_SPI_Init();用于初始化 MP3 解码芯片 VS1003B 需要用到的 I/O 口，包括数据口(SPI2)和控制 I/O。VS1003_SPI_Init();由用户在 vs1003.c 中实现：

```
1.  /*
2.   * 函数名: VS1003_SPI_Init
3.   * 描述   : VS1003 所用 I/O 初始化
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 外部调用
7.   */
8.  void VS1003_SPI_Init(void)
9.  {
10.   SPI InitTypeDef  SPI InitStructure;
11.   GPIO InitTypeDef GPIO InitStructure;
12.
13.   /* 使能 VS1003B 所用 I/O 的时钟 */
```

```
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |RCC_APB2Periph_GPIOB | RCC_APB2Periph GPIOC , ENABLE);
    /* 使能 SPI2 时钟 */
  RCC APB1PeriphClockCmd(RCC APB1Periph SPI2 ,ENABLE);

  /* 配置 SPI2 引脚: PB13-SCK, PB14-MISO 和 PB15-MOSI */
  GPIO_InitStructure.GPIO_Pin = 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_Init(GPIOB, &GPIO_InitStructure);

  /* PB12-XCS(片选) */
  GPIO InitStructure.GPIO Pin = GPIO Pin 12;
  GPIO InitStructure.GPIO Speed = GPIO Speed 50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* PB11-XRST(复位) */
  GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_11;
  GPIO Init(GPIOB, &GPIO InitStructure);


  /* PC6-XDCS(数据命令选择) */
  GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_6;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  /* PC7-DREQ(数据中断) */
  GPIO InitStructure.GPIO Mode = GPIO Mode IPD;
  GPIO InitStructure.GPIO Pin =  GPIO Pin 7;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  /* SPI2 configuration */
  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_Low;
  SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
  SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
  SPI InitStructure.SPI BaudRatePrescaler = SPI BaudRatePrescaler 32;
  SPI InitStructure.SPI FirstBit = SPI FirstBit MSB;
  SPI InitStructure.SPI CRCPolynomial = 7;
  SPI_Init(SPI2, &SPI_InitStructure);

  /* Enable SPI2  */
  SPI Cmd(SPI2, ENABLE);
}
```

假如我们要将数据口换成 SPI1 或者改变其他控制 I/O，只需改变这个函数即可，移植性非常强。关于 STM32 的 SPI 接口详细使用教程，请参照前面的 FLASH 或 E^2PROM 的文档。

`MP3_Start();` 使 MP3 进入就绪模式(standby)，随时播放音乐。`MP3_Start();`在 vs1003.c 中实现：

```
  * 函数名: MP3_Start
  * 描述   : 使 MP3 进入就绪模式，随时准备播放音乐。
  * 输入   : 无
  * 输出   : 无
  * 调用   : 外部调用
  */
void MP3_Start(void)
{
    u8 BassEnhanceValue = 0x00;       // 低音值先初始化为 0
    u8 TrebleEnhanceValue = 0x00;     // 高音值先初始化为 0
    TRST_SET(0);
    Delay_us( 1000 );                 // 1000*10us = 10ms
```

```
13.
14.     VS1003_WriteByte(0xff);           // 发送一个字节的无效数据，启动 SPI 传输
15.     TXDCS_SET(1);
16.     TCS_SET(1);
17.     TRST_SET(1);
18.     Delay_us( 1000 );
19.
20.     Mp3WriteRegister( SPI_MODE,0x08,0x00);      // 进入 VS1003 的播放模式
21.     Mp3WriteRegister(3, 0x98, 0x00);            // 设置 vs1003 的时钟,3 倍频
22.     Mp3WriteRegister(5, 0xBB, 0x81);            // 采样率 48k，立体声
23.     // 设置重低音
24.     Mp3WriteRegister(SPI_BASS, TrebleEnhanceValue, BassEnhanceValue);
25.     Mp3WriteRegister(0x0b,0x00,0x00);    // VS1003 音量
26.     Delay_us( 1000 );
27.
28.     while( DREQ == 0 );                 // 等待 DREQ 为高   表示能够接受音乐数据输入
29. }
```

函数中涉及到的宏定义都在 vs1003.h 这个头文件中实现。关于函数中为什么要这样操作寄存器，或者为什么要按照这个顺序来操作寄存器，请大家查阅 vs1003 的 pdf，里面讲得很详细，有 e 文跟中文资料。

`MP3_Play();` 这个函数逐个扫描我们卡里面的音频文件，把根目录下的所有音频文件播放一次，若音频文件放在其它目录，可以通过修改代码中的文件路径来实现。以下是 `MP3_Play();` 在 vs1003.c 中实现：

```
1.  /*
2.   * 函数名: MP3_Play
3.   * 描述   : 读取 SD 卡里面的音频文件，并通过耳机播放出来
4.   *          支持的格式: mp3,mid,wma, 部分的 wav
5.   * 输入   : 无
6.   * 输出   : 无
7.   * 说明   : 已添加支持长中文文件名
8.   */
9.  void MP3_Play(void)
10. {
11.
12.   FATFS fs;                   // Work area (file system object) for log
    ical drive
13.   FRESULT res;
14.   UINT br;                     /*读取出的字节数，用于判断是否到达文件尾*/
15.   FIL fsrc;                   // file objects
16.   FILINFO finfo;              /*文件信息*/
17.   DIR dirs;
18.   uint16_t count = 0;
19.
20.    char lfn[70];       /*为支持长文件的数组，[]最大支持 255*/
21.     char j = 0;
22.   char path[100] = {""}; /* MicroSD 卡根目录 */
23.   char *result1, *result2, *result3, *result4;
24.
25.    BYTE buffer[512];           /* 存放读取出的文件数据 */
26.
27.   finfo.lfname = lfn;          /*为长文件名分配空间*/
28.  finfo.lfsize = sizeof(lfn);    /*空间大小*/
29.
30.   f_mount(0, &fs);                                        /* 挂载文件系统到 0
    区 */
31.
```

```
32.     if (f_opendir(&dirs,path) == FR_OK)                 /* 打开根目
    录 */
33.   {
34.     while (f_readdir(&dirs, &finfo) == FR_OK)         /* 依次读取文件
    名 */
35.     {
36.
37.       if ( finfo.fattrib & AM_ARC )          /* 判断是否为存档型文档 */
38.       {
39.             if(finfo.lfname[0] == NULL && finfo.fname !=NULL)  /*当长文
    件名称为空，短文件名非空时转换*/
40.                           finfo.lfname =finfo.fname;
41.
42.
43.             if( !finfo.lfname[0] )    /* 文件名为空即到达了目录的末尾，退
    出 */
44.                       break;
45.
46.                 printf(  " \r\n 文件名为: %s \r\n",finfo.lfname );
47.
48.                 result1 = strstr( finfo.lfname, ".mp3" );    /* 判断是否
    为音频文件 */
49.                 result2 = strstr( finfo.lfname, ".mid" );
50.                 result3 = strstr( finfo.lfname, ".wav" );
51.                 result4 = strstr( finfo.lfname, ".wma" );

52.
53.                 if ( result1!=NULL || result2!=NULL || result3!=NULL |
    | result4!=NULL )
54.                 {
55.
56.                    if(result1 != NULL)/*若是 mp3 文件则读取 mp3 的信息*/
57.                     {
58.                         res = f_open( &fsrc, finfo.lfname, FA_OPEN_EXI
    STING | FA_READ ); /* 以只读方式打开 */
59.
60.
61.                             /* 获取歌曲信息
    (ID3V1 tag / ID3V2 tag) */
62.                             if ( Read_ID3V1(&fsrc, &id3v1) == TRUE )
63.                                 {// ID3V1 tag
64.                                     printf( "\r\n 曲
    目    : %s \r\n", id3v1.title );
65.                                     printf( "\r\n 艺术
    家  : %s \r\n", id3v1.artist );

66.                                     printf( "\r\n 专
    辑    : %s \r\n", id3v1.album );
67.                                 }
68.                             else
69.                                 {// 有些 MP3 文件没有 ID3V1 tag,只有
    ID3V2 tag
70.                                     res = f_lseek(&fsrc, 0);
71.                                     Read_ID3V2(&fsrc, &id3v2);
72.
73.                                     printf( "\r\n 曲
    目    : %s \r\n", id3v2.title );
74.                                     printf( "\r\n 艺术
    家  : %s \r\n", id3v2.artist );
75.                                 }
76.                         }
77.                         /* 使文件指针 fsrc 重新指向文件头，因为在调用
    Read_ID3V1/Read_ID3V2 时，
```

```
78.                             fsrc 的位置改变了 */
79.                         res = f_open( &fsrc, finfo.lfname, FA_OPEN_EXI
    STING | FA_READ );
80.                         res = f_lseek(&fsrc, 0);
81.
82.
83.                     br = 1;                                 /* br 为全局变
    量 */
84.                     TXDCS_SET( 0 );             /* 选择 VS1003 的数据接
    口 */
85. /* ------------------- 一曲开始 ---------------------*/
86.                     printf( " \r\n 开始播放 \r\n" );
87.                     for (;;)
88.                     {
89.                         res = f_read( &fsrc, buffer, sizeof(buffer), &
    br );
90.                         if ( res == 0 )
91.                         {
92.                                 count = 0;
      /* 512 字节完重新计数 */
93.
94.                                 Delay_us( 1000 );           /* 10ms 延
    时 */
95.                                 while ( count < 512)            /* SD 卡
    读取一个 sector，一个 sector 为 512 字节 */
96.                                 {
97.                             if ( DREQ != 0 )            /* 等待 DREQ 为高，请求
    数据输入 */
98.                                     {
99.                                         for (j=0; j<32; j++ ) /* VS100
    3 的 FIFO 只有 32 个字节的缓冲 */
100.                                            {
101.                                                VS1003_WriteByte( b
    uffer[count] );
102.                                                count++;
103.                                            }
104.                                    }
105.                                }
106.                    }
107.                            if (res || br == 0) break;    /* 出错或者到
    了 MP3 文件尾 */
108.                }
109.                            printf( " \r\n 播放结束 \r\n" );
110.      /* ------------------- 一曲结束 ---------------------*/
111.                        count = 0;
112.                          /* 根据 VS1003 的要求，在一曲结束后需发送 2048 个
    0 来确保下一首的正常播放 */
113.                        while ( count < 2048 )
114.                        {
115.                    if ( DREQ != 0 )
116.                                {
117.                                    for ( j=0; j<32; j++ )
118.                                {
119.                                        VS1003_WriteByte( 0 );
120.                                        count++;
121.                                }
122.                                }
123.                    }
124.                            count = 0;
125.                            TXDCS_SET( 1 );    /* 关闭 VS1003 数据端
    口 */
```

```
126.                         f_close(&fsrc);    /* 关闭打开的文件 */
127.                       }
128.             }
129.           } /* while (f_readdir(&dirs, &finfo) == FR_OK) */
130.         } /* if (f_opendir(&dirs, path) == FR_OK)  */
131.       } /* end of MP3_Play */
```

由于代码比较长，在格式编排上不是很好，野火建议大家还是配合源代码一起阅读^_^。

现在我们来大概分析下 `MP3_Play();` 这个函数，这里边涉及到一些文件系统操作的函数，关于这部分函数的操作大家可参考前面的教程或者阅读 FATFS 的官方文档，其实我的教程也不完全正确，阅读官方的文档才是最可靠的。

首先说一下为支持中文长文件名的文件系统配置。

要在 ffconf.h 文件中的 Namespace configuration 宏配置中设定如下：

```
1.  /*---------------------------------------------------------------------------
    -------/
2.  / Locale and Namespace Configurations
3.  /---------------------------------------------------------------------------
    -------*/
4.
5.  #define _CODE_PAGE  936
6.  /* The _CODE_PAGE specifies the OEM code page to be used on the target
     system.
7.  /  Incorrect setting of the code page can cause a file open failure.
8.  /
9.  /   932  - Japanese Shift-JIS (DBCS, OEM, Windows)
10. /   936  - Simplified Chinese GBK (DBCS, OEM, Windows)
11. /   949  - Korean (DBCS, OEM, Windows)
12. /   950  - Traditional Chinese Big5 (DBCS, OEM, Windows)
13. /   1250 - Central Europe (Windows)
14. /   1251 - Cyrillic (Windows)
15. /   1252 - Latin 1 (Windows)
16. /   1253 - Greek (Windows)
17. /   1254 - Turkish (Windows)
18. /   1255 - Hebrew (Windows)
19. /   1256 - Arabic (Windows)
20. /   1257 - Baltic (Windows)
21. /   1258 - Vietnam (OEM, Windows)
22. /   437  - U.S. (OEM)
23. /   720  - Arabic (OEM)
24. /   737  - Greek (OEM)
25. /   775  - Baltic (OEM)
26. /   850  - Multilingual Latin 1 (OEM)
27. /   858  - Multilingual Latin 1 + Euro (OEM)
28. /   852  - Latin 2 (OEM)
29. /   855  - Cyrillic (OEM)
30. /   866  - Russian (OEM)
31. /   857  - Turkish (OEM)
32. /   862  - Hebrew (OEM)
33. /   874  - Thai (OEM, Windows)
34. /   1    - ASCII only (Valid for non LFN cfg.)
35. */
36.
37.
38. #define _USE_LFN    2       /* 0 to 3 */
```

```
#define _MAX_LFN    255     /* Maximum LFN length to handle (12 to 255) */
/* The _USE_LFN option switches the LFN support.
/
/   0: Disable LFN feature. _MAX_LFN and _LFN_UNICODE have no effect.
/   1: Enable LFN with static working buffer on the BSS. Always NOT reentrant.
/   2: Enable LFN with dynamic working buffer on the STACK.
/   3: Enable LFN with dynamic working buffer on the HEAP.
/
/  The LFN working buffer occupies (_MAX_LFN + 1) * 2 bytes. To enable LFN,
/  Unicode handling functions ff_convert() and ff_wtoupper() must be added
/  to the project. When enable to use heap, memory control functions
/  ff_memalloc() and ff_memfree() must be added to the project. */


#define _LFN_UNICODE    0   /* 0:ANSI/OEM or 1:Unicode */
/* To switch the character code set on FatFs API to Unicode,
/  enable LFN feature and set _LFN_UNICODE to 1. */


#define _FS_RPATH       0   /* 0 to 2 */
/* The _FS_RPATH option configures relative path feature.
/
/   0: Disable relative path feature and remove related functions.
/   1: Enable relative path. f_chdrive() and f_chdir() are available.
/   2: f_getcwd() is available in addition to 1.
/
/  Note that output of the f_readdir fnction is affected by this option. */
```

修改的第一个宏配置是_CODE_PAGE 改成简体中文的 936.

Code page 是什么？我们知道 ASCII 码的前 7 位定义的是我们常用的标准字符集，于是 128 位以下的用处达成了共识，而 ASCII 码中的第 8 位没有被使用，对于 128 位以上的可能有不同的解释，这些不同的解释就叫做 code_page，我们使用 936 这个宏就是调用了简体中文的 code_page。所以要支持中文，还要添加 fatfs 源文件中 option 目录下的 cc936.c 文件到工程中。

接下来还要修改_USE_LFN 和 MAX_LFN 的宏，这两个是长文件名支持的配置。

MAX_LFN 定义了最大文件名长度，单位为 Byte。

_USE_LFN >= 1 则开启长文件支持。

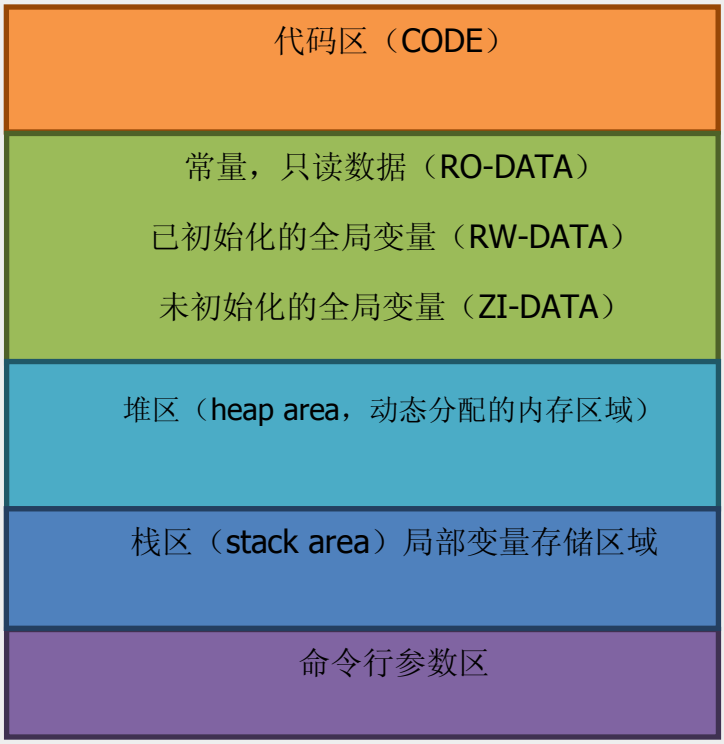
=1 表示长文件名的存储在 静态存储区。

=2 表示长文件名的存储在 栈区。

=3 表示长文件名的存储在 堆区。

这里涉及到变量的存储分布问题。

sram 内存变量分布图：



存储在全局变量的内存空间是不会被回收的，栈区是用来存放局部变量的，在子函数调用运行完成之后会释放内存，而且少用全局变量会让代码的移植性更好。所以这里的长文件名变量我们把它设置为 2，把它放在栈区。

另外，因为我们的 `mp3_play()` 函数中定义了很多局部变量，占用的栈空间很大，所以我们要修改启动文件 startup_stm32f10x_hd.s 中的栈空间大小：

```
1.  Stack_Size      EQU     0x00000f00          ;Stack_Size,标号。EQU 定义
2.                  AREA    STACK, NOINIT, READWRITE, ALIGN=3       ;定义名称
    为 stack 的栈，noinitStack_Mem      SPACE   Stack_Size          ;定义名称为
    Stack_Mem 大小为 stack_size 大小
3.  __initial_sp
```

在这个文件中把原来的

Stack_size EQU 0x00000400

改成了

Stack_size EQU 0x0000f00 。

有时我们调试程序的时候会发现代码莫名奇妙地卡在 harddefault 的硬中断里，这时可以检查一下是不是在启动文件中把栈大小设置得太保守了，可以根据实际需要把这个设置得大一点。

文件系统中的文件信息结构体：

```
/* File status structure (FILINFO) */

typedef struct {
    DWORD    fsize;          /* File size */
    WORD     fdate;          /* Last modified date */
    WORD     ftime;          /* Last modified time */
    BYTE     fattrib;        /* Attribute */
    TCHAR    fname[13];      /* Short file name (8.3 format) */
#if _USE_LFN
    TCHAR*   lfname;         /* Pointer to the LFN buffer */
    UINT     lfsize;         /* Size of LFN buffer in TCHAR */
#endif
} FILINFO;
```

关于长文件名（包管中英文）的支持，最后还要注意一点，在使用文件名信息时，不要再使用 FILINFO->fname（短文件名数组） 。而应该使用 FILINFO->lfname （长文件名指针）。而且长文件名在结构体中定义的是一个指针，在使用前我们要为这个指针分配内存空间，注意不要使用野指针。具体的使用方法可以参照 `mp3_play()` 函数中开头的变量定义和赋初值部分。

还要注意一下如果读取的文件名长度不超过 FILINFO->fname（短文件名）的空间时，文件名的信息只会保存在短文件名数组中，而 FILINFO->lfname （长文件名指针）的值将会是空的，所以我在代码中加了一个判断语句才可以进行正常的使用。

函数 `f_mount(0, &fs);` 为我们在文件系统中注册一个工作区，并初始化盘符的名为 0。这个函数还调用了底层的 `disk_initialize ()`，进行 sdio 的初始化，所以在文件操作之前必须调用这个函数。不建议在 main 函数直接调用 `disk_initialize ()` 来对 sdio 进行初始化，要尽量使用封装好的脱离硬件层的函数，这样会令代码移植性更好呀。

函数 `f_opendir(&dirs, path)` 用于打开卡的根目录，并将这个根目录关联到 dirs 这个结构指针，然后我们就可以通过这个结构指针来操作这个目录了，其实这个结构指针就类似 LINUX 下系统编程中的文件描述符，不论是操作还是目录都得通过文件描述符才能操作。

`f_readdir(&dirs, &finfo)` 函数通过刚刚的 dirs 结构指针来读取目录里面的信息，并将目录的信息储存在 `finfo` 这个结构体变量中。这个结构体中包括了文件名，文件大小，文件类型，修改时间等信息。

紧接着判断文件的属性，如果是存档型文件的话就将文件名打印出来，然后比较文件的后缀名，查看是否为音频文件，支持的音频格式有 mp3、mid、wav、wma。

如果是音频文件的话则调用 `f_open( &fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ );`打开这个音频文件。

如果是 mp3 类型的文件件，我们还可以调用 `Read_ID3V1()`和 `Read_ID3V2()`来读取 mp3 的文件信息，这些文件信息是属于 mp3 文件的内部数据，可以参照《mp3 文件的存储格式》这个文档来理解这两个函数，实质就是把文件记录的数据，按格式把相应的信息整合到结构体里便于使用而已。

我们把读取到的音频数据直接通过 SPI 接口送入到 vs1003 就可以进行各种音频数据的解码了。

`TXDCS_SET( 0 );` 用于选择 vs1003 的数据端口，准备往 vs1003 中输入数据。其中 `TXDCS_SET( 0 );`是在 vs1003.h 中实现的一个宏：

```
1.  #define XDCS    (1<<6)   // PC6-XDCS
2.
3.  #define TXDCS_SET(x)   GPIOC->ODR=(GPIOC->ODR&~XDCS)|(x ? XDCS:0)
```

紧接着进入一个大循环中播放我们的 mp3 文件。

函数 `f_read( &fsrc, buffer, sizeof(buffer), &br );`从文件中读取 512 个字节的数据到缓冲区中，至于为什么是 512 个字节，这是因为卡的一个 sector 是 512 个字节，一次只能读取一个 sector，实际上也可以一次读取 n 个 sector，但在这里没必要。

函数 `VS1003_WriteByte( buffer[count] );`将缓冲区中的数据写入 vs1003 的数据缓冲区。注意，这里一次只能写入 32 个字节，这是因为 vs1003 的 FIFO 的大小为 32 个字节，写多了无效。

当文件出错或者一曲播放完毕时就跳出 for 循环，并打印出“播放结束”的调试信息。

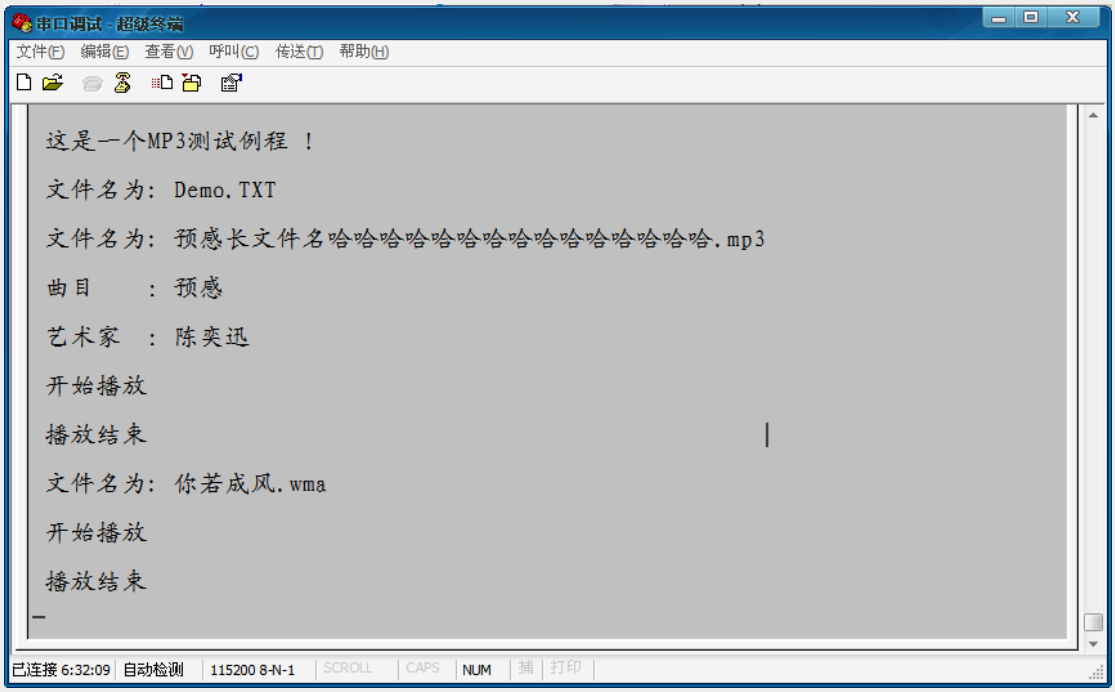
根据 VS1003 的要求，在一曲结束后需发送 2048 个 0 来确保下一首的正常播放。

一曲播放完毕我们关闭 vs1003 的数据端，关闭打开的文件，等待下一曲的播放，直到目录下的音频文件播放完为止。

这里面涉及到了 vs1003 操作的一些特性，需大家参考 vs1003 的 datasheet 来帮助理解。

3.4 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，插上 MicroSD 卡(野火用的是 1G，也可支持 2G、4G，8G)，在卡的根目录下要有 mp3 文件，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：



野火的卡的根目录下放了 1 个 mp3 文件，1 个 wma 文件。可以看到，这个代码支持了超长的中文文件名；也支持了 wma 的格式，根据 vs1003 的

datasheet 说明，还可以支持 mid 和部分的 wav 音频，大家可以尝试一下音量可通过耳机来调，前提是你的耳机要能调节音量才行。

4、液晶触摸画板

4.1 实验简介

本实验向大家介绍如何使用 STM32 的 FSMC 接口驱动 LCD 屏，及使用触摸屏控制器检测触点坐标。

4.2 LCD 控制器简介

LCD，即液晶显示器，因为其功耗低、体积小，承载的信息量大，因而被广泛用于信息输出、与用户进行交互，目前仍是各种电子显示设备的主流。

因为 STM32 内部没有集成专用的液晶屏和触摸屏的控制接口，所以在显示面板中应自带含有这些驱动芯片的驱动电路(液晶屏和触摸屏的驱动电路是独立的)，STM32 芯片通过驱动芯片来控制液晶屏和触摸屏。以野火 3.2 寸液晶屏(240*320)为例，它使用 *ILI9341* 芯片控制液晶屏，通过 *TSC2046* 芯片控制触摸屏。

4.2.1 ILI9341 控制器结构

液晶屏的控制芯片内部结构非常复杂，见**错误！未找到引用源。**。最主要的是位于中间 GRAM(Graphics RAM)，可以理解为*显存*。GRAM 中每个存储单元都对应着液晶面板的一个像素点。它右侧的各种模块共同作用把 GRAM 存储单元的数据转化成液晶面板的控制信号，使像素点呈现特定的颜色，而像素点组合起来则成为一幅完整的图像。

框图的左上角为 ILI9341 的主要控制信号线和配置引脚，根据其不同状态设置可以使芯片工作在不同的模式，如每个像素点的位数是 6、16 还是 18 位；使用 SPI 接口还是 8080 接口与 MCU 进行通讯；使用 8080 接口的哪种模式。MUC 通过 SPI 或 8080 接口与 ILI9341 进行通讯，从而访问它的控制寄存器(CR)、地址计数器(AC)、及 GRAM。

在 GRAM 的左侧还有一个 LED 控制器(LED Controller)。LCD 为非发光性的显示装置，它需要借助背光源才能达到显示功能，LED 控制器就是用来控制液晶屏中的 LED 背光源。

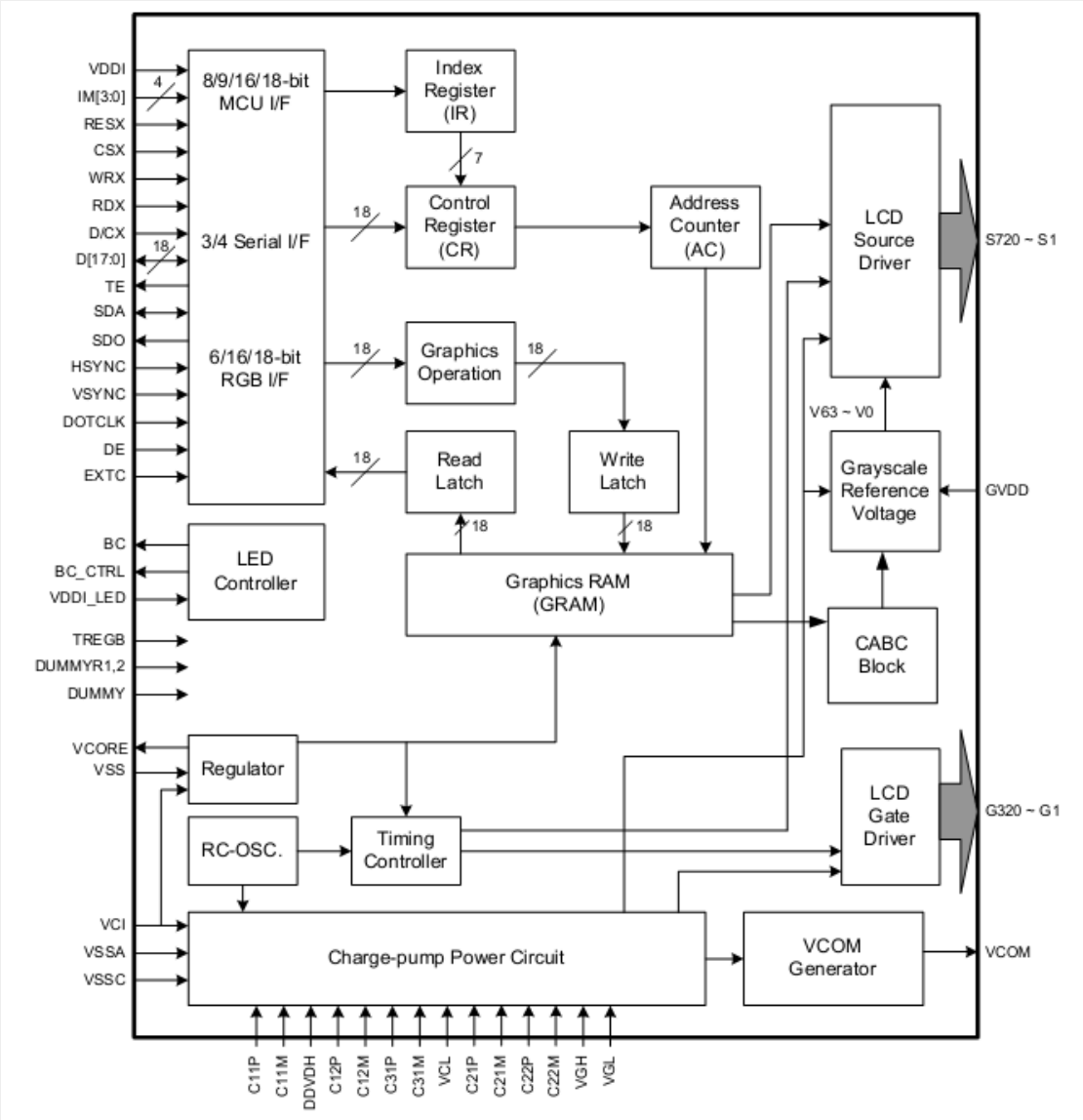


图 0-1 ILI9341 控制器内部框图

4.2.2 像素点的数据格式

图像数据的像素点由红(R)、绿(G)、蓝(B)三原色组成，三原色根据其深浅程度被分为 0~255 个级别，它们按不同比例的混合可以得出各种色彩。如 R：255，G255，B255 混合后为白色。根据描述像素点数据的长度，主要分为 8、16、24 及 32 位。如以 8 位来描述的像素点可表示 2^8=256 色，16 位描述的为

2^{16}=65536 色，称为真彩色，也称为 64K 色。实际上受人眼对颜色的识别能力的限制， 16 位色与 12 位色已经难以分辨了。

ILI9341 最高能够控制 18 位的 LCD，但为了数据传输简便，我们采用它的 16 位控制模式，以 16 位描述的像素点。按照标准格式，16 位的像素点的三原色描述的位数为 R：G：B =5：6：5，描述绿色的位数较多是因为人眼对绿色更为敏感。16 位的像素点格式见图 0-2。

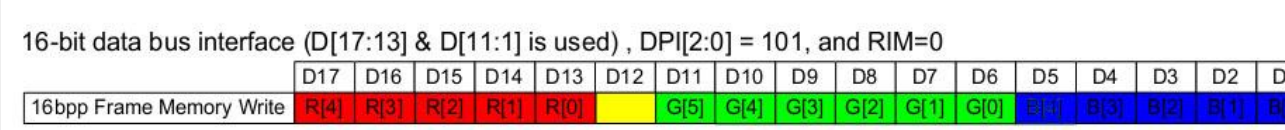


图 0-2 16 位像素点格式

图中的是默认 18 条数据线时，像素点三原色的分配状况，D1~D5 为蓝色，D6~D11 为绿色，D13~D17 为红色。这样分配有 D0 和 D12 位是无效的。若使用 16 根数据线传送像素点的数据，则 D0~D4 为蓝色，D5~D10 为绿色，D11~D15 为红色，使得刚好使用完整的 16 位。

RGB 比例为 5：6：5 是一个十分通用的颜色标准，在 GRAM 相应的地址中填入该颜色的编码，即可控制 LCD 输出该颜色的像素点。如黑色的编码为 0x0000，白色的编码为 0xffff，红色为 0xf800。

4.2.3 ILI9341 的通讯时序

目前，大多数的液晶控制器都使用 8080 或 6800 接口与 MCU 进行通讯，它们的时序十分相似，野火以 ILI9341 使用的 8080 通讯时序进行分析，实际上 ILI9341 也可以使用 SPI 接口来控制。

ILI9341 的 8080 接口有 5 条基本的控制信号线：

1. 用于片选的 CSX 信号线；
2. 用于写使能的 WRX 信号线；
3. 用于读使能的 RDX 信号线；
4. 用于区分数据和命令的 D/CX 信号线；
5. 用于复位的 RESX 信号线。

其中带 X 的表示低电平有效。除了控制信号，还有数据信号线，它的数目不定，可根据 ILI9341 框图中的 IM[3:0]来设定，这部分一般由制作液晶屏的厂家完成。为便于传输像素点数据，野火使用的液晶屏设定为 16 条数据线 D[15:0]。使用 8080 接口的写命令时序图见**错误！未找到引用源。**。

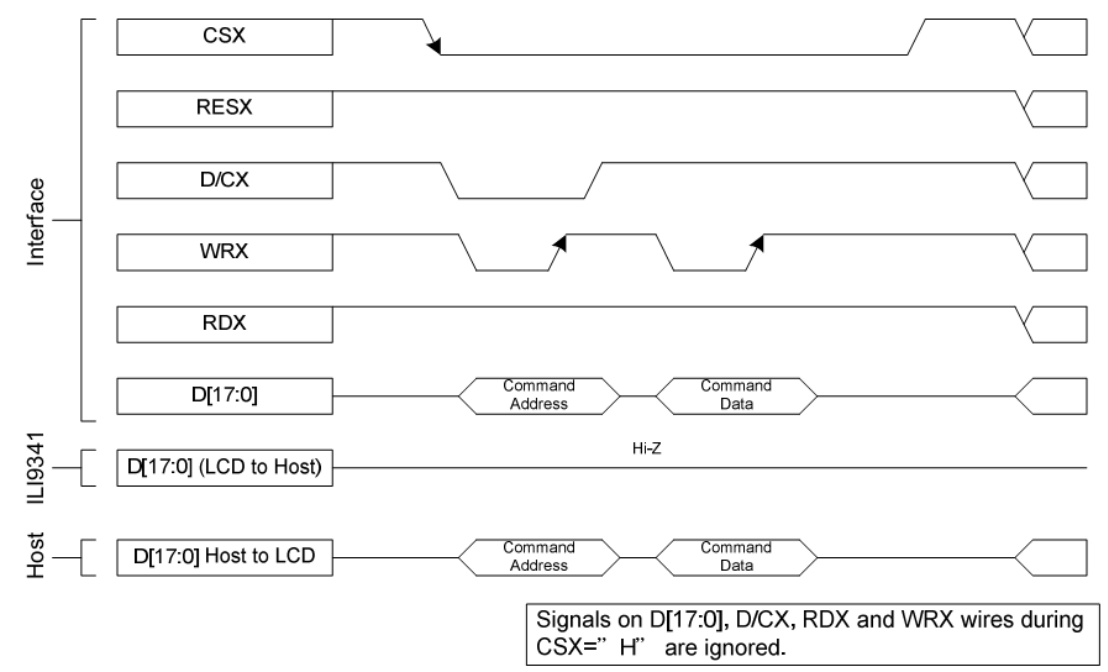


图 0-3 使用 18 条数据线的 8080 接口写命令时序

由图可知，写命令时序由 CSX 信号线拉低开始，*D/CX 信号线也置低电平*表示写入的是*命令地址*(可理解为命令编码，如软件复位命令：0x01)，以 WRX 信号线为低，RDX 信号为高表示数据传输方向为写入，同时，在数据线[17:0]输出命令地址，在第地二个传输阶段传送的为命令的参数，所以 *D/CX 要置高电平*，表示写入的是*命令数据*。

当我们需要向 GRAM 写入数据的时候，把 CSX 信号线拉低后，*把 D/CX 信号线置为高电平*，这时由 D[17:0]传输的数据则会被 ILI9341 保存至它的 GRAM 中。

4.3 用 STM32 驱动 LCD

ILI9341 的 8080 通讯接口时序可以由 STM32 使用普通 I/O 接口进行模拟，但这样效率较低，它提供了一种特别的控制方法——使用 FSMC 接口。

FSMC 简介

FSMC(flexible static memory controller)，译为静态存储控制器。可用于 STM32 芯片控制 NOR FLASH、PSRAM、和 NAND FLASH 存储芯片。其结构见图 0-4。

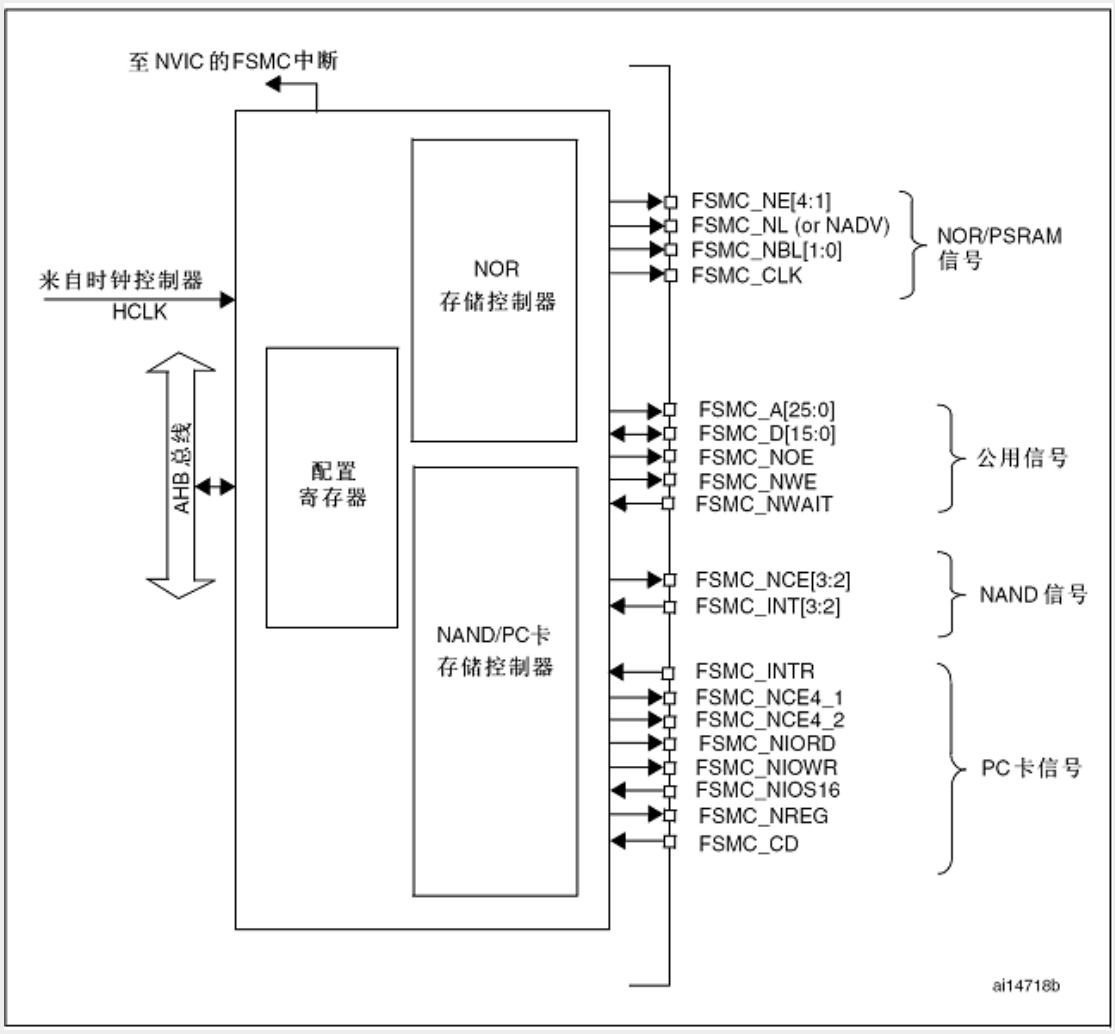


图 0-4 FSMC 结构图

我们是使用 FSMC 的 NOR\PSRAM 模式控制 LCD，所以我们重点分析框图中 NOR FLASH 控制信号线部分。控制 NOR FLASH 主要使用到如下信号线：

FSMC信号名称	信号方向	功能
CLK	输出	时钟(同步突发模式使用)
A[25:0]	输出	地址总线
D[15:0]	输入/输出	双向数据总线
NE[x]	输出	片选，x = 1...4
NOE	输出	输出使能
NWE	输出	写使能
NWAIT	输入	NOR闪存要求FSMC等待的信号

图 0-5 FSMC 控制 NOR FLASH 的信号线

根据 STM32 对寻址空间的地址映射，见前面的**错误！未找到引用源。**，*地址 0x6000 0000 ~0x9FFF FFFF 是映射到外部存储器的*，而其中的 0x6000 0000 ~0x6FFF FFFF 则是分配给 NOR FLASH、PSRAM 这类可直接寻址的器件。当 FSMC 外设被配置为正常工作，并且外部接了 NOR FLASH，这时若向 0x60000000 地址写入数据 0xffff，FSMC 会自动在各信号线上产生相应的电平信号，写入数据。该过程的时序图见图 0-6。

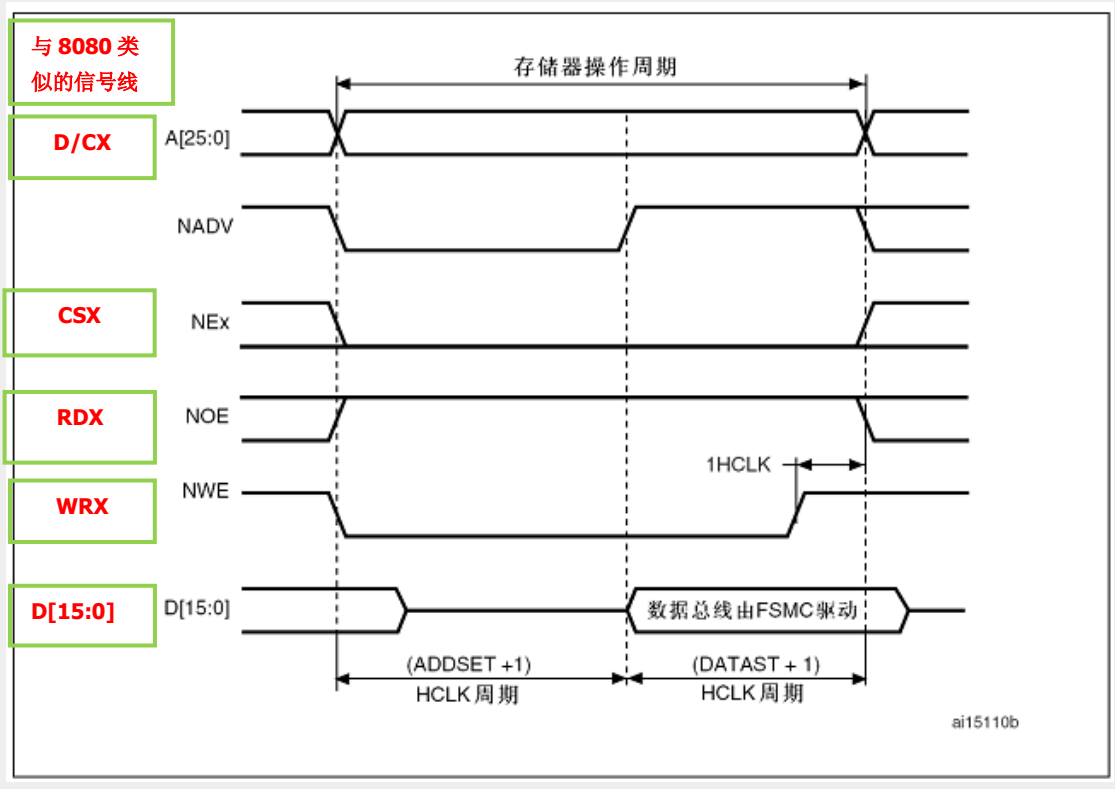


图 0-6 FSMC 写 NOR 时序图

它会控制*片选信号 NE[X]*选择相应的某块 NOR 芯片，然后使用*地址线 A[25:0]*输出 0x60000000，在 *NEW 写使能信号线*上发出写使能信号，而要写

入的数据信号 0xffff 则从*数据线 D[15:0]*输出，然后数据就被保存到 NOR FLASH 中了。

用 FSMC 模拟 8080 时序

在图 0-6 的时序图中 NADV 信号是在地址、信号线复用时作为锁存信号的，在此，我们略它。然后读者会发现，这个 FSMC 写 NOR 时序是跟 8080 接口的时序(见图 0-3)是十分相似的，对它们的信号线对比如下：

8080 信号线	功能	FSMC-NOR 信号线	功能
CSX	片选信号	NEx	片选
WRX	写使能	NWR	写使能
RDX	读使能	NOE	读使能
D[15:0]	数据信号	D[15:0]	数据信号
D\CX	数据/命令选择	A[25:0]	地址信号

前四种信号线都是完全一样的，仅在 8080 的数据\命令选择线与 FSMC 的地址信号线有区别。为了模拟出 8080 时序，我们把 FSMC 的 *A0 地址线*(也可以使用其它地址线)连接 8080 的 *D|CX*，即 A0 为*高电平*时，数据线 D[15:0]的信号会被理解 ILI9341 为*数值*，若 A0 为*低电平*时，传输的信号则会被理解为*命令*。

也就是说，当向地址为 0x6xxx xxx1、0x6xxx xxx3、0x6xxx xxx5…这些*奇数地址*写入数据时，地址线 A0(D/CX)会为*高电平*，这个数据被理解为*数值*；若向 0x6xxx xxx0 、0x6xxx xxx2、0x6xxx xxx4…这些*偶数地址*写入数据时，地址线 A0(D/CX)会为*低电平*，这个数据会被理解为*命令*。

有了这个基础，只要我们在代码中利用指针变量，向不同的地址单元写入数据，就能够由 FSMC 模拟出的 8080 接口向 ILI9341 写入控制命令或 GRAM 的数据了。

4.3.1 触摸屏感应原理

触摸屏常与液晶屏配套使用，组合成为一个可交互的输入输出系统。除了熟悉的电阻、电容屏外，触摸屏的种类还有超声波屏、红外屏。由于电阻屏的控制系统简单、成本低，且能适应各种上恶劣环境，被广泛采用。

电阻触摸屏的基本原理为分压，它由一层或两层阻性材料组成，在检测坐标时，在阻性材料的一端接参考电压 Vref，另一端接地，形成一个沿坐标方向的*均匀电场*。当触摸屏受到挤压时，阻性材料与下层电极接触，阻性材料被分为两部分，因而在*触摸点的电压，反映了触摸点与阻性材料的 Vref 端的距离，而且为线性关系*，而该触点的电压可由 ADC 测得。更改电场方向，以同样的方法，可测得另一方向的坐标。

4.3.2 TSC2046 触摸屏控制器

TSC2046 是专用在四线电阻屏的触摸屏控制器，MCU 可通过 SPI 接口向它写入控制字，由它测得 X、Y 方向的触点电压返回给 MCU。见图 0-7。

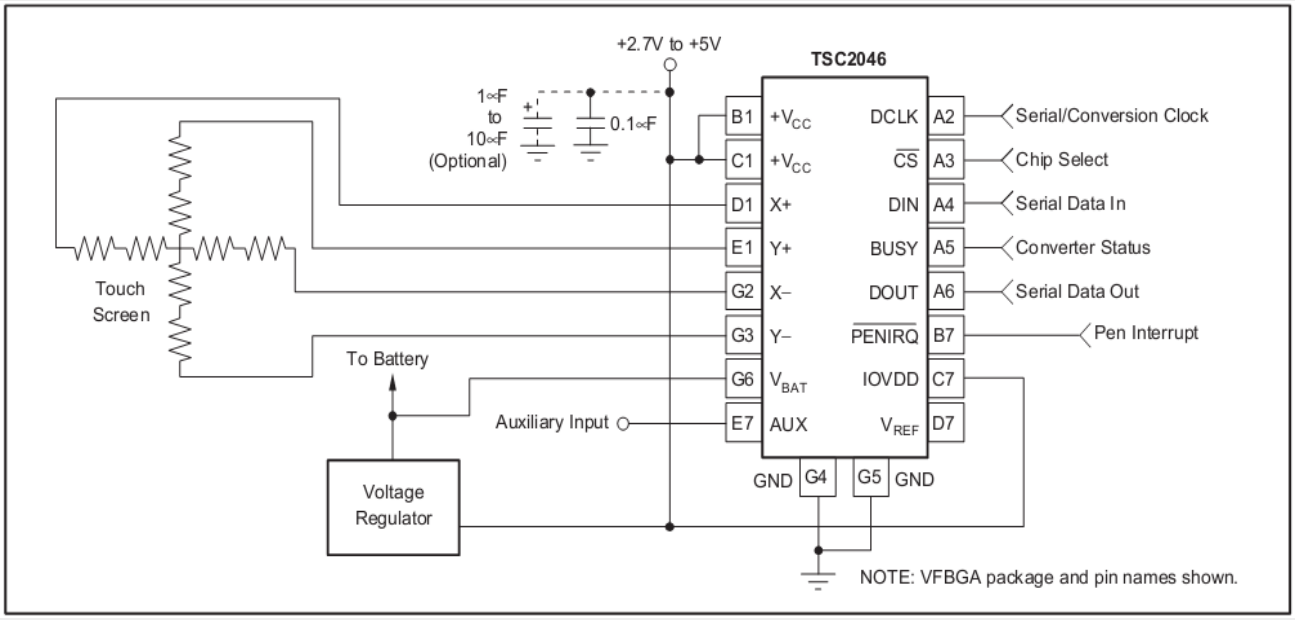


图 0-7 TSC2046 与电阻屏的连接图

图中，电阻屏两层阻性材料的两端分别接入到 TSC2046 的 X+、X-和 Y+、Y-。当要测量 X 坐标时，MCU 通过 SPI 接口写命令到 TSC2046，使它通过内部的*模拟开关*使 X+、X-接通电源，于是在电阻屏的 X 方向上产生一个匀强电场；把 Y+、Y-连接到 TSC2046 的 ADC。当电阻屏被触摸时，上、下两层的

阻性材料接触，在 *PENIRQ* 引脚产生一个*中断信号*，通知 MCU。该触点的电压由 Y+或 Y-(此时的 Y+Y-电阻很小，可忽略)引入到 ADC 进行测量，MCU 读取该电压，进行软件转换，就可以测得触点 X 方向的坐标。同理可以测得 Y 方向的坐标。

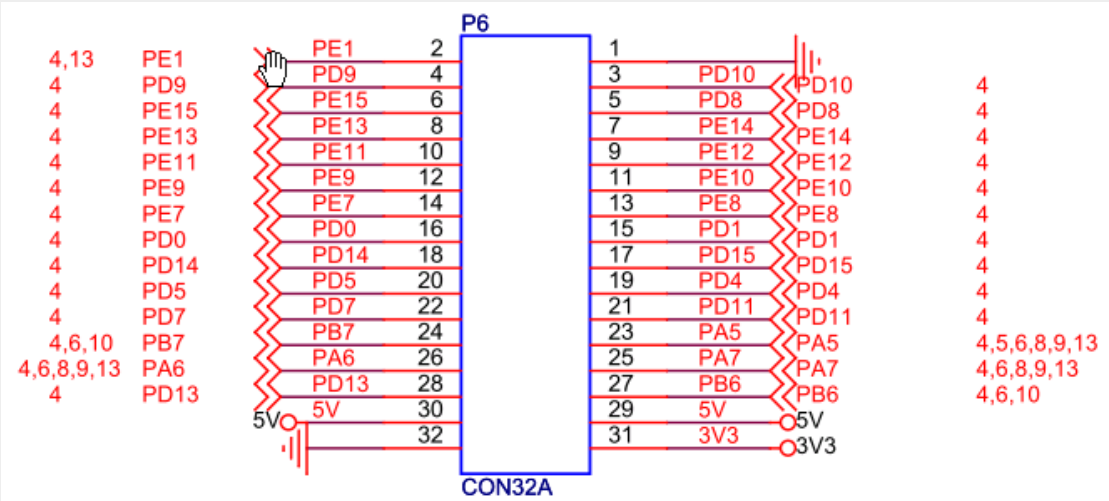
4.4 实验讲解

4.4.1 实验描述及工程文件清单

实验描述	野火 STM32 开发板驱动配套的 3.2 寸液晶、触摸屏，使用 FSMC 接口控制该屏幕自带的液晶控制器 ILI9341，使用 SPI 接口与触摸屏控制器 TSC2046 通讯。驱动成功后可在屏幕上使用基本的触摸绘图功能。
硬件连接	*TFT 数据线* PD14-FSMC-D0 ----LCD-DB0 PD15-FSMC-D1 ----LCD-DB1 PD0-FSMC-D2 ----LCD-DB2 PD1-FSMC-D3 ----LCD-DB3 PE7-FSMC-D4 ----LCD-DB4 PE8-FSMC-D5 ----LCD-DB5 PE9-FSMC-D6 ----LCD-DB6 PE10-FSMC-D7 ----LCD-DB7 PE11-FSMC-D8 ----LCD-DB8 PE12-FSMC-D9 ----LCD-DB9 PE13-FSMC-D10 ----LCD-DB10 PE14-FSMC-D11 ----LCD-DB11 PE15-FSMC-D12 ----LCD-DB12 PD8-FSMC-D13 ----LCD-DB13 PD9-FSMC-D14 ----LCD-DB14 PD10-FSMC-D15 ----LCD-DB15

	TFT 控制信号线
	PD4-FSMC-NOE ----LCD-RD
	PD5-FSMC-NEW ----LCD-WR
	PD7-FSMC-NE1 ----LCD-CS
	PD11-FSMC-A16 ----LCD-DC
	PE1-FSMC-NBL1 ----LCD-RESET
	PD13-FSMC-A18 ----LCD-BLACK-LIGHT
	触摸屏 TSC2046 控制线
	PA5-SPI1-SCK ----TSC2046-SPI -SCK
	PA7-SPI1-MOSI ----TSC2046-SPI - MOSI
	PA6-SPI1-MISO ----TSC2046-SPI – MISO
	PB7-I2C1-SDA ----TSC2046-SPI-CS
	PB6-I2C1-SCL ----TSC2046- INT_IRQ
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c *FWlib/misc.c* *FWlib/stm32f10x_spi.c* *FWlib/stm32f10x_rcc.c* *FWlib/stm32f10x_exti.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_fsmc.c*
用户编写的文件	USER/main.c USER/stm32f10x_it.c *USER/lcd.c* *USER/SysTick.c* *USER/lcd_botton.c* *USER/Touch.c*

野火 STM32 开发板 3.2 寸 LCD 硬件连接图(兼容 V1 版本的 2.4 寸屏)



4.4.2 配置工程环境

本 LCD 触摸屏画板实验中我们用到了 *GPIO、RCC、SPI、EXTI、FSMC* 外设，所以我们先要把以下库文件添加到工程：*stm32f10x_gpio.c* 、*stm32f10x_rcc.c*、*stm32f10x_spi.c*、*stm32f10x_exti.c*、*stm32f10x_fsmc.c*。由于在 TSC2046 的触摸检测中使用了中断，所以还要把 *misc.c* 文件添加进工程。

本工程使用了旧的用户文件 *SysTick.c*，用作定时，把它添加到新工程之中，并新建 *lcd_botton.c*、*lcd.c*、*Touch.c* 及相应的头文件。其中 *lcd_botton.c* 文件定义了最底层的 LCD 控制函数，LCD 上层的函数如画点、显示字符等位于 *lcd.c* 文件中。

最后在 *stm32f10x_conf.h* 中把使用到的 ST 库的头文件注释去掉。

```
1. /**
2.   ******************************************************
3.   * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.   * @author  MCD Application Team
5.   * @version V3.5.0
6.   * @date    08-April-2011
7.   * @brief   Library configuration file.
8.   ******************************************************/
9.
10.#include "stm32f10x_exti.h"
11.#include "stm32f10x_fsmc.h"
12.#include "stm32f10x_gpio.h"
13.#include "stm32f10x_rcc.h"
14.#include "stm32f10x_spi.h"
```

```
15. #include "misc.h"
```

4.4.3 main 文件

从本工程的 main 文件分析代码的执行流程：

```
1.  /*
2.   * 函数名：main
3.   * 描述  ：主函数
4.   * 输入  ：无
5.   * 输出  ：无
6.   */
7.  int main(void)
8.  {
9.      SysTick_Init();                         /*systick 初始化*/
10.     LCD_Init();                             /*LCD 初始化*/
11.     Touch_init();                           /*触摸初始化*/
12.
13.     while(Touchl_Calibrate() !=0);          /*等待触摸屏校准完毕*/
14.     Init_Palette();                         /*画板初始化*/
15.
16.     while (1)
17.     {
18.         if(touch_flag == 1)             /*如果触笔按下了*/
19.         {
20.                         /*获取点的坐标*/
21.             if(Get_touch_point(&display, Read_2046_2(), &touch_para )
    !=DISABLE)
22.             {
23.                         /*画点*/
24.                 Palette_draw_point(display.x,display.y);
                                        /*画点*/
25.             }
26.         }
27.     }
28. }
```

其执行流程如下：

1. 调用 *SysTick_Init()*、*LCD_Init()*、*Touch_init()*初始化了 STM32 的 Systick、FSMC、SPI 外设，并用 FSMC 和 SPI 接口初始化了 ILI9341 和 TSC2046 控制器。
2. 调用 *Touch1_Calibrate()*函数进行触摸屏校准，使得触摸屏与液晶屏的坐标匹配。
3. 调用 *Init_Palette()*函数初始化触摸画板的应用程序，使得在 LCD 上显示画板界面，并能够正常响应触摸屏的信号。
4. 第 16~27 行的 *while* 循环，通过不断检测触笔按下标志 *touch_flag*，判断触摸屏是否被触笔按下。触摸屏控制器 *TSC2046* 检测到触笔信号

时，由它的 *PENIRQ* 引脚触发 STM32 的中断，在中断服务函数中对 *touch_flag* 标志置 1。

5. 第 21 行，若检测到触笔按下，调用 *Get_touch_point()*函数读取 *TSC2046* 的寄存器，获得与触点的 X、Y 坐标相关的电压信号，转化成 LCD 的 X、Y 坐标。
6. 获取了触点坐标后，使 LCD 液晶屏在该坐标点显示为对应的颜色。
7. 循环触点捕捉、画点过程，就实现了触摸画板的功能。

4.4.5 初始化 FSMC 模式

4.4.5.1 初始化液晶屏流程

在 main 函数中调用的 *LCD_Init()*函数，它对液晶控制器 ILI9341 用到的 GPIO、FSMC 接口进行了初始化，并且向该控制器写入了命令参数，配置好了 LCD 液晶屏的基本功能。其函数定义位于 lcd_botton.c 文件，如下：

```
1.  /*****************************************
2.   * 函数名：LCD_Init
3.   * 描述   ：LCD 控制 I/O 初始化
4.   *          LCD FSMC 初始化
5.   *          LCD 控制器 HX8347 初始化
6.   * 输入   ：无
7.   * 输出   ：无
8.   * 举例   ：无
9.   * 注意   ：无
10. *****************************************/
11. void LCD_Init(void)
12. {
13.     unsigned long i;
14.
15.     LCD_GPIO_Config();       //初始化使用到的 GPIO
16.     LCD_FSMC_Config();       //初始化 FSMC 模式
17.     LCD_Rst();               //复位 LCD 液晶屏
18.     Lcd_init_conf();         //写入命令参数，对液晶屏进行基本的初始化配置
19.     Lcd_data_start();        //发送写 GRAM 命令
20.     for(i=0; i<(320*240); i++)
21.     {
22.         LCD_WR_Data(GBLUE); //发送颜色数据，初始化屏幕为 GBLUE 颜色
23.
24.     }
25. }
```

*LCD_Init()*函数执行后，最直观的结果是使 LCD 整个屏幕显示编码为 0X07FF 的 GBLUE 颜色。

函数中调用的 *LCD_GPIO_Config()*主要工作是把液晶屏(不包括触摸屏)中使用到的 GPIO 引脚和使能外设时钟，除了*背光、复位*用的 *PD13* 和 *PD1* 设置为*通用推挽输出*外，其它的与 FSMC 接口相关的*地址信号、数据信号、控制信号*的端口均设置为*复用推挽输出*。

4.4.5.2 初始化 FSMC 模式

接下来 *LCD_Init()*函数调用 *LCD_FSMC_Config()*设置 FSMC 的模式，我们的目的是使用它的 NOR FLASH 模式模拟出 8080 接口，在 LCD 接口中我们使用的是 FSMC 地址线 *A16* 作为 8080 的 *D/CX* 命令选择信号的。

*LCD_FSMC_Config()*具体代码如下：

```
/*******************************************
 * 函数名：LCD_FSMC_Config
 * 描述  ：LCD  FSMC 模式配置
 * 输入  ：无
 * 输出  ：无
 * 举例  ：无
 * 注意  ：无
*******************************************/
void LCD_FSMC_Config(void)
{
    FSMC_NORSRAMInitTypeDef  FSMC_NORSRAMInitStructure;
    FSMC_NORSRAMTimingInitTypeDef  p;


    p.FSMC_AddressSetupTime = 0x02;  //地址建立时间
    p.FSMC_AddressHoldTime = 0x00;   //地址保持时间
    p.FSMC_DataSetupTime = 0x05;          //数据建立时间
    p.FSMC_BusTurnAroundDuration = 0x00;   //总线恢复时间
    p.FSMC_CLKDivision = 0x00;            //时钟分频
    p.FSMC_DataLatency = 0x00;            //数据保持时间
    p.FSMC_AccessMode = FSMC_AccessMode_B;    //在地址数\据线不复用的情况下，ABCD 模式的区别不大
                                              //本成员配置只有使用扩展模式才有效


    FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM1;
   //NOR FLASH 的 BANK1
    FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable;       //数据线与地址线不复用
    FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_NOR;
                //存储器类型 NOR FLASH
    FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b;         //数据宽度为 16 位
    FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable;     //使用异步写模式，禁止突发模式
    FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;   //本成员的配置只在突发模式下有效，等待信号极性为低
```

```
31.     FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
                  //禁止非对齐突发模式
32.     FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalA
    ctive_BeforeWaitState;    //本成员配置仅在突发模式下有效。NWAIT 信号在什么时期
    产生
33.     FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disabl
    e;                //本成员的配置只在突发模式下有效，禁用 NWAIT 信号
34.     FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disabl
    e;                //禁止突发写操作
35.     FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperatio
    n_Enable;          //写使能
36.     FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Di
    sable;            //禁止扩展模式，扩展模式可以使用独立的读、写模式
37.     FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &p;
                  //配置读写时序
38.     FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &p;
                  //配置写时序
39.
40.
41.     FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
42.
43.     /* 使能 FSMC Bank1_SRAM Bank */
44.     FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM1, ENABLE);
45. }
```

本函数主要使用了两种类型的结构体对 FSMC 进行配置，第一种为 *FSMC_NORSRAMInitTypeDef* 类型的结构体主要用于 NOR FLASH 的模式配置，包括存储器类型、数据宽度等。另一种的类型为 *FSMC_NORSRAMTimingInitTypeDef*，这是用于配置 FSMC 的 NOR FLASH 模式下读写时序中的地址建立时间、地址保持时间等，代码中用它定义了结构体 *p*，这个第二种类型的结构体在前一种结构体中被指针调用。

下面先来分析 *FSMC_NORSRAMInitTypeDef* 的结构体成员：

1. *FSMC_Bank*

用于*选择外接存储器的区域*(或地址)，见图 0-8 FSMC 存储块，STM32 的存储器映射中，把 0x6000 0000~0x9fff ffff 的地址都映射到被 FSMC 控制的外存储器中，其中属于 NOR FLASH 的为 0x6000 0000~0x6fff ffff。*而属于 NOR FLASH 的这部分地址空间又被分为 4 份，每份大小为 64MB，编号为 BANK1 ~BANK4*。分 BANK 是由 FSMC 寻址范围决定的，该接口的地址线最多为 26 条，即最大寻址空间为 2^{26} =64MB。为了扩展寻址空间，可把地址与数据线与多片 NOR FLASH 并联，由不同的*片选信号 NE[3:0]*区分不同的 BANK。

在本实验中，我们使用的是 FSMC 的信号线 *NE1* 作为控制 8080 的 CSX 片选信号，所以我们把本成配置为 *FSMC_Bank1_NORSRAM1* (NE1 片选 BANK1)。

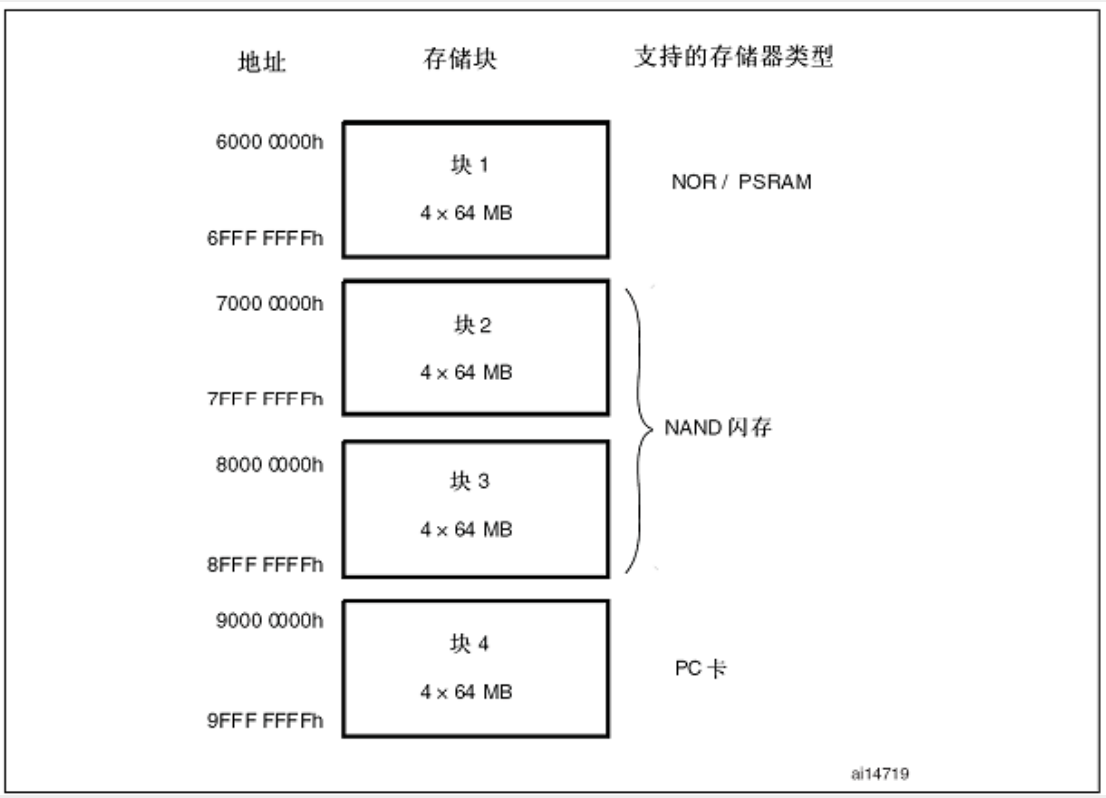


图 0-8 FSMC 存储块

2. *FSMC_DataAddressMux*

本成员用于配置 FSMC 的*数据线与地址线是否复用*。FSMC 支持数据与地址线复用或非复用两种模式。在*非复用模式*下 16 位数据线及 26 位地址线分开始用；*复用模式*则低 16 位数据/地址线复用。在复用模式下，推荐使用地址锁存器以区分数据与地址。当 *NADV 信号线*为低时，复用信号线 ADx(x=0...15)上出现地址信号 Ax，当 NADV 变高时，ADx 上出现数据信号 Dx。

本实验中用 FSMC 模拟 8080 接口，*地址线 A16* 提供 8080 的 *D/CX 信号*，实际上就只使用了这一条地址线，I\O 资源并不紧张，所以把本成员配置为 *FSMC_DataAddressMux_Disable* (非复用模式)。

3. *FSMC_MemoryType*

本成员用于配置 FSMC 外接的*存储器的类型*，可被配置为 *NOR FLASH 模式*、*PSARM 模式及 SRAM 模式*。

在本实验的应用中，由于 *NOR FLASH 模式的时序与 8080 更接近*，所以本结构体被配置为 *FSMC_MemoryType_NOR*(NOR FLASH 模式)。

4. *FSMC_MemoryDataWidth*

本成员用于设置 FSMC 接口的*数据宽度*，可被设置为 8Bit 或 16bit。对于 16 位宽度的外部存储器。在 STM32 地址映射到 FSMC 接口的结构中，HADDR 信号线是需要转换到外部存储器的内部 AHB 地址线，是*字节地址*。

若存储器的数据线宽为 8Bit，FSMC 的 *26* 条地址信号线 FSMC_A[25:0]直接可以引入到与 AHB 相连的 HADDR[25:0]，*26* 条*字节地址*信号线最大寻址空间为 64MB。见图 0-9。

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 x 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 x 16 = 512 M位

图 0-9 外部存储器地址

若存储器的数据线宽 16Bit，则存储器的地址信号线是*半字地址(16Bit)*。为了使 HADDR 的*字节地址*信号线与存储器匹配，FSMC 的 *25* 条地址信号线 FSMC_A[24:0]与 HADDR[25:1]相连，由于变成了*半字地址(16Bit)*，仅需要 *25* 条*半字字地址*信号线就达到最大寻址空间 64MB。正因地址线的不对称相连，*16bit 数据线宽下，实际的访问地址为右移一位之后的地址*。

本实验中 8080 接口采用 16bit 模式，所以我们把本成员配置为 *FSMC_MemoryDataWidth_16b*，由于地址线不对称相连，这会影响到我们用*地址信号线 A16* 控制的 8080 接口的 *D/CX 信号*。

5. *FSMC_BurstAccessMode*

本成员用于配置*访问模式*。FSMC 对存储器的访问分为异步模式和突发模式(同步模式)。在*异步模式*下，每次传送数据都需要产生一个确定的地址，而*突发模式*可以在开始始提供一个地址之后，把数据成组地连续写入。

本实验中使用 FSMC 模拟 8080 端口，更适合使用异步模式，因而向本成员赋值为 *FSMC_WriteBurst_Disable*。

6. 突发模式参数配置

代码中的 30~34 行，都是关于使用突发模式时的一些参数配置，这些成员为： *FSMC_WaitSignalPolarity(配置等待信号极性)*、*FSMC_WrapMode(配置是否使用非对齐方式)*、*FSMC_WaitSignalActive(配置等待信号什么时期产生)*、*FSMC_WaitSignal（配置是否使用等待信号）*、*FSMC_WriteBurst(配置是否允许突发写操作)*，这些成员均需要在突发模式开启后配置才有效。

这些成员在开启突发模式时才有效，本实验使用的是异步模式，所以这些成员的参数没有意义。

7. *FSMC_WriteOperation*

本成员用于配置*写操作使能*，如果禁止了写操作，FSMC 不会产生写时序，但仍可从存储器中读出数据。

本实验需要写时序，所以向本成员赋值为 *FSMC_WriteOperation_Enable*(写使能)

8. *FSMC_ExtendedMode*

本成员用于配置是否使用*扩展模式*，在扩展模式下，读时序和写时序可以使用独立时序模式。如读时序使用模式 A，写时序使用模式 B，这些 A、B、C、D 模式实际上差别不大，主要是在使用*数据/地址线*复用的情况下，NADV 信号产生的时序不一样，具体的时序图可查阅《STM32 参考手册》。

本实验中数据/地址线不复用，所以读写时序中不同的 NADV 信号并没影响，禁止使用扩展模式 *SMC_ExtendedMode_Disable*。

9. *FSMC_ReadWriteTimingStruct 及 FSMC_WriteTimingStruct*

这两个参数分别用来设置 FSMC 的*读时序及写时序的时间参数*。若使用了扩展模式，则前者配置的是读时序，后者为写时序；若禁止了扩展模式，则读写时序都使用 *FSMC_ReadWriteTimingStruct* 结构体中的参数。

在配置这两个参数时，使用的是类型 *FSMC_NORSRAMTimingInitTypeDef* 时序初始化结构体，对这种类型结构体的成员进行赋值。它的成员分别有：*FSMC_AddressSetupTime(地址建立时间)*、*FSMC_AddressHoldTime(地址保持时间)*、*FSMC_DataSetupTime(数据建立时间)*、*FSMC_DataLatency(数据保持时间)* 、*FSMC_BusTurnAroundDuration(总线恢复时间)*、*FSMC_CLKDivision(时钟分频)*、*FSMC_AccessMode(访问模式)*。对以上各个时间成员赋的数值 X 表示 X 个时钟周期，它的时钟是由 HCLK 经过成员时钟分频得来的，该分频值在成员 *FSMC_CLKDivision(时钟分频)*中设置。其中 *FSMC_AccessMode(访问模式)*成员的设置只在开启了扩展模式才有效，而且开启了扩展模式后，读时序和写时序的设置可以是独立的。

本实验中的时序设置是根据 ILI9341 的 datasheet 设置的，调试的时候可以先把这些值设置得大一些，然后慢慢靠近 datasheet 要求的最小值，这样会取得比较好的效果。时序的参数设置对 LCD 的显示效果有一定的影响。

配置完初始化结构体后，要调用库函数 *FSMC_NORSRAMInit()*把这些配置参数写到控制寄存器，还要调用 *FSMC_NORSRAMCmd()*使能 *BANK1*。如果是使用 FSMC 配置其它存储器如 NAND FLASH，要使用其它的库函数及初始化结构体。

4.4.6 FSMC 模拟 8080 读写参数、命令

回到 *LCD_Init()*函数的执行流程，初始化完成 FSMC 接口后，就可以使用它控制 ILI9341 了。在 *LCD_Init()*中调用了 *Lcd_init_conf()*函数向 ILI9341 写入了一系列的控制参数：

```
1.  /*********************************
2.   * 函数名：Lcd_init_conf
3.   * 描述   ：ILI9341 LCD 寄存器初始配置
4.   * 输入   ：无
5.   * 输出   ：无
6.   * 举例   ：无
7.   * 注意   ：无
8.  *********************************/
9.  void Lcd_init_conf(void)
10. {
11.     DEBUG_DELAY();
12.     LCD_ILI9341_CMD(0xCF);
13.     LCD_ILI9341_Parameter(0x00);
```

```
    LCD_ILI9341_Parameter(0x81);
    LCD_ILI9341_Parameter(0x30);

    DEBUG_DELAY();
    LCD_ILI9341_CMD(0xED);
    LCD_ILI9341_Parameter(0x64);
    LCD_ILI9341_Parameter(0x03);
    LCD_ILI9341_Parameter(0x12);
    LCD_ILI9341_Parameter(0x81);

    DEBUG_DELAY();
    LCD_ILI9341_CMD(0xE8);
    LCD_ILI9341_Parameter(0x85);
    LCD_ILI9341_Parameter(0x10);
    LCD_ILI9341_Parameter(0x78);
  // ·····此处省略几十行·····
```

本函数十分长，由于篇幅问题，以上只是该函数其中的一部分，省略部分的代码也是这样的模板，只是写入的命令和参数不一样而已，这些命令和参数设置了像素点颜色格式、屏幕扫描方式、横屏\竖屏等初始化配置，这些命令的意义从 ILI9341 的 datasheet 命令列表中可以查到。该函数通过调用 *LCD_ILI9341_CMD()*写入*命令*，用 *LCD_ILI9341_Parameter()*写入*参数*。它们实质是两个宏：

```
/*****************************************************/
#define LCD_ILI9341_CMD(index)          LCD_WR_REG(index)
#define LCD_ILI9341_Parameter(val)      LCD_WR_Data(val)

/****为了移植方便，上面的宏只是封装，以下才是最底层的宏**********/
/* 选择 BANK1-BORSRAM1 连接 TFT，地址范围为 0X60000000~0X63FFFFFF
 * FSMC_A16 接 LCD 的 DC(寄存器/数据选择)脚
 * 16 bit => FSMC[24:0]对应 HADDR[25:1]
 * 寄存器基地址 = 0X60000000
 * RAM 基地址 = 0X60020000 = 0X60000000+2^16*2 = 0X60000000 + 0X20000 = 0X60020000
 * 当选择不同的地址线时，地址要重新计算。
 */
#define Bank1_LCD_D     ((u32)0x60020000)       //Disp Data ADDR
#define Bank1_LCD_C     ((u32)0x60000000)       //Disp Reg ADDR

/*选定 LCD 指定寄存器（命令编码）*/
#define LCD_WR_REG(index)     ((*(__IO u16 *) (Bank1_LCD_C)) = ((u16)index))

/*往 LCD 写入数据*/
#define LCD_WR_Data(val)        ((*(__IO u16 *) (Bank1_LCD_D)) = ((u16)(val)))
```

这部分是 FSMC 模拟 8080 接口的精髓。

读写参数、命令

先来看第 21 行的宏，*LCD_WR_Data(val)*，这是一个带参宏，用于向 LCD 控制器写入参数，参数为 val。它的宏展开为：

```
1.  ((*(__IO u16 *) (Bank1_LCD_D)) = ((u16)(val)))
```

宏展开中的*(Bank1_LCD_D)*是一个在第 14 行定义的宏，它的值为 0x6002 0000，实质是一个地址，这个地址的计算在后面介绍。

*(__IO u16 *) (Bank1_LCD_D)*表示把*(Bank1_LCD_D)*强制转换成一个 16 位的地址。*((*(__IO u16 *) (Bank1_LCD_D))*表示再对这个地址作”*”指针运算，取该指针对象的内容，并把它的内容赋值为 *= ((u16)(val)))*。所以整个宏的操作就是：把参数 val 写入到地址为 0x6002 0000 的地址空间。

由于这个地址被 STM32 映射到外存储器，所以会由 FSMC 外设以访问 NOR FLASH 的形式、时序，在地址线上发出 0x6002 0000 地址信号，在数据线上发出 val 数据信号，写入参数到外存储器中。而 FSMC 接口又被我们模拟成了 8080 接口，最终 val 被 8080 接口理解为参数，传输到 ILI9341 控制器中。

计算地址

见图 0-10。计算地址前，再明确一下在本实验中，使用的是 *FSMC_NE1* 作为 *8080_CS 片选信号*，以 *FSMC_A16* 作为 *8080_D/CX 数据/命令信号*(图中为 RS，意义相同)。

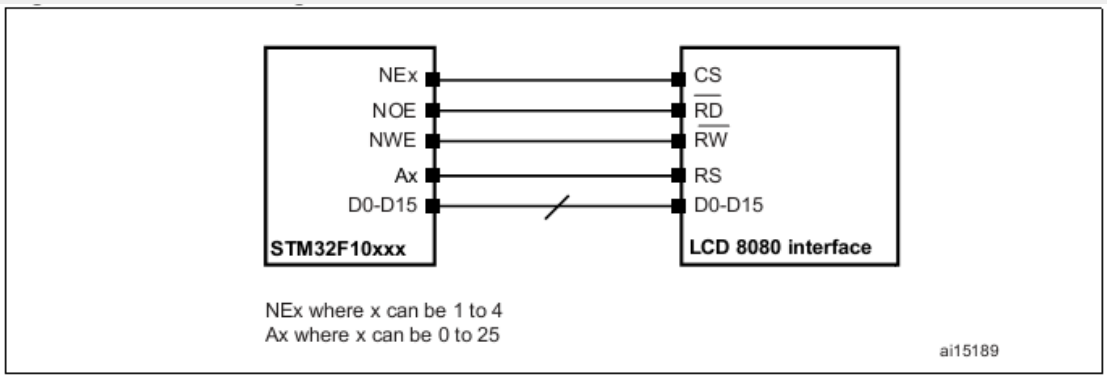


图 0-10 FSMC 与 8080 端口连接简图

按这种连接时，FSMC_NE1 为*低电平*、FSMC_A16 为*高电平*，表示通过 D[15:0]发送\接收的数据被 8080 接口解释为*参数(数值)*，当我们访问 *0x6002 0000* 这个地址的时候，正好符合这个条件。该地址的计算过程如下：

由于选择的是使用 FSMC_NE1 片选信号线，片选的为 BANK1，所以基地址为 0x6000 0000。要把地址线 FSMC_A16 置为*高电平*，可以采用下列算式：

```
1.  0x6000 0000 |= 1<<16;  //结果 = 0x6001 0000
```

但是，这样计算出来的地址只是数据线为 8Bit 模式下的字节地址。由于我们采用的是 16Bit 数据线，FSMC[24:0]与 HADDR[25:1]对齐，HADDR 地址要左移一位才是 FSMC 的访问地址。因此为了把 FSMC 中的 *FSMC_A16 (D/CX 线)*置 1，实际上要对应到 HADDR 地址(AHB 地址)的 *HADDR_A17*，即正确的计算地址公式应为：

```
1.  0x6000 0000 |= 1<<(16+1);  //此为正确结果 = 0x6002 0000
```

对于 16 位数据线模式，能使 FSMC_NEX 为低电平，FSMC_A16 为高电平的地址，并不只有 *0x6002 0000* 一个，只要是属于 *0X60000000~0X63FFFFFF* 范围内(BANK1 地址范围)，HADDR_A17 位为高电平的地址均可，这是因为我们只采用了 FSMC_A16 用于 8080 的 D/CX 信号，所以地址线的电平状态并无影响。如 0x6002 0001 地址，在本实验中是与 0x6002 0000 等价的。

若修改这个地址，使 FSMC_NEX 为低电平，FSMC_A16 为*低电平*，即 D/CX 被置 0，*8080 会把由数据线传输的信号理解为命令*。以同样的方式计算，符合这样要求的其中一个地址为 *0x6000 0000*，向这个地址空间赋值，这个值最终会被 8080 接口解释为*命令*。宏 *LCD_WR_REG(index)* 就是这样实现的。

给整个屏幕上色

再次回到 *LCD_Init()*函数，它调用完 *Lcd_init_conf()*初始化了液晶屏后，向使用 *Lcd_data_start()*函数发送了一个命令——写 GRAM 内容，即后面发送的数据都被解析为显示到屏幕像素点的数据。代码中使用 for 循环把语句

*LCD_WR_Data(GBLUE)*执行了 320*240 次，即把所有像素点都显示为 GBLUE 颜色。

4.4.6.1 液晶屏画点函数

初始化了液晶屏后，就可以控制液晶上每个像素点的颜色了。如果能够实现一个*画点函数*，在指定的(x,y)坐标像素点上显示指定的颜色，那么就能够实现一切液晶屏最复杂的显示功能，如在液晶屏指定位置显示形状、文字、图像，都可以通过调用画点函数或以类似的方式控制液晶的像素点。本实验中的画点函数 *LCD_Point()*在 *lcd.c* 文件中定义如下：

```
/*****************************************************
 * 函数名：LCD_Point
 * 描述   ：在指定坐标处显示一个点
 * 输入   ：-x 横向显示位置 0~319
 *          -y 纵向显示位置 0~239
 * 输出   ：无
 * 举例   ：    LCD_Point(100,200);
 *            LCD_Point(10,200);
 *            LCD_Point(300,220);
 * 注意   ：    (0,0)位置为液晶屏左上角 已测试
*****************************************************/
void LCD_Point(u16 x,u16 y)
{
    LCD_open_windows(x,y,1,1);
    LCD_WR_Data(POINT_COLOR);
}
```

本函数首先调用了一个液晶显示窗口函数 *LCD_open_windows()*用于开辟液晶屏上的显示区域，后面写入的颜色数据将被显示到该区域中。示窗函数按*(x,y,1,1)*的参数调用时，该函数开辟了一个坐标为(x,y)的像素点。接着调用 *LCD_WR_Data()*向 ILI9341 写入颜色数据，像素的坐标即为示窗函数开辟的显示坐标。于是 *LCD_Point()*函数就实现了控制特定坐标的像素点。

接下来分析 *LCD_open_windows()*函数是如何开辟显示区域的：

```
/*********************************
 * 函数名：LCD_open_windows
 * 描述   ：开窗(以 x,y 为坐标起点，长为 len,高为 wid)
 * 输入   ：-x     窗户起点
 *          -y       窗户起点
 *          -len   窗户长
 *          -wid 窗户宽
 * 输出   ：无
 * 举例   ：无
```

```
 * 注意  : 无
**********************************/
void LCD_open_windows(u16 x,u16 y,u16 len,u16 wid)
{

    if(display_direction == 0)       /*如果是横屏选项*/
    {

        LCD_ILI9341_CMD(0X2A);
        LCD_ILI9341_Parameter(x>>8);     //start 起始位置的高 8 位
        LCD_ILI9341_Parameter(x-((x>>8)<<8));  //起始位置的低 8 位
        LCD_ILI9341_Parameter((x+len-1)>>8);     //end 结束位置的高 8 位
        LCD_ILI9341_Parameter((x+len-1)-(((x+len-1)>>8)<<8));  //结束位置的低 8 位

        LCD_ILI9341_CMD(0X2B);
        LCD_ILI9341_Parameter(y>>8);    //start
        LCD_ILI9341_Parameter(y-((y>>8)<<8));
        LCD_ILI9341_Parameter((y+wid-1)>>8);    //end
        LCD_ILI9341_Parameter((y+wid-1)-(((y+wid-1)>>8)<<8));

    }
    else
    {
        LCD_ILI9341_CMD(0X2B);
        LCD_ILI9341_Parameter(x>>8);
        LCD_ILI9341_Parameter(x-((x>>8)<<8));
        LCD_ILI9341_Parameter((x+len-1)>>8);
        LCD_ILI9341_Parameter((x+len-1)-(((x+len-1)>>8)<<8));

        LCD_ILI9341_CMD(0X2A);
        LCD_ILI9341_Parameter(y>>8);
        LCD_ILI9341_Parameter(y-((y>>8)<<8));
        LCD_ILI9341_Parameter((y+wid-1)>>8);
        LCD_ILI9341_Parameter((y+wid-1)-(((y+wid-1)>>8)<<8));

    }

      LCD_ILI9341_CMD(0x2c);
}
```

本函数主要使用了三个 ILI9341 的控制命令：

1. *0x2A 列地址控制命令*

   用于设置显示区域的*列像素区域*。它有四个参数，分别为 SC 的高 8 位、SC 的低 8 位及 EC 的高 8 位、EC 的低 8 位。SC 和 EC 的意义见图 0-11。*SC*表示要控制的显示区域的*列起始坐标*，*EC*表示显示区域的*列结束坐标*。

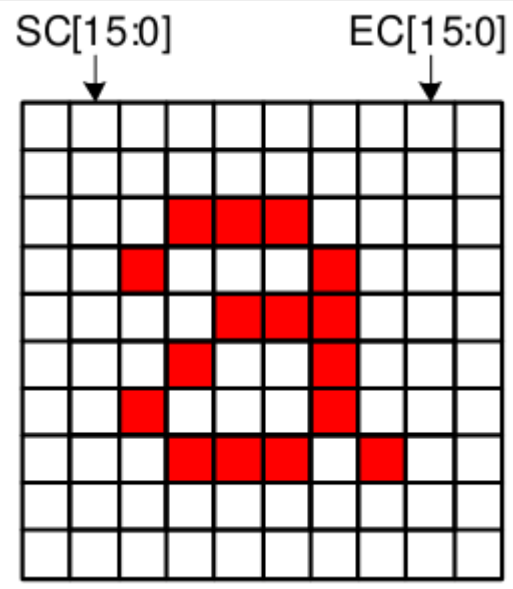
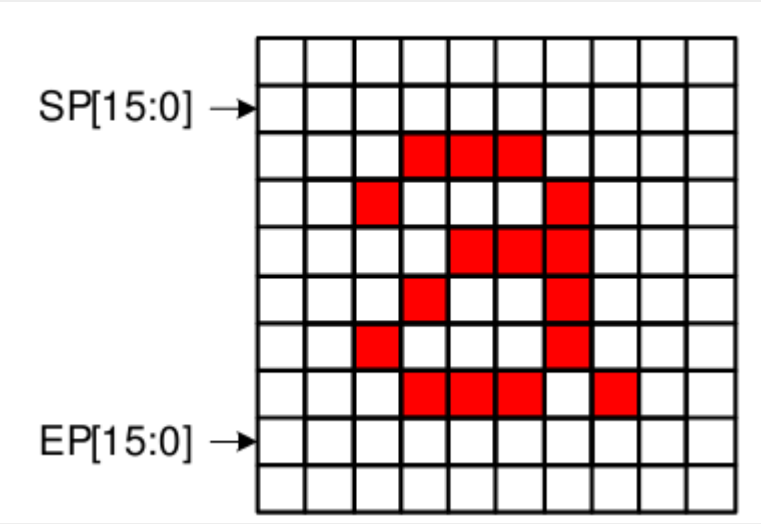


图 0-11 列控制命令 SC 和 EC

2. *0x2B 页地址控制命令*

用于设置显示区域的 *页(行)像素区域*。与上一命令类似，也有四个参数，分别为 SP 的高 8 位、SP 的低 8 位及 EP 的高 8 位、EP 的低 8 位。SP 和 EP 的意义见**错误！未找到引用源。**。*SP*表示要控制的显示区域的*行起始坐标*，*EP*表示显示区域的*行结束坐标*。

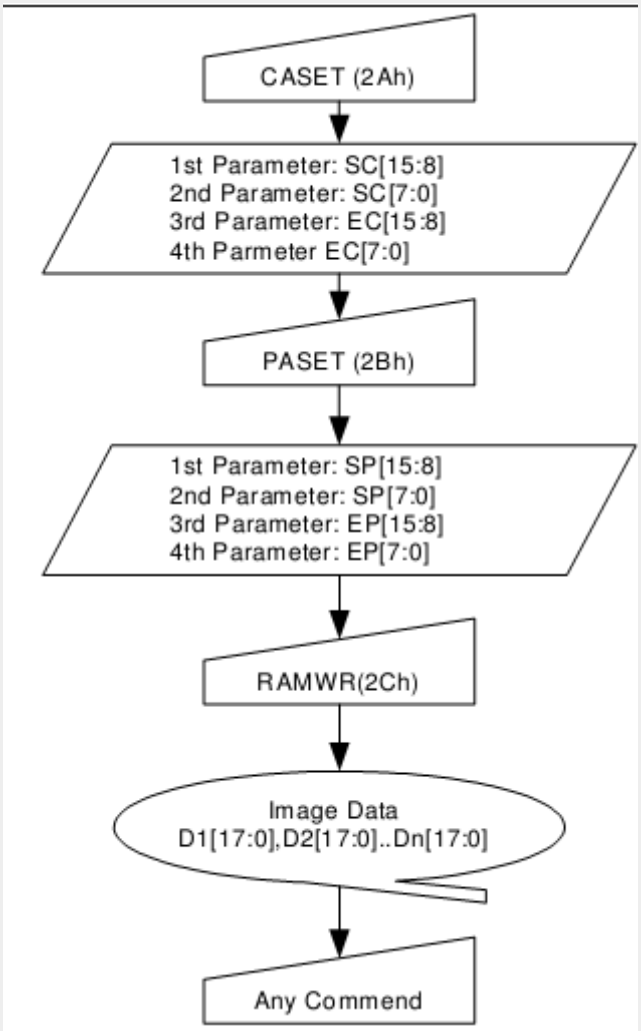


3. *0x2C 写 RAM 命令*

本命令用于表示*开始写入像素显示数据*，紧跟着本命令后面的即为写入到 GRAM 的 RGB 5：6：5 的颜色数据。在初始化液晶屏函数中也调用到本命令。使用本命令，后面的颜色数据按照一个接着一个预设的扫描方式(扫描方式决定了是横屏和竖屏)写入到由 0x2A 和 0x2B 设置的显

示区域中。横屏扫描方式为从左到右，扫描完一行(页)像素点再扫描下一行(页)。竖屏扫描为从上到下，扫描完一列再扫描下一列。

了解这三个命令后，就知道示窗函数的执行流程了。



调用本示窗函数后，就可以开辟一个起始坐标为(x,y)，长为 len 宽为 wid 的矩形显示窗口了。特别地，当 len=wid=1 时， 开辟的为一个坐标为(x,y)的像素点。

4.4.6.2 触摸屏校正

在 main 函数初始化完成 LCD 之后，调用了 *Touchl_Calibrate()*函数进行触摸屏校正。

```
1. /*****************************************************
2. * 函数名：Touchl_Calibrate
3. * 描述   ：触摸屏校正函数
```

```
* 输入  : 无
* 输出  : 0    --- 校正成功
                  1    --- 校正失败
* 举例  : 无
* 注意  : 无
*******************************************************/
int Touchl_Calibrate(void)
{
    uint8_t i;
    u16 test_x=0, test_y=0;
    u16 gap_x=0, gap_y=0;
    Coordinate * Ptr;
   // delay_init();
    Set_direction(0);    //设置为横屏
    for(i=0;i<4;i++)
    {
        LCD_Rectangle(0,0,320,240,CAL_BACKGROUND_COLOR);   //使整个屏幕显示背景颜色
        LCD_Str_6x12_O(10, 10,"Touch Calibrate", 0);   //显示提示信息
        LCD_Num_6x12_O(10,25, i+1,  0);                //显示触点次数

        delay_ms(500);
        DrawCross(DisplaySample[i].x,DisplaySample[i].y);  //显示校正用的"十"字
        do
        {
            Ptr=Read_2046();  //读取 TSC2046 数据到变量 ptr
        }
        while( Ptr == (void*)0 );  //当 ptr 为 0 时表示没有触点被按下
        ScreenSample[i].x= Ptr->x;     //把读取的原始数据存放到 ScreenSample 结构体
        ScreenSample[i].y= Ptr->y;

    }

        /* 用原始参数计算出 原始参数与坐标的转换系数。 */
    Cal_touch_para( &DisplaySample[0],&ScreenSample[0],&touch_para ) ;

        /*计算 X 值*/
    test_x = ( (touch_para.An * ScreenSample[3].x) +
               (touch_para.Bn * ScreenSample[3].y) +
               touch_para.Cn
               ) / touch_para.Divider ;

        /*计算 Y 值*/
    test_y = ( (touch_para.Dn * ScreenSample[3].x) +
               (touch_para.En * ScreenSample[3].y) +
               touch_para.Fn
               ) / touch_para.Divider ;

    gap_x = (test_x > DisplaySample[3].x)?(test_x - DisplaySample[3].x):(DisplaySample[3].x - test_x);
    gap_x = (test_y > DisplaySample[3].y)?(test_y - DisplaySample[3].y):(DisplaySample[3].y - test_y);


     LCD_Rectangle(0,0,320,240,CAL_BACKGROUND_COLOR);
    if((gap_x>11)||(gap_y>11))
    {

        LCD_Str_6x12_O(100, 100,"Calibrate fail", 0);
        LCD_Str_6x12_O(100, 120,"  try again   ", 0);
        delay_ms(2000);
        return 1;
```

```
63.     }
64.
65.
66.     aa1 = (touch_para.An*1.0)/touch_para.Divider;
67.     bb1 = (touch_para.Bn*1.0)/touch_para.Divider;
68.     cc1 = (touch_para.Cn*1.0)/touch_para.Divider;
69.
70.     aa2 = (touch_para.Dn*1.0)/touch_para.Divider;
71.     bb2 = (touch_para.En*1.0)/touch_para.Divider;
72.     cc2 = (touch_para.Fn*1.0)/touch_para.Divider;
73.
74.     LCD_Str_6x12_O(100, 100,"Calibrate Success", 0);
75.     delay_ms(1000);
76.
77.     return 0;
78. }
```

本函数的主要作用是在指定的几个*液晶屏坐标*(逻辑坐标)显示"十"字交叉点，由用户使用触笔点击触摸屏交叉点，读取由 TSC2046 测得的*触点电压*(物理坐标)。采集 4 个不同位置的触点电压(物理坐标)，然后根据触摸校准算法*把逻辑坐标与物理坐标转换公式的系数 A、B、C、D、E、F 计算出来*。

若使用此函数校准成功后，用户再点击触摸屏时，*可把测量出的触点电压(物理坐标)代入已知系数的转换公式，计算出对应的液晶屏坐标(逻辑坐标)*。

转换公式的系数为以上代码 66~72 行中的 *aa1、bb1、cc1、aa2、bb2、cc3* 这几个全局变量，如果把这几个数据保存在非易失性存储器（SD 卡、EEPROM 等）中，上电后向这几个变量赋值，就不需要每次上电都进行一次触屏校准了。

本函数中大部分都是关于触摸屏校准算法的数学运算，有兴趣的读者可查阅其它相关资料来理解。在代码中的 18~30 行，与触摸屏的触点电压获取有关，分析如下：

1. 第 20~25 行，调用 *LCD_Rectangle()、LCD_Str_6x12_O()、LCD_Num_6x12_O()、DrawCross()*由液晶屏显示背景、提示信息及校准用的“十”字。这些函数都与液晶的画点函数原理类似，关于字符显示的在下一个章节进行说明。
2. 第 28 行，调用 *Read_2046()*函数获取触点的电压，该函数通过向 TSC2046 控制器发送控制命令：若触笔点击触摸屏时采集触点的电压，采集 10 个电压取平均值，结果返回给变量 Ptr；若没有触点，则

Ptr 的值为 0，由 do-while 循环等待至采集到数据为止。Ptr 中保存的电压数据在后面被用于校准算法计算。

*Read_2046()*函数定义如下：

```
/*******************************************************
* 函数名：Read_2046
* 描述   ：得到滤波之后的 X Y
* 输入   ：无
* 输出   ：Coordinate 结构体地址
* 举例   ：无
* 注意   ：速度相对比较慢
*******************************************************/
Coordinate *Read_2046(void)
{
    static Coordinate  screen;
    int m0,m1,m2,TP_X[1],TP_Y[1],temp[3];
    uint8_t count=0;

        /* 坐标 X 和 Y 进行 9 次采样*/
    int buffer[2][9]={{0},{0}};
    do
    {
        Touch_GetAdXY(TP_X,TP_Y);
        buffer[0][count]=TP_X[0];
        buffer[1][count]=TP_Y[0];
        count++;

    }   /*用户点击触摸屏时即 TP_INT_IN 信号为低 并且 count<9*/
    while(!INT_IN_2046&& count<9);

//由于篇幅问题，此处省略很多行，省略部分主要为计算 10 个采样电压的平均值

                ..................................
```

在 *Read_2046()*函数中，调用了 *Touch_GetAdXY()*，它用于获取一次触点(x,y)电压。实际上，驱动 TSC2046 最底层的是命令 *WR_CMD(CHX)*和 *WR_CMD(CHY)*，发送了这两个命令后，TSC2046 开始采集相应的触点电压，通过 SPI 传送触点电压数据到 STM32。

命令语句中的 CHX 宏展开为 0xd0，CHY 为 0x90，它们是根据 TSC2046 的命令格式设定的。驱动 TSC2046 的命令控制字格式。

位 7 (MSB)	位 6	位 5	位 4	位 3	位 2	位 1	位 0 (LSB)
S	A2	A1	A0	MODE	SER/$\overline{DFR}$	PD1	PD0

其中 S 为数据传输起始标志位该位必为 1，A2~A0 进行通道选择 MOD 用于转换 精度选择，1 为 8 位精度，0 为 12 位精度。

A2	A1	A0	+REF	-REF	Y-	X+	Y+	Y-位置	X-位置	Z1-位置	Z2-位置	驱 动
0	0	1	Y+	Y-		+IN		M				Y+,Y-
0	1	1	Y+	X-		+IN				M		Y+,X-
1	0	0	Y+	Y+	+IN						M	Y+,X-
1	0	1	X+	X-			+IN		M			X+,X-

所谓通道选择即为检测哪一个通道的坐标。如 A2~A0 为 001 时，即命令控制字为 0x90，根据表格知，芯片会给触摸屏的 Y 阻性材料层的两端提供 Y+、Y-的电压，若有触笔点击，则 Y 触点电压可经过 X+利用 ADC 读取得。同理命令控制字为 0xd0 时，A2~A0 为 101，即给 X 阻性材料层提供电压，触点电压经过 Y+由 ADC 读取得。这就是 TSC2046 采集触点电压的原理。

4.4.6.3 检测触点、画点

回到 main 函数。触摸屏也校准好后，剩下的就是应用程序代码了，调用 *Init_Palette()*使液晶屏显示出画板的界面， (由于印刷原因，无法分辨具体颜色)

该画板的界面左侧为各种颜色方块，右侧为提供给用户进行绘画的空间。显示完该界面后，循环检测 touch_flag 标志，它在中 *stm32f10x_it.c* 文件中的中断服务函数被赋值。当触摸屏被按下时会进入该函数，对 touch_flag 赋值，如下：

```
1.  void EXTI9_5_IRQHandler(void)
2.  {
3.
4.
5.    if(EXTI_GetITStatus(EXTI_Line6) != RESET)
6.    {
```

```
7.       GPIO_ResetBits(GPIOB, GPIO_Pin_5);

8.
9.              touch_flag=1;
10.
11.     EXTI_ClearITPendingBit(EXTI_Line6);
12.   }
13. }
```

若 touch_flag 标志为 1，即触摸屏被按下，main 中调用函数 *Get_touch_point()*，该函数通过 *Read_2046_2()*获取触点电压，根据公式把电压转换为液晶坐标，并保存到 *display* 结构体中。得到液晶坐标后，main 调用 *Palette_draw_point()*对相应的坐标点进行处理，若触点位于画板界面的颜色方块中，则使画笔变为该颜色，其后在空白界面的触点将显示该颜色的笔迹。

4.5 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，接上液晶屏，将编译好的程序下载到开发板。运行后，可在 LCD 屏幕看到提示信息“Touch Calibrate” 和触摸屏校正用的“十”字。点击“十”字的中间（共四个），若校正成功后会出现画板界面，在画板界面的右侧可进行绘画，点击左侧的颜色块可选择笔迹的颜色。

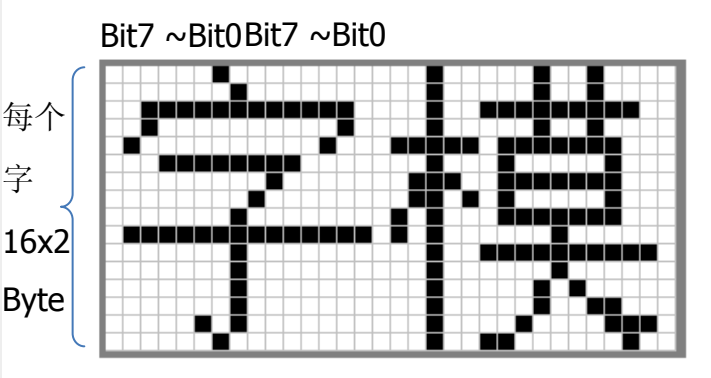
5、液晶显示（中、英、Pic）

5.1 实验简介

在《液晶触摸画板》中，我们已经成功地实现了驱动 LCD 和触摸屏，并制作了触摸画板小应用，但是若要显示文字或图片文件，则还需要利用文件系统，读取保存在 SD 卡中的字库文件、图片文件。

5.2 什么是字模

我们知道其实液晶屏就是一个由像素点组成的点阵，若要显示文字，则需要很多像素点的共同构成。见下**错误！未找到引用源。**，图中是两个由 16*16 的点阵显示的两个汉字。



如果我们规定：*每个汉字都由这样 16*16 的点阵来显示，把笔迹经过的像素点以“1”表示，没有笔迹的点以“0”表示，每个像素点的状态以一个二进制位来记录，用 16*16/8 =32 个字节就可以把这个字记录下来。*这 32 个字节数据就称为该文字的字模，还有其它常用字模是 24*24、32*32 的。16*16 的“字”的字模数据为：

```
/* 字 */
unsigned char code Bmp003[]=
{
/*------------------------------------------------------------
;  源文件 / 文字 : 字
;  宽×高（像素）: 16×16
;  字模格式/大小 : 单色点阵液晶字模，横向取模，字节正序/32 字节
------------------------------------------------------------*/

```

```
10. 0x02,0x00,0x01,0x00,0x3F,0xFC,0x20,0x04,0x40,0x08,0x1F,0xE0,0x00,0x40,
    0x00,0x80,
11. 0xFF,0xFF,0x7F,0xFE,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x05,0x00,
    0x02,0x00,
12. };
```

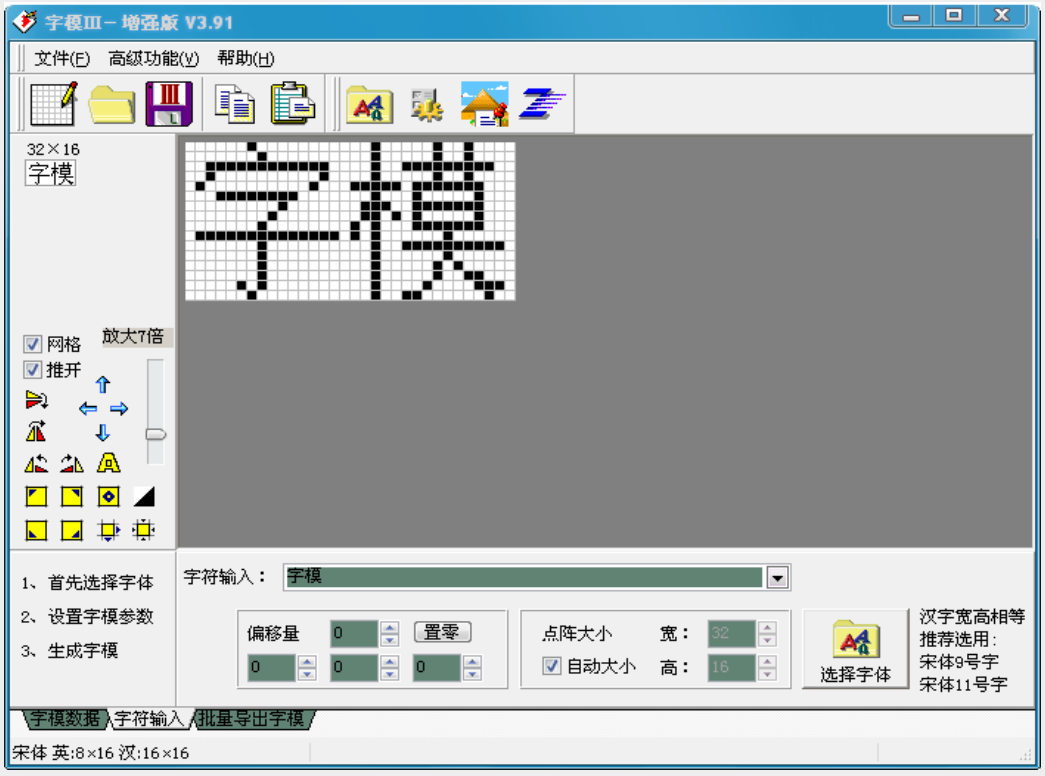
在这样的字模中，以两个字节表示一行像素点，16 行构成一个字模。如果使用 LCD 的画点函数，按位来扫描这些字模数据，把为 1 的位以黑色来显示(也可以使用其它颜色)，即可把整个点阵还原出来，显示在液晶屏上。

5.3 制作字模

我们采用“字模 III-增强版 v3.91” 

软件来制作中文字库，步骤如下：

1. 打开字模软件

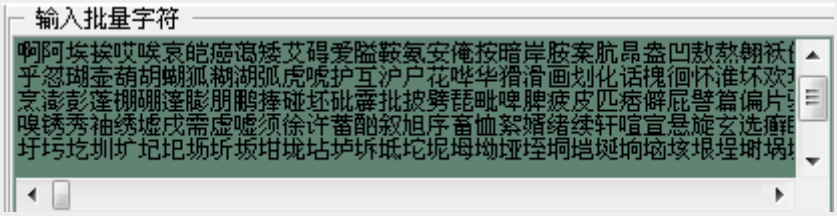


2. 点击“自动批量生成字库”按钮选项  。

软件界面左下角将出现一下几个按钮选项：

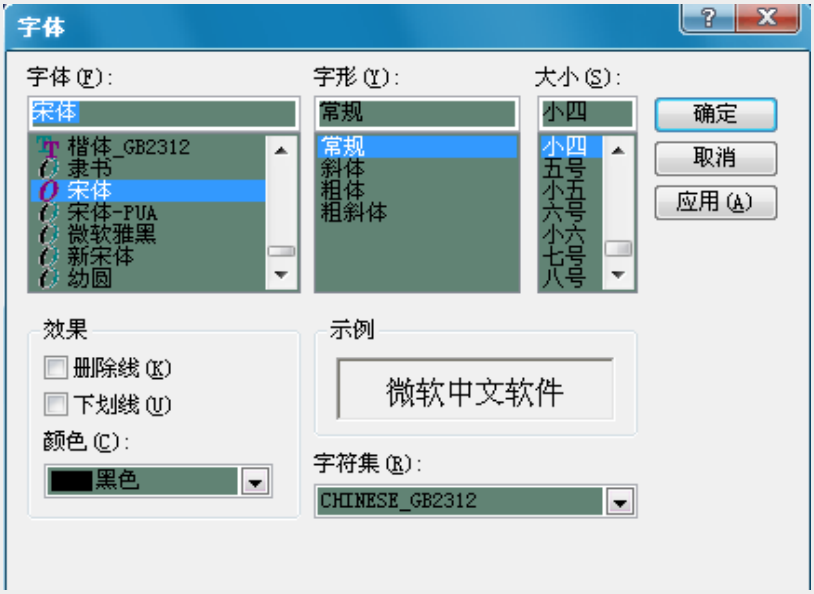
3. 点击选择“二级汉字库”按钮。

在“输入批量字符”框里面将会列出二级汉字的所有汉字，其中共收录了6768个汉字字符，非特殊情况下都能够满足大家的要求，如图：

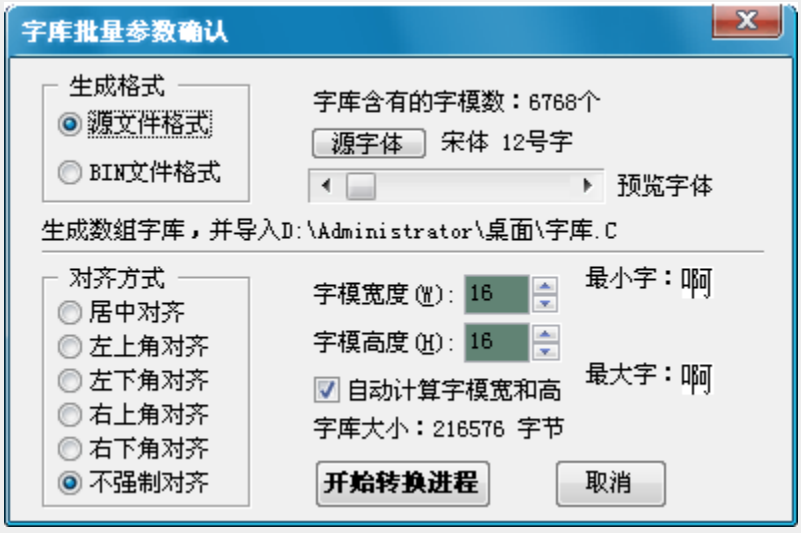


4. 点击“字库智能生成”按钮 字库智能生成 ，弹出“字库批量参数确认”对话框。

我们在“源字体”选项里面做如下设置，需要注意的是大小问题，因为我们本次的设计目标是实现16*16的汉字，所以在此选择'小四'字体。



设置好之后如下：

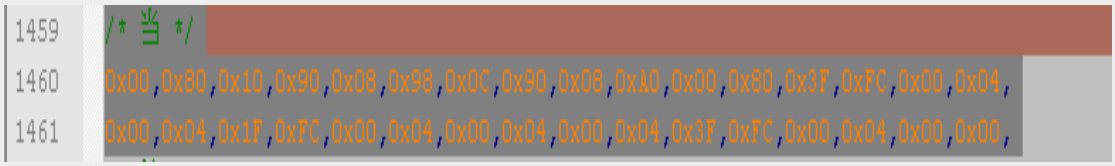


5. 点击“开始转换进程”按钮 

.就会在安装目录下或者你设置好的目录下生成.c 后缀的字库文件。

6. 对于 LCD 显示来说，只要能够在指定的位置描写制定颜色的点，那么就能够很好地根据汉字字模信息来描写汉字。在此， 为了能够更好的清楚字模的取向和高低位的排列顺序，我们可以现先在 pc 测试我们刚才制作好的库文件。

   在这里我们取“当”字符的数据来测试。

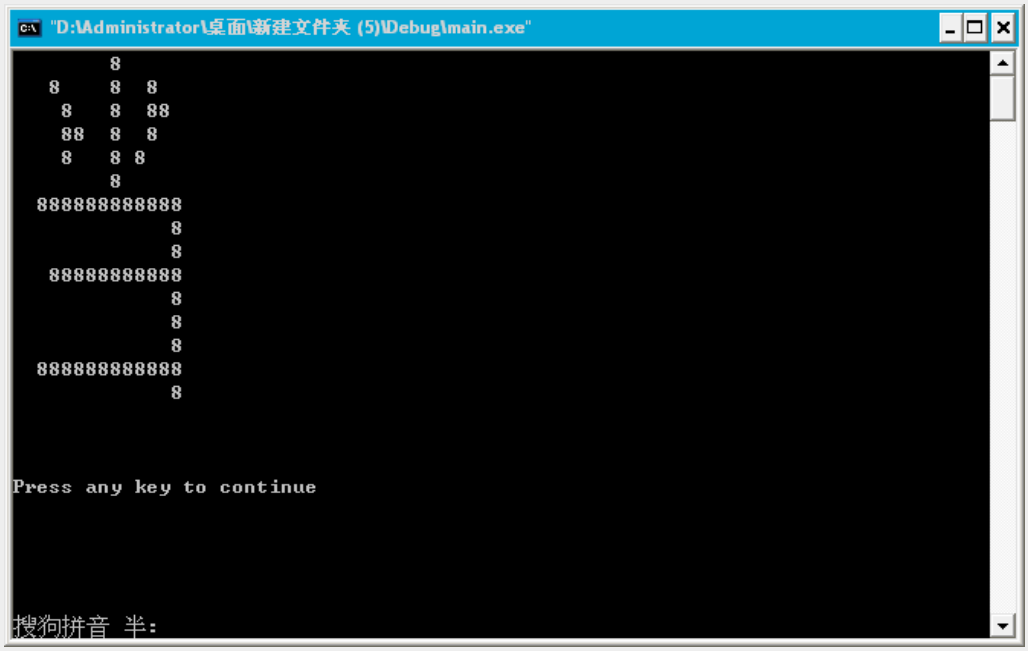


VC6.0 测试源码如下，该代码实现了把字模中为 1 的点都用数字“8”来表示：

```
#include <stdio.h>

unsigned char cc[] =
{/*"当"字符*/
0x00,0x80,0x10,0x90,0x08,0x98,0x0C,0x90,0x08,0xA0,0x00,0x80,0x3F,0xFC,
0x00,0x04,
0x00,0x04,0x1F,0xFC,0x00,0x04,0x00,0x04,0x00,0x04,0x3F,0xFC,0x00,0x04,
0x00,0x00
};

void main()
{
    int i,j;
    unsigned char kk;
    for ( i=0;  i<16; i++)
```

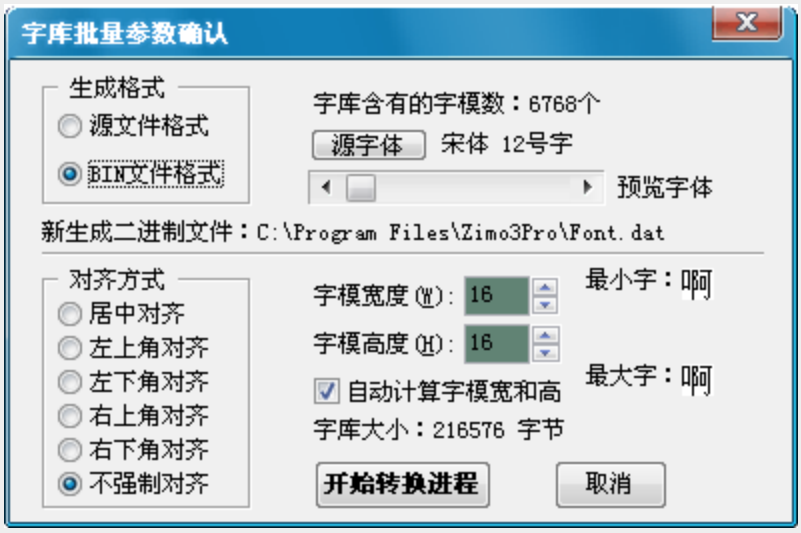
```
14.     {
15.         for(j=0; j<8; j++)
16.         {
17.             kk = cc[2*i] << j ;   //左移 J 位
18.
19.             if( kk & 0x80)   //如果最高位为 1
20.             {
21.                     printf("8");
22.             }
23.             else
24.             {
25.                     printf(" ");
26.             }
27.         }
28.
29.         for(j=0; j<8; j++)
30.         {
31.
32.             kk = cc[2*i+1] << j ;   //左移 J 位
33.
34.             if( kk & 0x80)        //如果最高位为 1
35.             {
36.                     printf("8");
37.             }
38.             else
39.             {
40.                     printf(" ");
41.             }
42.
43.         }
44.
45.         printf("\n");
46.
47.     }
48.     printf("\n\n");
49.
50.        }
51.
52.
```

测试结果如下：

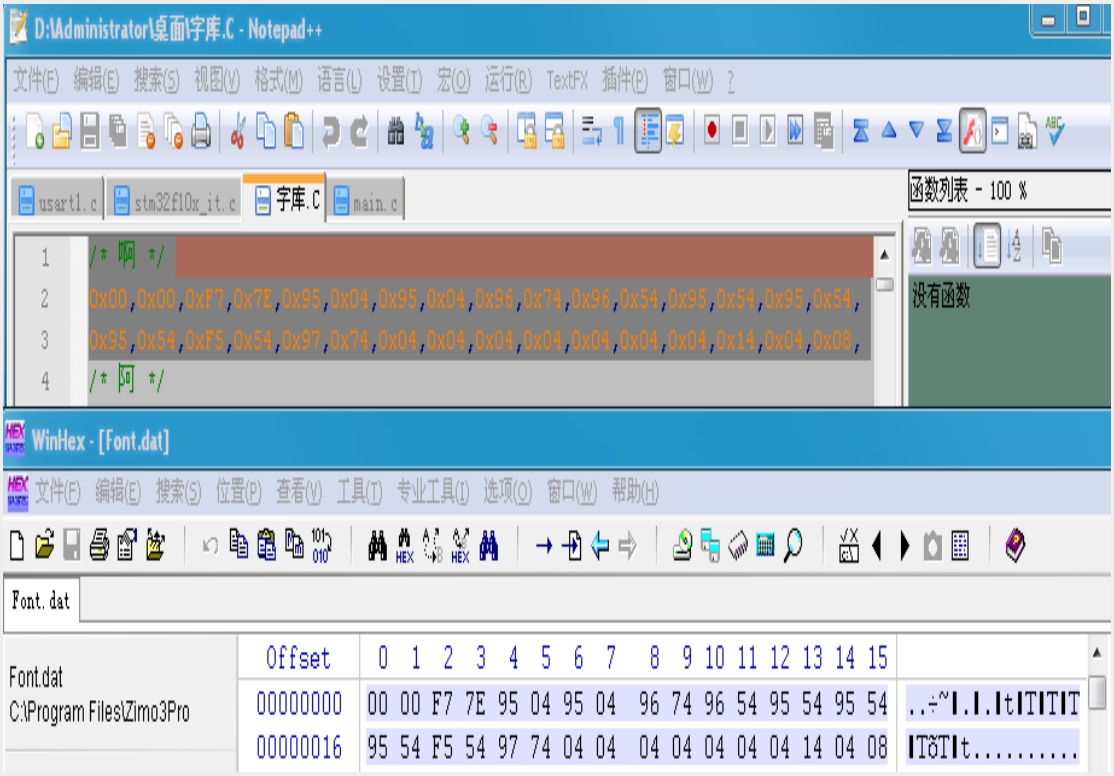


看到以上的测试结果，相信大家对汉字的取模方向和高低位的排列顺序有了比较直观的了解。

7. 回到“字模 III-增强版 v3.91”软件，采用与之前同样的方式生成 bin 格式的字库文件(即“生成格式”选项设置为“bin 文件格式”)。



在软件安装目录下会生成 Font.dat 文件,我们用“WinHex”软件查看他的具体内容，与刚才制作的.c 字库的文件内容是一致的，对比如下：



将生成的汉字字库拷贝到 SD 卡根目录下并重命名为“HZLIB.bin”。把该文件保存到 SD 卡中，STM32 芯片通过文件系统读取文件即可获得字库的数据。

5.4. BMP 图片格式

使用 LCD 屏来显示 BMP 图片，就如同播放 MP3 文件一样，首先要了解其文件的格式。

BMP 文件格式，又称为*位图*（Bitmap）或是 DIB(Device-Independent Device，设备无关位图)，是 Windows 系统中广泛使用的图像文件格式。BMP 文件保存了一幅图像中所有的像素。

BMP 格式可以保存单色位图、16 色或 256 色索引模式像素图、24 位真彩色图象，每种模式中单一像素点的大小分别为 1/8 字节，1/2 字节，1 字节和 32 字节。目前最常见的是 256 位色 BMP 和 24 位色 BMP。

BMP 文件格式还定义了像素保存的几种方法，包括不压缩、RLE 压缩等。*常见的 BMP 文件大多是不压缩的。*

Windows 所使用的 BMP 文件，在开始处有一个文件头，大小为 54 字节。保存了包括文件格式标识、颜色数、图像大小、压缩方式等信息，因为我们仅讨论 24 位色不压缩的 BMP，所以文件头中的信息基本不需要注意，只有“大小”这一项对我们比较有用。图像的宽度和高度都是一个 32 位整数，在文件中的地址分别为 0x0012 和 0x0016。54 个字节以后，如果是 16 色或 256 色 BMP，则还有一个颜色表，但在 24 位色 BMP 文件则没有，我们这里不考虑。接下来就是实际的像素数据了。因此总的来说 BMP 图片的优点是简单。

5.4.1 BMP 图片分析

下面来用 WinHex 软件(跟 UltraEdit 软件功能类似)来分析一下 BMP 图像的文件内容：

测试图片为 123.bmp，先用 ACDSee 软件打开，查看图像，其图像内容见图 0-1。

图 0-1 测试图片

可知，这幅图像的大小是 15*8，是 24 位真彩色 bmp 图像。

接着使用 WinHex 软件打开，可读取到该文件的原始数据见图 0-2。

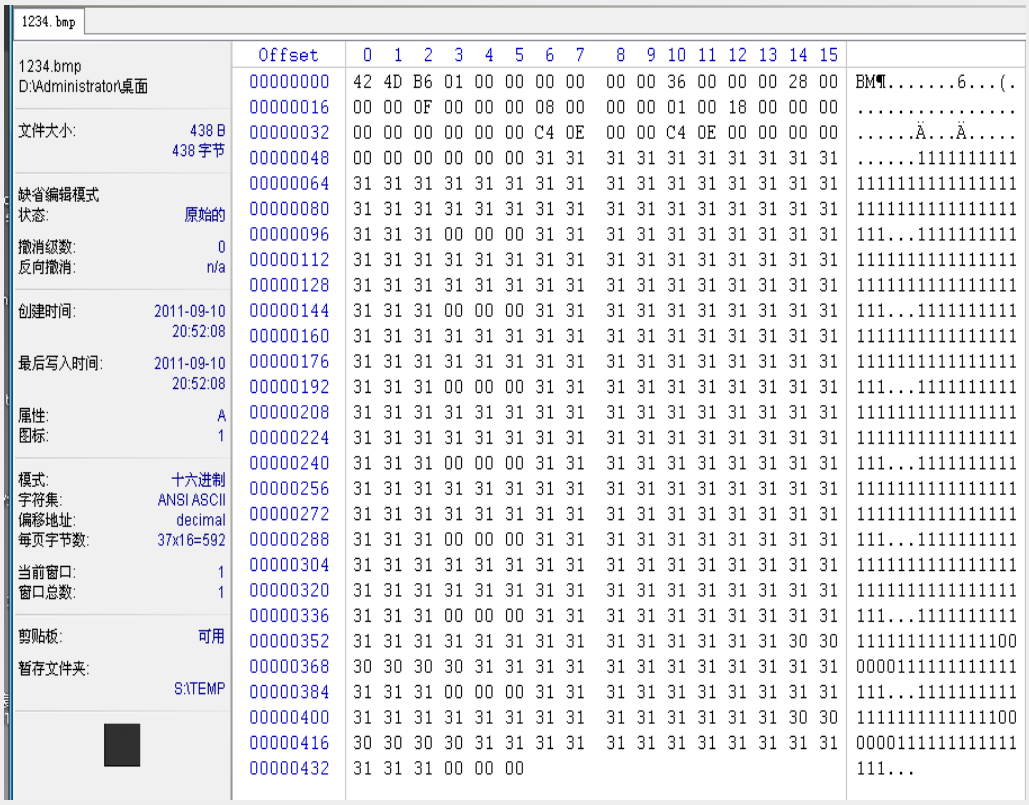


图 0-2 测试图片的原始数据

1. 文件头部信息部分(前面 54 字节)

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00000000	42	4D	B6	01	00	00	00	00	00	00	36	00	00	00	28	00
00000016	00	00	0F	00	00	00	08	00	00	00	01	00	18	00	00	00
00000032	00	00	00	00	00	00	C4	0E	00	00	C4	0E	00	00	00	00
00000048	00	00	00	00	00	00	31	31	31	31	31	31	31	31	31	31

图 0-3 文件头部信息

图 0-3，阴影部分是文件头部信息。它可以分为两块：*BMP 文件头*和*位图信息头*。

0~1 字节

BMP 文件的 0 和 1 字节用于*表示文件的类型*。如果是位图文件类型，必须分别为 0x42 和 0x4D ，0x424D=’BM’。

3~14 字节

3 到 14 字节的意义可以用一个结构体来描述。如下：

```
1.  typedef struct tagBITMAPFILEHEADER
2.  {
3.      //attention: sizeof(DWORD)=4   sizeof(WORD)=2
4.      DWORD bfSize;       //文件大小
5.      WORD bfReserved1;   //保留字，不考虑
6.      WORD bfReserved2;   //保留字，同上
7.      DWORD bfOffBits;    //实际位图数据的偏移字节数，即前三个部分长度之和
8.  } BITMAPFILEHEADER,tagBITMAPFILEHEADER;
```

14~53 字节

头部信息剩下的部分就是*位图信息头*，14 到 53 字节内容的意义如下：

```
1.  typedef struct tagBITMAPINFOHEADER
2.  {
3.    //attention: sizeof(DWORD)=4   sizeof(WORD)=2
4.  DWORD biSize;           //指定此结构体的长度，为 40
5.  LONG biWidth;           //位图宽，说明本图的宽度，以像素为单位
```

```
6.  LONG biHeight;          //位图高，指明本图的高度，像素为单位
7.  WORD biPlanes;          //平面数，为 1
8.  WORD biBitCount;        //采用颜色位数，可以是 1，2，4，8，16，24 新的可以是 32
9.  DWORD biCompression;     //压缩方式，可以是 0，1，2，其中 0 表示不压缩
10. DWORD biSizeImage;       //实际位图数据占用的字节数
11. LONG biXPelsPerMeter;    //X 方向分辨率
12. LONG biYPelsPerMeter;    //Y 方向分辨率
13. DWORD biClrUsed;         //使用的颜色数，如果为 0，则表示默认值(2^颜色位数)
14. DWORD biClrImportant;    //重要颜色数，如果为 0，则表示所有颜色都是重要的
15.
16. } BITMAPINFOHEADER,tagBITMAPINFOHEADER;
```

由上述分析与 WinHex 软件的分析内容结合得到该图片的信息如下：

文件大小:438

保留字:0

保留字:0

实际位图数据的偏移字节数:54

位图信息头:

结构体的长度:40

位图宽:15

位图高:8

biPlanes 平面数:1

biBitCount 采用颜色位数:24

压缩方式:0

biSizeImage 实际位图数据占用的字节数:0

X 方向分辨率:3780

Y 方向分辨率:3780

使用的颜色数:0

重要颜色数:0

2. 图像像素数据部分(如果是 24 位真彩色,则 54 字节之后就是像素部分):

```
00000048  00 00 00 00 00 00 31 31  31 31 31 31 31 31 31 31
00000064  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000080  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000096  31 31 31 00 00 00 31 31  31 31 31 31 31 31 31 31
00000112  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000128  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000144  31 31 31 00 00 00 31 31  31 31 31 31 31 31 31 31
00000160  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000176  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000192  31 31 31 00 00 00 31 31  31 31 31 31 31 31 31 31
00000208  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000224  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000240  31 31 31 00 00 00 31 31  31 31 31 31 31 31 31 31
00000256  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000272  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000288  31 31 31 00 00 00 31 31  31 31 31 31 31 31 31 31
00000304  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000320  31 31 31 31 31 31 31 31  31 31 31 31 31 31 31 31
00000336  31 31 31 00 00 00 31 31  31 31 31 31 31 31 31 31
00000352  31 31 31 31 31 31 31 31  31 31 31 31 31 31 30 30
00000368  30 30 30 30 31 31 31 31  31 31 31 31 31 31 31 31
00000384  31 31 31 00 00 00 31 31  31 31 31 31 31 31 31 31
00000400  31 31 31 31 31 31 31 31  31 31 31 31 31 31 30 30
00000416  30 30 30 30 31 31 31 31  31 31 31 31 31 31 31 31
00000432  31 31 31 00 00 00
```

图 0-4 图像像素数据

有些读者可能已经发现，*在像素部分夹杂着一些值为 0 的数据信息*，如下图灰色部分所示：

```
00000096  31 31 31 00 00 00 31 31  31 31 31 31 31 31 31 31
```

本例子中整张图片都是灰色的，为什么会有 0 像素的出现呢？

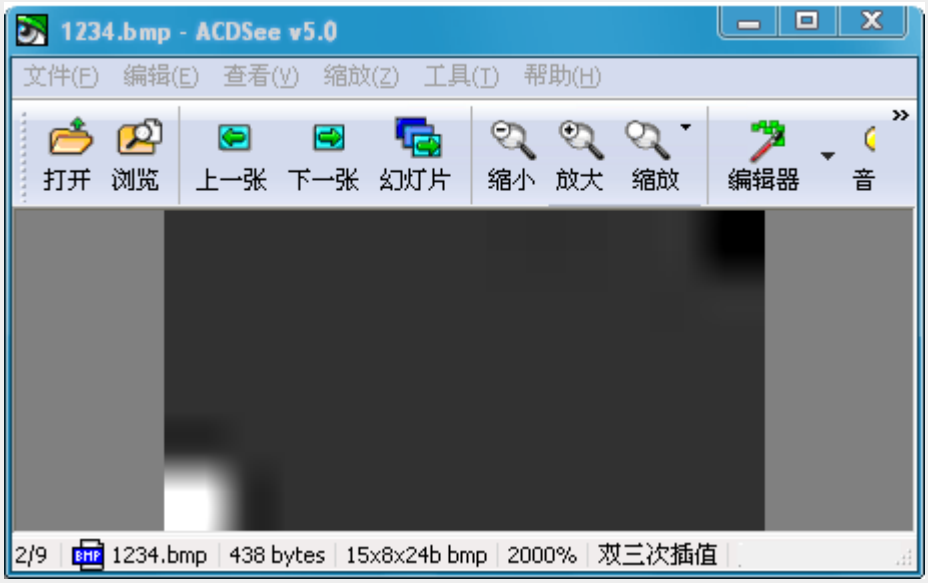
对齐的数据，更容易被操作系统或者硬件调入 cache，使得 cache 的命中率提高，从而提高访问效率。也就是基于性能上得考虑，所以 Windows 规定一个扫描行所占的字节数必须是 4 的倍数(即以 long 为单位)，不足的以 0 填充。由前面位图信息头的分析可知，*位图宽为 15 个像素点*，由于是 24 位图，每个像素点由 3 个字节构成。 因此，未补充字节前字节数为 15*3 等于 45 字节，要到 4 的倍数，必须向上取 4 的倍数即 48，所以就有了不上 3 个字节 0 的结果。

因此，整个图像文件的大小为：54+ （15*3 + 3）*8 等于 438 字节，和前面所得到的信息*文件大小为 438* 是一致的。

另外一点需要注意的是显示图像的顺序是 *由下到上，由左到右*。即像素数据部分的第一像素数据是我们见到的图像的左下角的像素的数据；而像素数据的最后一个有效数据是我们见到的图像的右上角的像素的数据。

下面我们修改图片来验证一下：

我们将左下角画上白色，右上角画上黑色，图片放大之后如下



图片数据分析如下：

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00000000	42	4D	B6	01	00	00	00	00	00	00	36	00	00	00	28	00
00000016	00	00	0F	00	00	00	08	00	00	00	01	00	18	00	00	00
00000032	00	00	80	01	00	00	C4	0E	00	00	C4	0E	00	00	00	00
00000048	00	00	00	00	00	00	FF	FF	FF	31	31	31	31	31	31	31
00000064	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000080	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000096	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000112	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000128	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000144	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000160	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000176	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000192	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000208	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000224	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000240	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000256	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000272	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000288	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000304	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000320	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000336	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000352	31	31	31	31	31	31	31	31	31	31	31	31	31	31	30	30
00000368	30	30	30	30	31	31	31	31	31	31	31	31	31	00	00	00
00000384	00	00	00	00	00	00	31	31	31	31	31	31	31	31	31	31
00000400	31	31	31	31	31	31	31	31	31	31	31	31	31	31	30	30
00000416	30	30	30	30	31	31	31	31	31	31	31	31	31	00	00	00
00000432	00	00	00	00	00	00										

我们可以看到像素部分开始部分是白色像素(灰色部分)，对应我们图像的左下角：

00000048	00	00	00	00	00	00	FF	FF	FF	31	31	31	31	31	31	31

后尾部分是黑色像素（灰色部分）对应我们图像的右上角：

00000416	30	30	30	30	31	31	31	31	31	31	31	31	31	00	00	00
00000432	00	00	00	00	00	00										

对 BMP 图像文件内容有了具体的了解之后就可以开始编写我们的应用程序，根据图片的头部信息，合理地读出其中的像素部分，把读出的像素点数据送到 LCD 屏，就可以显示出该图片了。

5.5 显示中英文及 BMP 图片实验

5.5.1 实验描述及工程文件清单

实验描述	使用软件制作自定义类型的字库，然之后将字库放入 SD 卡中，并且在 SD 卡中放入三张 BMP 图片。最后调用截屏函数截取 LCD 背景并保存为 BMP 图片。
硬件连接	本实验的硬件连接包括 MicroSD 卡控制信号、TFT 控制信号线及触摸屏 TSC2046 控制线，这些连接与前面章节的一样，可参照前面的电路图。
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c *FWlib/misc.c* *FWlib/stm32f10x_rcc.c* *Fwlib/stm32f10x_systick.c* *FWlib/stm32f10x_exti.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_sdio.c* *FWlib/stm32f10x_dma.c* *FWlib/stm32f10x_usart.c* *FWlib/stm32f10x_fsmc.c*
用户编写的文件	USER/main.c USER/stm32f10x_it.c *USER/systick.c* *USER/usart1.c* *USER/lcd.c* *USER/ff.c*

	USER/ sdio_sdcard.c USER/lcd_botton.c USER/Sd_bmp.c USER/sd_fs_app.c
文件系统文件	ff9/diskio.c ff9/ff.c ff9/cc936.c

5.5.2 配置工程环境

本实验需要制作字库，其文件名为：*HZLIB.bin*，三个 BMP 图片文件，文件名为：*pic1.bmp*、*pic2.bmp*、*pic3.bmp*，把这四个文件保存到 SD 卡中，再把该 SD 卡插入开发板的 SD 卡接口。

本实验中要把旧文件：*systick.c*、*usart1.c*、*lcd.c*、*ff.c*、*sdio_sdcard.c* 、*lcd_botton.c* 文件添加进新工程，新建 *Sd_bmp.c*、*sd_fs_app.c* 文件，分别用于编写 BMP 文件相关的函数和字模获取函数。

5.5.3 main 文件

从 main 函数看起，它调用了很多函数，用于显示 BMP 图和沿各种方向排列的文字。

```
1.  int main(void)
2.  {
3.
4.     /* USART1 config */
5.      USART1_Config();
6.      SysTick_Init();
7.      LCD_Init();                          /* LCD 初始化*/
8.      sd_fs_init();
9.
10.     /*显示图像*/
11.     Lcd_show_bmp(0, 0,"/pic3.bmp");
12.     Lcd_show_bmp( 0,0,"/pic2.bmp");
13.     Lcd_show_bmp( 0,0,"/pic1.bmp");
14.
15.     /*横屏显示*/
16.     LCD_Str_O(20, 10, "lCD_DEMO",0);
17.     LCD_Str_CH(20,30,"阿莫论坛野火专区",0,0xffff);
18.     LCD_Str_CH_O(20,50,"阿莫论坛野火专区",0);
```

```
19.     LCD_Num_6x12_O(20, 70, 65535, BLACK);
20.     LCD_Str_6x12_O(20, 90,"LOVE STM32", BLACK);
21.
22.     /*竖屏显示*/
23.     LCD_Str_O_P(300, 10, "Runing", 0);
24.     LCD_Str_CH_P(280,10,"阿莫论坛野火专区欢迎你",0xff,0xffff);
25.     LCD_Str_CH_O_P(260,10,"阿莫论坛野火专区",0);
26.     LCD_Str_6x12_O_P(240, 10,"LOVE STM32", 0);
27.     LCD_Str_ENCH_O_P(220,10,"欢迎使用野火 stm32 开发板",0);
28.
29.     /*截图*/
30.     LCD_Str_CH(20,150,"正在截图",0,0xffff);
31.     Screen_shot(0, 0, 240, 320, "/myScreen");
32.     LCD_Str_CH(20,150,"截图完成",0,0xffff);
33.
34.
35.   while (1)
36.   {}
37. }
```

5.5.4 显示汉字

这里先说汉字字符的显示，第 17 行：`LCD_Str_CH(20,30,"阿莫论坛野火专区",0,0xffff);`

该函数功能是显示汉字字符串，源码如下：

```
1.  /*****************************************************************
2.   * 函数名：LCD_Str_CH
3.   * 描述   ：在指定坐标处显示 16*16 大小的指定颜色汉字字符串
4.   * 输入   :     - x: 显示位置横向坐标
5.   *            - y: 显示位置纵向坐标
6.   *            - str: 显示的中文字符串
7.   *            - Color: 字符颜色
8.   *            - bkColor: 背景颜色
9.   * 输出   ：无
10.  * 举例   :     LCD_Str_CH(0,0,"阿莫论坛野火专区",0,0xffff);
11.             LCD_Str_CH(50,100,"阿莫论坛野火专区",0,0xffff);
12.             LCD_Str_CH(320-16*8,240-16,"阿莫论坛野火专区
    ",0,0xffff);
13.  * 注意   :     字符串显示方向为横向 已测试
14. *****************************************************************/
15. void LCD_Str_CH(u16 x,u16 y,const u8 *str,u16 Color,u16 bkColor)
16. {
17.
18.     Set_direction(0);
19.     while(*str != '\0')
20.     {
21.             if(x>(320-16))
22.             {
23.                   /*换行*/
24.                 x =0;
25.                 y +=16;
26.
27.             }
28.             if(y >(240-16))
29.             {
30.                   /*重新归零*/
31.                   y =0;
```

```
                x =0;
            }
          LCD_Char_CH(x,y,str,Color,bkColor);
        str += 2 ;
        x += 16 ;
    }
}
```

该函数其实没做到什么工作，对超出屏幕范围的显示坐标进行换行处理，并把字符串中的汉字一个一个提取出来调用单字符显示函数 *LCD_Char_CH()*显示出来，*LCD_Char_CH()*函数的源码如下：

```
/*********************************************
 * 函数名：LCD_Char_CH
 * 描述   ：显示单个汉字字符
 * 输入   :     x: 0~(319-16)
 *            y: 0~(239-16)
 *            str: 中文字符串首址
 *            Color: 字符颜色
 *            bkColor: 背景颜色
 * 输出   ：无
 * 举例   :      LCD_Char_CH(200,100,"好",0,0);
 * 注意     ：如果输入大于 1 的汉字字符串，显示将会截断，只显示最前面一个汉字
*********************************************************/
void LCD_Char_CH(u16 x,u16 y,const u8 *str,u16 Color,u16 bkColor)
{

#ifndef NO_CHNISEST_DISPLAY          /*如果汉字显示功能没有关闭*/
    u8 i,j;
    u8 buffer[32];
    u16 tmp_char=0;


   GetGBKCode_from_sd(buffer,str);  /* 取字模数据 */

    for (i=0;i<16;i++)
    {
        tmp_char=buffer[i*2];
        tmp_char=(tmp_char<<8);
        tmp_char|=buffer[2*i+1];
        for (j=0;j<16;j++)
        {
            if ( (tmp_char >> 15-j) & 0x01 == 0x01)
            {
                LCD_ColorPoint(x+j,y+i,Color);
            }
            else
            {
                LCD_ColorPoint(x+j,y+i,bkColor);
            }
        }
    }

#endif
}
```

函数中的条件编译*#ifndef NO_CHNISEST_DISPLAY*，是用于开关汉字显示功能的，若定义了*NO_CHNISEST_DISPLAY*，则本函数为空，关闭了显示汉字的功能。

在*LCD_Char_CH()*这个函数中，首先调用*GetGBKCode_from_sd()*从 SD 卡中读出我们需要显示在 LCD 上的指定汉字的字模数据。

接着在 22~40 行的代码就根据字模数据来描写，把字模中为 1 的数据位，在 LCD 屏中的像素点中使用画点函数*LCD_ColorPoint()*显示特定的颜色。思路和前面 VC 测试部分用数字“8”来显示“当”字是一样的。

5.5.4.1 查找字模

读者现在可能在想，字库里面保存着大量的汉字字幕信息，现在输入*GetGBKCode_from_sd(buffer,str)*就能够拷贝出这个字符的字模数据，是怎样定位字模信息所在的位置的呢？换句话说，假如现在要显示“吾”字，是怎样根据这个字来确定“吾”字符在字库中的保存位置的呢？其实这里面有一定的映射关系，那就是接下来要说的汉字“区码”和“位码”。

在前面生成的*HZLIB.bin*文件，实际是按国标 GB2312 生成的二级汉字库。在国标 GB2312—80 中规定，所有的国标汉字及符号在字库中的存储形式是：分配在一个 94 行、94 列的阵列中，阵列的每一行称为一个“区”，共有 01 区到 94 区；每一列称为一个“位”，共有 01 位到 94 位，阵列中的每一个汉字和符号所在的区号和位号组合在一起形成的四个阿拉伯数字就是它们的“区位码”。区位码的前两位是它的区号，后两位是它的位号。*我们生成的汉字库就是这样按区位码排列的阵列，通过区位码，就能查找出该字的字模。*

*汉字的机内码*是指在计算机中表示一个汉字的编码。为避免与 ASCII 码混淆。用机内码的两个字节表示一个汉字，这两个字节分别称为高位字节和低位字节。

高位字节 = 区码 + 20H + 80H(或区码 + A0H)

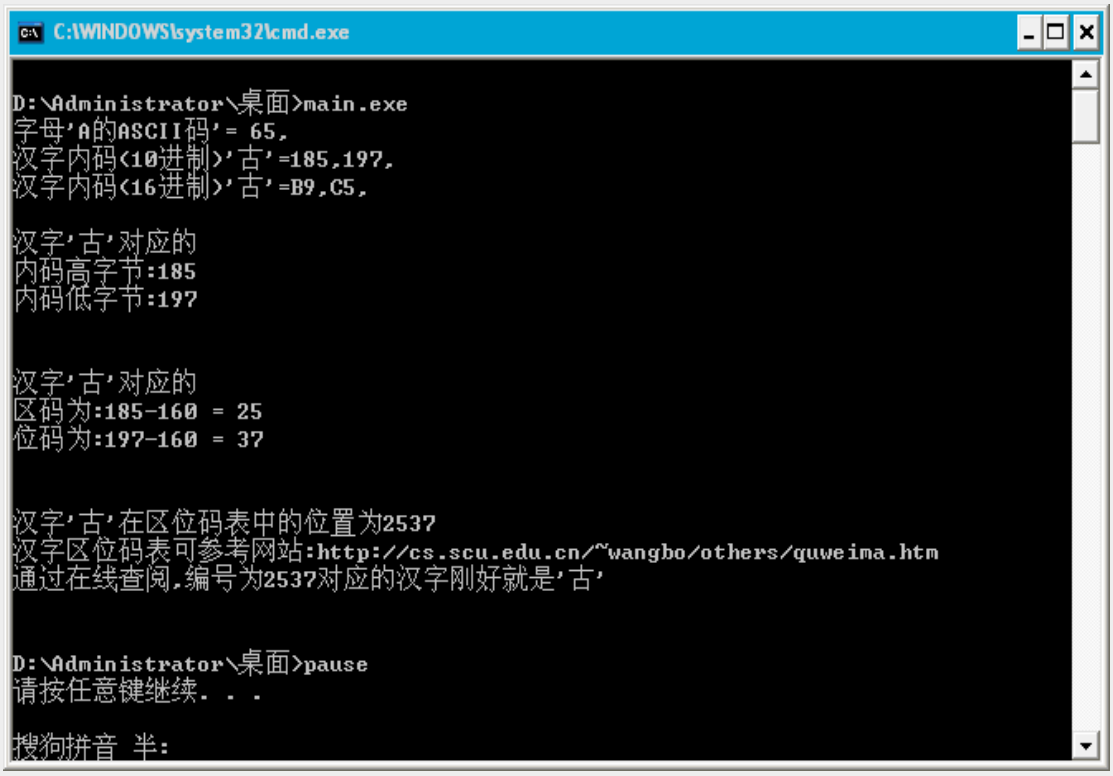
低位字节 = 位码 + 20H + 80H(或位码 + AOH)

因此，我们就可以通过汉字的机内码，运算得出汉字在字库中的区位码，由区位码查找出该汉字的字模。

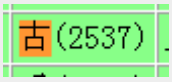
下面以 vc6.0 的测试源码来说明机内码、区位码的关系：

```
#include <stdio.h>
void main ()
{
    unsigned char * s , * e = "A" , * c = "古" ;
    unsigned char high_byte,lower_byte; //内码高字节，内码低字节
    printf ( "字母'%s 的 ASCII 码'=",e ) ;
    s = e ;

    while ( * s != 0 )                    //C 的字符串以 0 为结束符*
    {
        printf ( "%3d," , *s ) ;
        s ++ ;
    }
    printf ( "\n 汉字内码(10 进制)'%s'=",c ) ;

    s = c ;
    while ( *s != 0 )
    {
        printf ( "%3d," , * s );
        s ++ ;
    }

        printf ( "\n 汉字内码(16 进制)'%s'=",c ) ;

    s = c ;
    while ( *s != 0 )
    {
        printf ( "%0X," , * s );
        s ++ ;
    }


    s = c ;
    high_byte = *s;

    s ++ ;
    lower_byte = *s;

    printf("\n\n 汉字'%s'对应的\n 内码高字节:%d\n 内码低字节:%d\n",c,high_byte,lower_byte);
    printf("\n\n 汉字'%s'对应的\n 区码为:%d-160 = %d\n 位码为:%d-160 = %d\n",c,high_byte,high_byte-160,lower_byte,lower_byte-160);

    printf("\n\n 汉字'%s'在区位码表中的位置为%d%d\n",c,high_byte-160,lower_byte-160);
    printf("汉字区位码表可参考网站:http://cs.scu.edu.cn/~wangbo/others/quweima.htm\n");
    printf("通过在线查阅,编号为%d%d 对应的汉字刚好就是'%s'\n\n",high_byte-160,lower_byte-160,c);

}
```

测试结果如下：



打开汉字区位码表在线查询网站：*http://www.jscj.com/index/gb2312.php*

查询“古”汉字的区位码刚好如计算所得，为 

。

上面的测试结果说明了每一个汉字的内码具体作用。回到本实验工程中获取字模函数 *GetGBKCode_from_sd()*中，它的具体定义如下：

```
1.  /**********************************************************
2.   * 函数名：GetGBKCode_from_sd
3.   * 描述   ：从 sd 卡上的字库文件中拷贝指定汉字的字模数据
4.   * 输入   ：pBuffer---数据保存地址
5.   *           c--汉字字符低字节码
6.   * 输出   ：     0                    (成功)
7.   *               -1                   (失败)
8.   **********************************************************/
9.
10. int GetGBKCode_from_sd(unsigned char* pBuffer,unsigned char * c)
11. {
12.     unsigned char High8bit,Low8bit;
13.     unsigned int pos;
14.     High8bit=*c;
15.     Low8bit=*(c+1);
16.
17.     pos = ((High8bit-0xb0)*94+Low8bit-0xa1)*2*16 ;
18.     f_mount(0, &myfs[0]);
19.     myres = f_open(&myfsrc , "0:/HZLIB.bin", FA_OPEN_EXISTING | FA_REA
    D);
20.
21.     if ( myres == FR_OK )
22.     {
23.         f_lseek (&myfsrc, pos);                          //制定读取位置
```

```
24.         myres = f_read( &myfsrc, pBuffer, 32, &mybr ); //16*16 大小的汉字其字模占用 16*2 个字节
25.         f_close(&myfsrc);
26.
27.         return 0;
28.     }
29.
30.     else
31.         return -1;
32.
33. }
```

第 17 行中的： *pos = ((High8bit-0xb0)*94+Low8bit-0xa1)*2*16* 语句就是根据约定的映射关系由汉字内码求得该汉字字模在字库中的存放起始位置。之后就到指定的位置去拷贝字模数据就可以了。

以上就是利用 SD 卡字库实现 LCD 显示汉字的具体流程。对于 ASCII 码(包括英文字符)的显示，原理实际上也是一样的，由于 ASCII 码占用空间较少，所以我们直接把它的字模数据以数组的形式存储在代码中，这些数组在文件 *ascii.h* 和 *asc_font.h* 中定义。

5.6 实现 SD 卡 BMP 图像的读取与保存

5.6.1 显示 BMP 图

再回到前面的 main 函数，其中调用了一个 BMP 图片显示函数 *Lcd_show_bmp()*，其定义如下：

```
1.  /*****************************************************
2.   * 函数名: Lcd_show_bmp
3.   * 描述   : LCD 显示 RGB888 位图图片
4.   * 输入   : x                    --显示横坐标(0-319)
5.              y                      --显示纵坐标(0-239)
6.   *                pic_name        --图片名称
7.   * 输出   : 无
8.   * 举例   : Lcd_show_bmp(0, 0,"/test.bmp");
9.   * 注意   : 图片为 24 为真彩色位图图片
10.                       图片宽度不能超过 320
11.                       图片在 LCD 上的粘贴范围为:纵向:  [x,x+图像高度]    横向 [Y, Y+图像宽度]
12.                       当图片为 320*240 时--建议 X 输入 0   y 输入 0
13.  *****************************************************/
14. void Lcd_show_bmp(unsigned short int x, unsigned short int y,unsigned char *pic_name)
15. {
16.     int i, j, k;
17.     int width, height, l_width;
18.
19.     BYTE red,green,blue;
```

```
20.     BITMAPFILEHEADER bitHead;
21.     BITMAPINFOHEADER bitInfoHead;
22.     WORD fileType;
23.
24.     unsigned int read_num;
25.     unsigned char tmp_name[20];
26.     sprintf((char*)tmp_name,"0:%s",pic_name);
27.     f_mount(0, &bmpfs[0]);
28.
29.     BMP_DEBUG_PRINTF("file mount ok \r\n");    //使用串口输出调试信息
30.
31.     bmpres = f_open( &bmpfsrc , (char *)tmp_name, FA_OPEN_EXISTING | F
    A_READ);
32.     Set_direction(0);
33.
34.     if(bmpres == FR_OK)
35.     {
36.         BMP_DEBUG_PRINTF("Open file success\r\n");
37.
38.         //读取位图文件头信息
39.         f_read(&bmpfsrc,&fileType,sizeof(WORD),&read_num);
40.
41.         if(fileType != 0x4d42)
42.         {
43.             BMP_DEBUG_PRINTF("file is not .bmp file!\r\n");
44.             return;
45.         }
46.         else
47.         {
48.             BMP_DEBUG_PRINTF("Ok this is .bmp file\r\n");
49.         }
50.
51.         f_read(&bmpfsrc,&bitHead,sizeof(tagBITMAPFILEHEADER),&read_num
    );
52.
53.         showBmpHead(&bitHead);
54.         BMP_DEBUG_PRINTF("\r\n");
55.
56.         //读取位图信息头信息
57.         f_read(&bmpfsrc,&bitInfoHead,sizeof(BITMAPINFOHEADER),&read_nu
    m);
58.         showBmpInforHead(&bitInfoHead);
59.         BMP_DEBUG_PRINTF("\r\n");
60.     }
61.     else
62.     {
63.         BMP_DEBUG_PRINTF("file open fail!\r\n");
64.         return;
65.     }
66.
67.     width = bitInfoHead.biWidth;
68.     height = bitInfoHead.biHeight;
69.
70.     l_width = WIDTHBYTES(width* bitInfoHead.biBitCount);         //计算
    位图的实际宽度并确保它为 32 的倍数
71.
72.     if(l_width>960)
73.     {
74.         BMP_DEBUG_PRINTF("\nSORRY, PIC IS TOO BIG (<=320)\n");
75.         return;
76.     }
77.
78.     if(bitInfoHead.biBitCount>=24)
79.     {
80.
81.         bmp_lcd(x,240-y-height,width, height);     //LCD 参数相关设置
82.
```

```
        for(i=0;i<height+1; i++)
        {

            for(j=0; j<l_width; j++)  //将一行数据全部读入
            {

                f_read(&bmpfsrc,pColorData+j,1,&read_num);
            }

            for(j=0;j<width;j++)//一行有效信息
            {
                k = j*3;
                                    //一行中第 k 个像素的起点
                red = pColorData[k+2];
                green = pColorData[k+1];
                blue =  pColorData[k];
                LCD_WR_Data(RGB24TORGB16(red,green,blue));       //写入
  LCD-GRAM
            }
              }
              bmp_lcd_reset();    //lcd 扫描方向复原
          }
          else
          {
      BMP_DEBUG_PRINTF("SORRY, THIS PIC IS NOT A 24BITS REAL COLOR");

              return ;
          }

          f_close(&bmpfsrc);
      }
```

该函数的主要工作流程是：读取头部信息确定宽度和高度并确定每一行后面具体需要读出的字节数(保证是 4 字节的倍数)----->读取一行像素点并显示-->读取下一行并显示--> .....直至读完所有行。

另外一点就是：*RGB24TORGB16*是个宏定义，因为图像数据是 RGB888 即 24 为真彩色，而我们的 LCD 是 RGB565 即 16 位色度的，所以我们需要按比例将 24 为真彩色压缩为 16 位。宏定义如下：

```
#define RGB24TORGB16(R,G,B) ((unsigned short int)((((R)>>3)<<11)|(((G)
>>2)<<5)| ((B)>>3)))
```

5.6.2 LCD 截图功能

为了实现截图功能，我们可以根据用户截屏范围的宽和高来构造 BMP 文件的信息头，并且*根据位图宽与四字节对齐的关系来补充需要的 0 大小字节*。在 main 函数中，我们调用了 *Screen_shot()*这个函数来截图。保存的图片是 24 位的真彩色。*Screen_shot()*的源码如下：

```
/*************************************************************
 * 函数名：Screen_shot
 * 描述   ：截取 LCD 指定位置  指定宽高的像素  保存为 24 位真彩色 bmp 格式图片
 * 输入   :    x                                    ---水平位置
 *                  y                                      ---竖直位置
 *                  Width                             ---水平宽度
 *                  Height                          ---竖直高度
 *                  filename                        ---文件名
 * 输出   :    0       ---成功
 *                  -1      ---失败
 *                8           ---文件已存在
 * 举例   : Screen_shot(0, 0, 320, 240, "/myScreen");-----全屏截图
 * 注意   : x 范围[0,319]  y 范围[0,239]  Width 范围[0,320-x]  Height 范围[0,240-y]
 *                  如果文件已存在,将直接返回
 *************************************************************/
int Screen_shot(unsigned short int x, unsigned short int y, unsigned short int Width, unsigned short int Height, unsigned char *filename)
{
    unsigned char header[54] =
    {
        0x42, 0x4d, 0, 0, 0, 0,
        0, 0, 0, 0, 54, 0,
        0, 0, 40,0, 0, 0,
        0, 0, 0, 0, 0, 0,
        0, 0, 1, 0, 24, 0,
        0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0,
        0, 0, 0, 0, 0,
        0, 0, 0, 0, 0,
        0, 0, 0
    };
    int i;
    int j;
    long file_size;
    long width;
    long height;
    unsigned short int tmp_rgb;
    unsigned char r,g,b;
    unsigned char tmp_name[30];
    unsigned int mybw;
    char kk[4]={0,0,0,0};


   // if(!(Width%4))
    //    file_size = (long)Width * (long)Height * 3 + 54;
    //else
    file_size = (long)Width * (long)Height * 3 + Height*(Width%4) + 54;      //宽*高 +补充的字节 + 头部信息

    header[2] = (unsigned char)(file_size &0x000000ff);
    header[3] = (file_size >> 8) & 0x000000ff;
    header[4] = (file_size >> 16) & 0x000000ff;
    header[5] = (file_size >> 24) & 0x000000ff;


    width=Width;
    header[18] = width & 0x000000ff;
    header[19] = (width >> 8) &0x000000ff;
    header[20] = (width >> 16) &0x000000ff;
    header[21] = (width >> 24) &0x000000ff;

    height = Height;
    header[22] = height &0x000000ff;
    header[23] = (height >> 8) &0x000000ff;
```

```
63.     header[24] = (height >> 16) &0x000000ff;
64.     header[25] = (height >> 24) &0x000000ff;
65.
66.     sprintf((char*)tmp_name,"0:%s.bmp",filename);
67.     f_mount(0, &bmpfs[0]);
68.
69.
70.     bmpres = f_open( &bmpfsrc , (char*)tmp_name, FA_CREATE_NEW | FA_WR
    ITE);
71.
72.      f_close(&bmpfsrc);    //新建文件之后要先关闭再打开才能写入
73.     bmpres = f_open( &bmpfsrc , (char*)tmp_name, FA_OPEN_EXISTING | FA
    _WRITE);
74.     if ( bmpres == FR_OK )
75.     {
76.         bmpres = f_write(&bmpfsrc, header,sizeof(unsigned char)*54, &m
    ybw);
77.         for(i=0;i<Height;i++)                          //高
78.         {
79.             if(!(Width%4))
80.             {
81.                 for(j=0;j<Width;j++)     //宽
82.                 {
83.                                          #ifdef HX8347
84.                     tmp_rgb = bmp4(j+y,Height-i+x);
85.                                          #else
86.                                          tmp_rgb = bmp4(Height-
    i+x,j+y);
87.                                          #endif
88.
89.                     r =  GETR_FROM_RGB16(tmp_rgb);
90.                     g =  GETG_FROM_RGB16(tmp_rgb);
91.                     b =  GETB_FROM_RGB16(tmp_rgb);
92.
93.                     bmpres = f_write(&bmpfsrc, &b,sizeof(unsigned char
    ), &mybw);
94.                     bmpres = f_write(&bmpfsrc, &g,sizeof(unsigned char
    ), &mybw);
95.                     bmpres = f_write(&bmpfsrc, &r,sizeof(unsigned char
    ), &mybw);
96.
97.                 }
98.
99.             }
100.                else
101.                {
102.                    for(j=0;j<Width;j++)
103.                    {
104.                                             #ifdef HX8347
105.                        tmp_rgb = bmp4(j+y,Height-i+x);
106.                                             #else
107.                                             tmp_rgb = bmp4(Height-
    i+x,j+y);
108.                                             #endif
109.
110.                        r =  GETR_FROM_RGB16(tmp_rgb);
111.                        g =  GETG_FROM_RGB16(tmp_rgb);
112.                        b =  GETB_FROM_RGB16(tmp_rgb);
113.
114.                        bmpres = f_write(&bmpfsrc, &b,sizeof(unsign
    ed char), &mybw);
115.                        bmpres = f_write(&bmpfsrc, &g,sizeof(unsign
    ed char), &mybw);
116.                        bmpres = f_write(&bmpfsrc, &r,sizeof(unsign
    ed char), &mybw);
117.
118.
```

```
119.                        }
120.
121.                        bmpres = f_write(&bmpfsrc, kk,sizeof(unsigned c
    har)*(Width%4), &mybw);
122.
123.
124.                    }
125.                }
126.
127.                f_close(&bmpfsrc);
128.                return 0;
129.            }
130.            else if ( bmpres == FR_EXIST )  //如果文件已经存在
131.            {
132.                f_close(&bmpfsrc);
133.                return FR_EXIST;                        //8
134.            }
135.
136.            else
137.            {   f_close(&bmpfsrc);
138.                return -1;
139.            }
140.        }
```

该函数中，调用了 *bmp4()*这个函数，该函数返回 LCD 上指定位置的像素信息。*GETG_FROM_RGB16（绿色）*、*GETB_FROM_RGB16（蓝色）*和 *GETR_FROM_RGB16 (红色)*都是宏定义，将 RGB565 即 16 位色度抽取出其中的 RGB 数据并分别将其线性映射为 8 位数据即映射为 RGB888 真彩色。宏定义的内容如下：

```
1. #define GETR_FROM_RGB16(RGB565)  ((unsigned char)(( ((unsigned short i
   nt )RGB565) >>11)<<3))           //返回 8 位 R
2. #define GETG_FROM_RGB16(RGB565)  ((unsigned char)(( ((unsigned short i
   nt )(RGB565 & 0x7ff)) >>5)<<2))  //返回 8 位 G
3. #define GETB_FROM_RGB16(RGB565)  ((unsigned char)(( ((unsigned short i
   nt )(RGB565 & 0x1f))<<3)))       //返回 8 位 B
```

利用 *Screen_shot()*函数，就实现了把屏幕的当前显示的图像转换为 BMP 文件的功能。

5.7 实验现象

把文件 *HZLIB.bin*、*pic1.bmp*、*pic2.bmp*、*pic3.bmp*，保存到 SD 卡中，再把该 SD 卡插入开发板的 SD 卡接口（也可直接把工程下的 SD 字库备份文件夹下的内容复制到 SD 卡的根目录），然后将野火 STM32 开发板供电

(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，接上液晶屏，将编译好的程序下载到开发板。

程序运行之后截图保存为“myScreen.bmp”如下：

(^ ^ 截图保存图片功能+摄像头模块 就构成了一个完整的照相机啦！！)

6、UsbDevice（模拟 U 盘）

6.1 实验描述及工程文件清单

实验描述	这是一个 USB Mass Storage 实验，用 USB 线连接 PC 机与开发板，在电脑上就可以像操作普通 U 盘那样来操作开发板中的 MicroSD 卡，并在超级终端中打印出相应的调试信息。在这里野火用的 MicroSD 卡的容量是 1G。
硬件连接	PE3-USB-MODE (PE3 为普通 I/O) PA11-USBDM(D-) PA12-USBDP(D+)
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/misc.c FWlib/stm32f10x_dma.c FWlib/stm32f10x_sdio.c FWlib/stm32f10x_flash.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/usart1.c USER/sdcard.c USER/usb_istr.c USER/usb_prop.c

	USER/usb_pwr.c USER/hw_config.c USER/memory.c
USB 文件	usb_core.c usb_init.c usb_mem.c usb_regs.c usb_bot.c usb_scsi.c usb_data.c usb_desc.c usb_endp.c

其中 USER 文件夹下红色标注的 5 个 c 文件也是 USB 库文件，只因为我们需要修改它们才没有把它们放在 USBLIB 目录下。

stm32f10x_it.c：该文件中包含 USB 中断服务程序，由于 USB 中断有很多情况，这里的中断服务程序只是调用 usb_Istr.c 文件中的 USB_Istr 函数，由 USB_Istr 函数再做轮询处理。

usb_istr.c：该文件中只有一个函数，即 USB 中断的 USB_Istr 函数，该函数对各类引起 USB 中断的事件作轮询处理。

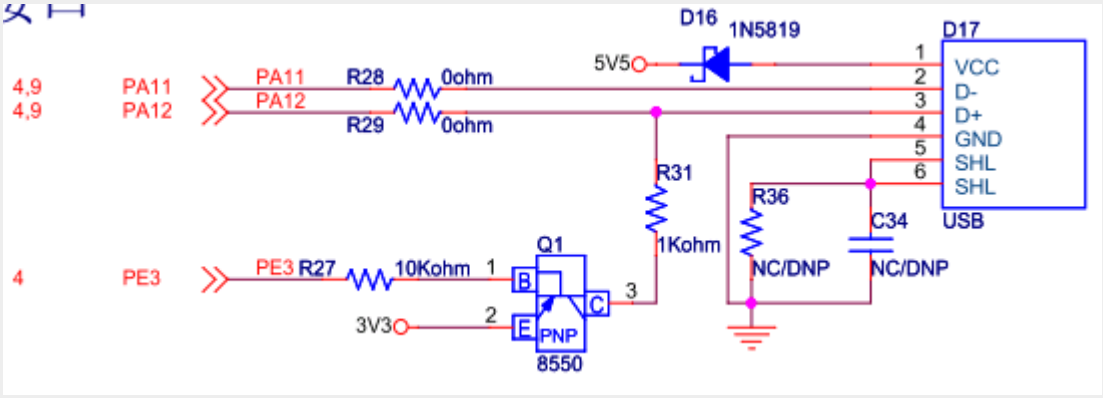
usb_prop.c：该文件用于实现相关设备的 USB 协议，例如初始化、SETUP 包、IN 包、OUT 包等等。

usb_pwr.c：该文件中包含处理上电、调电、挂起和恢复事件的函数。

memory.c：该文件中包含 USB 读写 SD 卡的函数。

hw_config.c：该文件中包含系统配置的函数。

野火 STM32 开发板中 USB 硬件原理图



6.2 USB 简介

USB 模块为 PC 主机和微控制器所实现的功能之间提供了符合 USB 规范的通信连接。PC 主机和微控制器之间的数据传输是通过共享一专用的数据缓冲区来完成的，该数据缓冲区能被 USB 外设直接访问。这块专用数据缓冲区的大小由所使用的端点数目和每个端点最大的数据分组大小所决定，每个端点最大可使用 512 字节缓冲区，最多可用于 16 个单向或 8 个双向端点。

USB 模块同 PC 主机通信，根据 USB 规范实现令牌分组的检测，数据发送/接收的处理，和握手分组的处理。整个传输的格式由硬件完成，其中包括 CRC 的生成和校验。每个端点都有一个缓冲区描述块，描述该端点使用的缓冲区地址、大小和需要传输的字节数。当 USB 模块识别出一个有效的功能/端点的令牌分组时，(如果需要传输数据并且端点已配置)随之发生相关的数据传输。

USB 模块通过一个内部的 16 位寄存器实现端口与专用缓冲区的数据交换。在所有的数据传输完成后，如果需要，则根据传输的方向，发送或接收适当的握手分组。在数据传输结束时，USB 模块将触发与端点相关的中断，通过读状态寄存器和/或者利用不同的中断处理程序，微控制器可以确定：

- 哪个端点需要得到服务
- 产生如位填充、格式、CRC、协议、缺失 ACK、缓冲区溢出/缓冲区未满等错误时，正在进行的是哪种类型的传输。

有关更多 USB 的介绍请参考《STM32 参考手册中文》。USB 是一个很复杂的设备，要想全面的学习 USB，光靠做完这个实验和看《STM32 参考手册中文》是远远不够的，还要大量阅读 ST 的官方 USB 文档。还有网上有本关于 USB 方面的书《圈圈教你学 USB》很不错，大家可以去买来研究研究。总之，一句话，USB 耐学，^_^。

6.3 友情提示

USB 是一个比较复杂的知识点，单单 USB 就可以出一本书，在这里野火不可能为大家讲解地非常详细，但这个文档可以为大家扫清代码阅读的障碍。要想更深入的学习 USB，仍需大家的努力，网上有本叫《圈圈教你学 USB》的书就将得非常好，有兴趣的朋友可以买来看看，虽然有电子版，但大伙还是买本书支持下圈圈，毕竟圈圈写书也不容易呀^_^……

6.4 实验讲解

在配置好我们需要用到的库文件之后，我们就从 main 函数开始分析，关于如何添加库文件请参考前面的教程，这里不再详述。

```
/*
 * 函数名：main
 * 描述  ：无
 * 输入  ：无
 * 输出  ：无
 */
int main(void)
{
    /* 配置系统时钟为 72M */
    SystemInit();

    /* 串口 1 配置 15200-8-N-1 */
    USART1_Config();

    /* SD 卡中断配置，优先级最高 */
    NVIC_Configuration();

    /* 等待 SD 卡底层初始化成功 */
    while ( SD_USER_Init() != SD_OK );


    /* 获取 SD 卡容量信息，并在串口打印出来 */
    Get_Medium_Characteristics();

    /* 配置 USB 时钟为 48M */
    Set_USBClock();

    /* 配置 USB 中断 */
    USB_Interrupts_Config();
```

```
30.
31.     /* 配置 USB D+ 线为全速模式 */
32.     USB Cable Config (ENABLE);
33.
34.     /* USB 系统初始化 */
35.     USB Init();
36.
37.    /* 等待 USB 中断到来，在中断函数中进行数据的传输 */
38.     while (1)
39.     {
40.     }
41. }
```

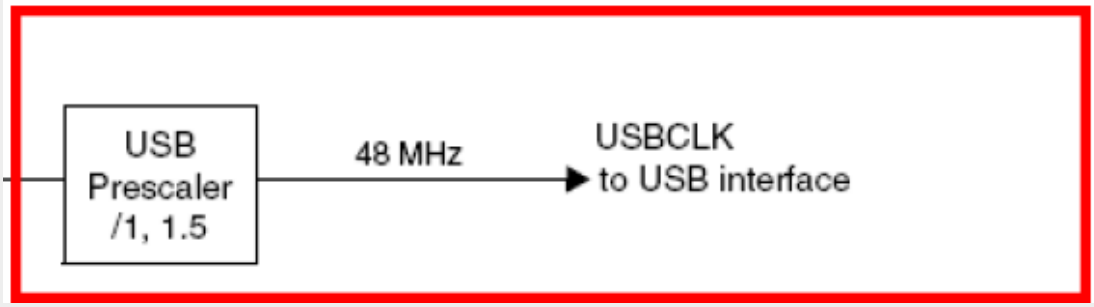
系统库函数 `SystemInit();`将系统时钟配置为 72MHZ， `USART1_Config();`初始化等下要用到的串口 1，用于打印 MicroSD 卡的容量信息。有关这两个函数的详细介绍请参考前面的教程，这里不再详述。

`NVIC_Configuration();`用于配置 MicroSD 卡的中断优先级，本实验中配置为最高优先。

`while ( SD_USER_Init() != SD_OK );`用于等待 MicroSD 卡底层硬件初始化成功，`SD_USER_Init()`在 sdcard.c 中实现。只有这个初始化成功了，接下来才能通过 USB 的方式来访问，所以才采用 `while ( );`的写法，如果初始化不成功的话则一直等待，直到初始化成功为止。`SD_USER_Init()`在 main.c 中实现。

`Get_Medium_Characteristics();`用来获取 MicroSD 卡的容量信息，并通过串口 1 在超级终端中打印出来。`Get_Medium_Characteristics();`也在 main.c 中实现。

`Set_USBClock();`将 USB 的时钟设置为 48MHZ，其实 USB 的时钟必须设置为 48MHZ，如下图所示，截图来自《STM32 参考手册中文》第 47 页。`Set_USBClock();`在 hw_config.c 中实现。




```
1.  /*
2.   * 函数名：Set_USBClock
3.   * 描述  ：配置 USB 时钟(48M)
4.   * 输入  ：无
5.   * 输出  ：无
```

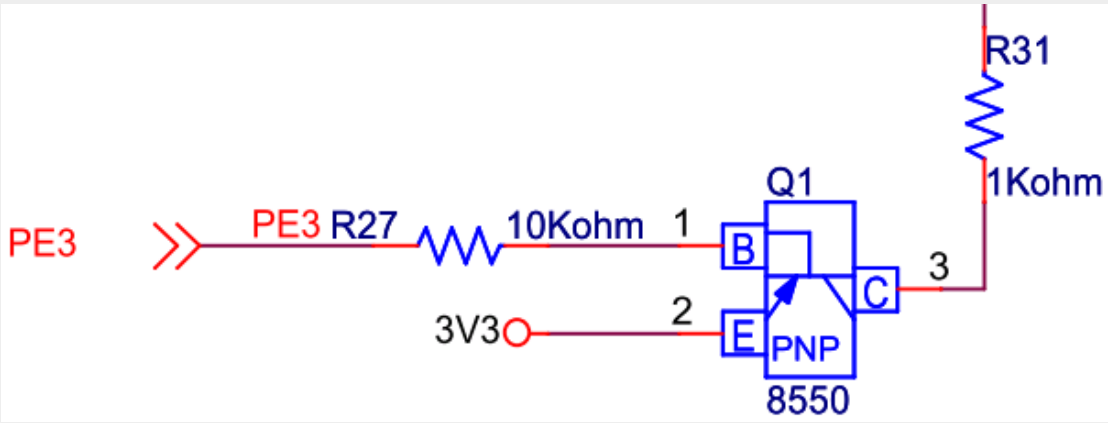
```
6.   */
7.  void Set_USBClock(void)
8.  {
9.     /* USBCLK = PLLCLK */
10.   RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);
11.
12.   /* Enable USB clock */
13.   RCC_APB1PeriphClockCmd(RCC_APB1Periph_USB, ENABLE);
14. }
```

USB_Interrupts_Config();用于配置 USB 的中断优先级，本实验中 USB 的中断优先级次于 MicroSD 卡的中断优先级。USB_Interrupts_Config();在 hw_config.c 中实现。

USB_Cable_Config (ENABLE); 用于配置 USB D+ 这跟数据线为全速模式，即 D+这根数据线接一个上拉电阻。当 D-数据线接上拉电阻时则工作于低速模式。USB_Cable_Config (); 在 hw_config.c 中实现：

```
1.  /*
2.   * 函数名: USB_Cable_Config
3.   * 描述  : Software Connection/Disconnection of USB Cable
4.   * 输入  : -NewState: new state
5.   * 输出  : 无
6.   */
7.  void USB_Cable_Config (FunctionalState NewState)
8.  {
9.      GPIO_InitTypeDef GPIO_InitStructure;
10.     RCC APB2PeriphClockCmd(RCC APB2Periph GPIOE, ENABLE);
11.
12.   /* PE3 输出 0 时 D+ 接上拉电阻工作于全速模式 */
13.   GPIO InitStructure.GPIO Pin = GPIO Pin 3;
14.   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
15.   GPIO InitStructure.GPIO Mode = GPIO Mode Out OD;      /* 开漏输出 */
16.   GPIO_Init(GPIOE, &GPIO_InitStructure);
17.
18.   if (NewState != DISABLE)
19.   {
20.     GPIO_ResetBits(GPIOE, GPIO_Pin_3);                    /* USB 全速模式 */
21.   }
22.   else
23.   {
24.     GPIO_SetBits(GPIOE, GPIO_Pin_3);                          /* 普通模式 */
25.   }
26. }
```

在硬件设计上我们通过 I/O 控制一个三极管的通断来决定 D+这跟数据线是否上拉：



USB_Init();用于系统初始化，USB_Init();在 usb_init.c 中实现：

```
1.  /*******************************************************
2.  * Function Name  : USB_Init
3.  * Description    : USB system initialization
4.  * Input          : None.
5.  * Output         : None.
6.  * Return         : None.
7.  *******************************************************/
8.  void USB_Init(void)
9.  {
10.   pInformation = &Device_Info;
11.   pInformation->ControlState = 2;
12.   pProperty = &Device_Property;
13.   pUser_Standard_Requests = &User_Standard_Requests;
14.   /* Initialize devices one by one */
15.   pProperty->Init();
16. }
```

如果我们在调试程序时，不成功的话，一般都是这个函数出了问题，而一般的问题又会是硬件的问题，我当初调试的时候就郁闷了很久。

最后让程序在一个 while (1) { } 无限循环总等待 USB 中断的到来，然后进行中断服务程序的处理。USB 中断函数 void USB_Istr(void)在 usb_istr.c 中实现：

```
1.  /*******************************************************
2.  * Function Name  : USB_Istr
3.  * Description    : ISTR events interrupt service routine
4.  * Input          : None.
5.  * Output         : None.
6.  * Return         : None.
7.  *******************************************************/
8.  void USB_Istr(void)
```

```
9.  {
10.   wIstr =  GetISTR();
11.
12. #if (IMR_MSK & ISTR_RESET)
13.   if (wIstr & ISTR_RESET & wInterrupt_Mask)
14.   {
15.     _SetISTR((u16)CLR_RESET);
16.     Device Property.Reset();
17. #ifdef RESET CALLBACK
18.     RESET Callback();
19. #endif
20.   }
21. #endif
22.   /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*/
23. #if (IMR_MSK & ISTR_DOVR)
24.   if (wIstr & ISTR_DOVR & wInterrupt_Mask)
25.   {
26.     _SetISTR((u16)CLR_DOVR);
27. #ifdef DOVR_CALLBACK
28.     DOVR Callback();
29. #endif
30.   }
31. #endif
32.   /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*/
33. #if (IMR_MSK & ISTR_ERR)
34.   if (wIstr & ISTR_ERR & wInterrupt_Mask)
35.   {
36.     _SetISTR((u16)CLR_ERR);
37. #ifdef ERR CALLBACK
38.     ERR Callback();
39. #endif
40.   }
41. #endif
42.   /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*/
43. #if (IMR_MSK & ISTR_WKUP)
44.   if (wIstr & ISTR_WKUP & wInterrupt_Mask)
45.   {
46.     SetISTR((u16)CLR WKUP);
47.     Resume(RESUME_EXTERNAL);
48. #ifdef WKUP_CALLBACK
49.     WKUP Callback();
50. #endif
51.   }
52. #endif
53.   /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*/
54. #if (IMR_MSK & ISTR_SUSP)
55.   if (wIstr & ISTR_SUSP & wInterrupt_Mask)
56.   {
57.
58.     /* check if SUSPEND is possible */
59.     if (fSuspendEnabled)
60.     {
61.       Suspend();
62.     }
63.     else
64.     {
65.       /* if not possible then resume after xx ms */
66.       Resume(RESUME LATER);
67.     }
68.     /* clear of the ISTR bit must be done after setting of CNTR FSUSP */
69.     _SetISTR((u16)CLR_SUSP);
70. #ifdef SUSP_CALLBACK
71.     SUSP_Callback();
72. #endif
73.   }
74. #endif
75.   /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*/
76. #if (IMR_MSK & ISTR_SOF)
77.   if (wIstr & ISTR_SOF & wInterrupt_Mask)
78.   {
79.     _SetISTR((u16)CLR_SOF);
80.     bIntPackSOF++;
81.
82. #ifdef SOF_CALLBACK
83.     SOF Callback();
84. #endif
85.   }
```

```
#endif
  /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*/
#if (IMR_MSK & ISTR_ESOF)
  if (wIstr & ISTR_ESOF & wInterrupt_Mask)
  {
    _SetISTR((u16)CLR_ESOF);
    /* resume handling timing is made with ESOFs */
    Resume(RESUME_ESOF); /* request without change of the machine state */

#ifdef ESOF_CALLBACK
    ESOF_Callback();
#endif
  }
#endif
          /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
    */
        #if (IMR_MSK & ISTR_CTR)
          if (wIstr & ISTR_CTR & wInterrupt_Mask)
          {
            /* servicing of the endpoint correct transfer interrupt */
            /* clear of the CTR flag into the sub */
            CTR_LP();
        #ifdef CTR_CALLBACK
            CTR_Callback();
        #endif
          }
        #endif
        } /* USB_Istr */
```

实验到了这里，我们已经可以通过 USB 来操作我们开发板上的 MicroSD 卡了。有关更多实验的细节请大家阅读源码。

6.5 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上 MicroSD 卡，插上方口的 USB 线，将编译好的程序下载到开发板，如果程序运行成功，则可在电脑上看到开发板上的 U 盘(我用的是 1G 的卡)，如下所示：

打开 U 盘，可看到里面的文件。还可以进行新建、复制、粘贴、格式化等操作。与操作我们的普通 U 盘没什么区别。

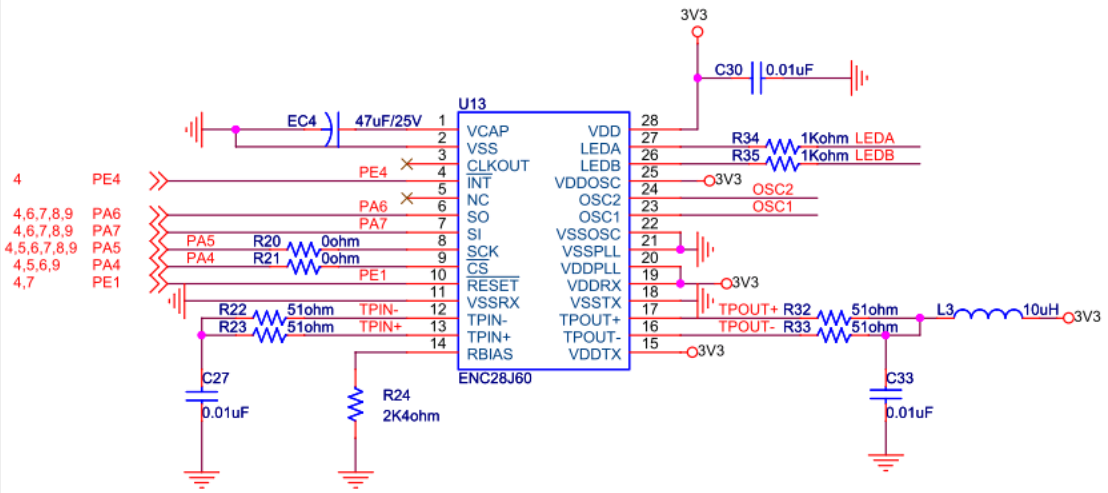
7、以太网（ENC28J60）

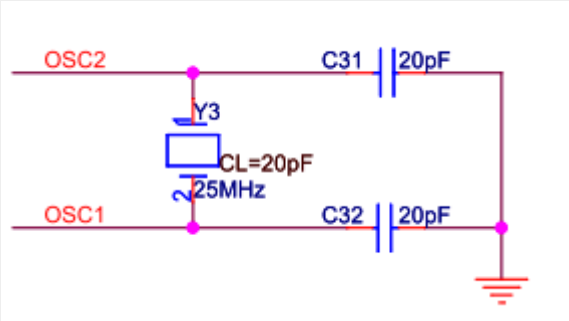
7.1 实验描述及工程文件清单

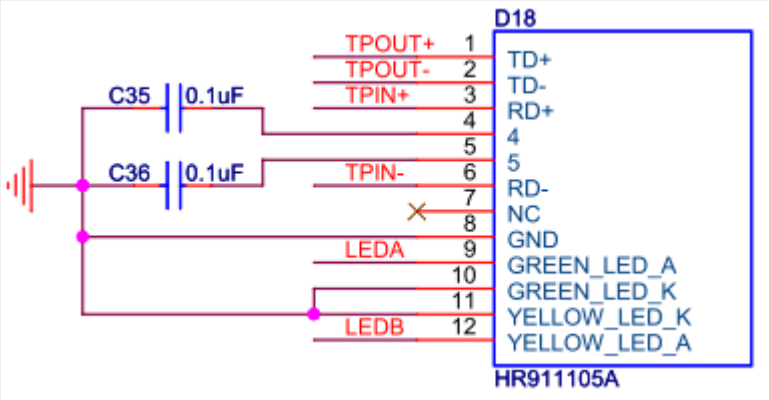
实验描述	在浏览器上创建一个 web 服务器，通过 web 里面的命令来控制开发板上的 LED 的亮灭。 应用->: 1:在 PC 机的 DOS 界面输入： ping 192.168.1.15 ，看能否 ping 通。 2:在 IE 浏览器中输入：http://192.168.1.15/123456 则会出现一个网页，通过网页中的命令可以控制开发板中的 LED 的亮灭。
硬件连接	PE4 ：ENC28J60-INT PA6-SPI1-MISO ：ENC28J60-SO PA7-SPI1-MOSI ：ENC28J60-SI PA5-SPI1-SCK ：ENC28J60-SCK PA4-SPI1-NSS ：ENC28J60-CS PE1 ：ENC28J60-RST
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/stm32f10x_spi.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c

	USER/led.c USER/usart.c USER/spi_enc28j60.c USER/enc28j60.c USER/ip_arp_udp_tcp.c USER/web_server.c

野火 STM32 开发板中 10M 以太网 ENC28J60 的硬件原理图







7.2 ENC28J60 简介

ENC28J60 是带有行业标准串行外设接口 （SerialPeripheral Interface，SPI）的独立以太网控制器。 它可作为任何配备有 SPI 的控制器的以太网接口。*ENC28J60* 符合 IEEE 802.3 的全部规范，采用了一系列包过滤机制以对传入数据包进行限制。 它还提供了一个内部 DMA 模块，以实现快速数据吞吐和硬件支持的 IP 校验和计算。 与主控制器的通信通过两个中断引脚和 SPI 实现，数据传输速率高达 10 Mb/s。两个专用的引脚用于连接 LED，进行网络活动状态指示。

下图所示为 ENC28J60 的简化框图。 图 1-2 所示为使用该器件的典型应用电路。 要将单片机连接到速率为 10 Mbps 的以太网，只需 ENC28J60、两个脉冲变压器和一些无源元件即可。本开发板中用的网络变压器的型号为 911105A。

图 1-2： 典型的 ENC28J60 接口

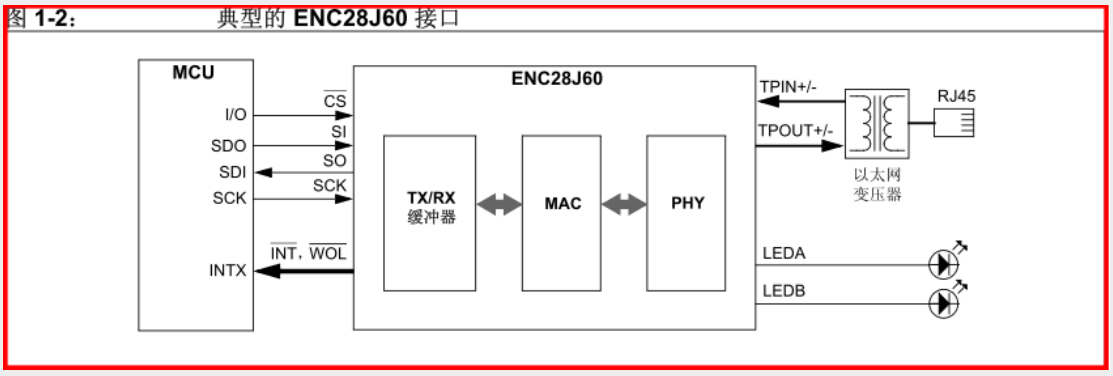
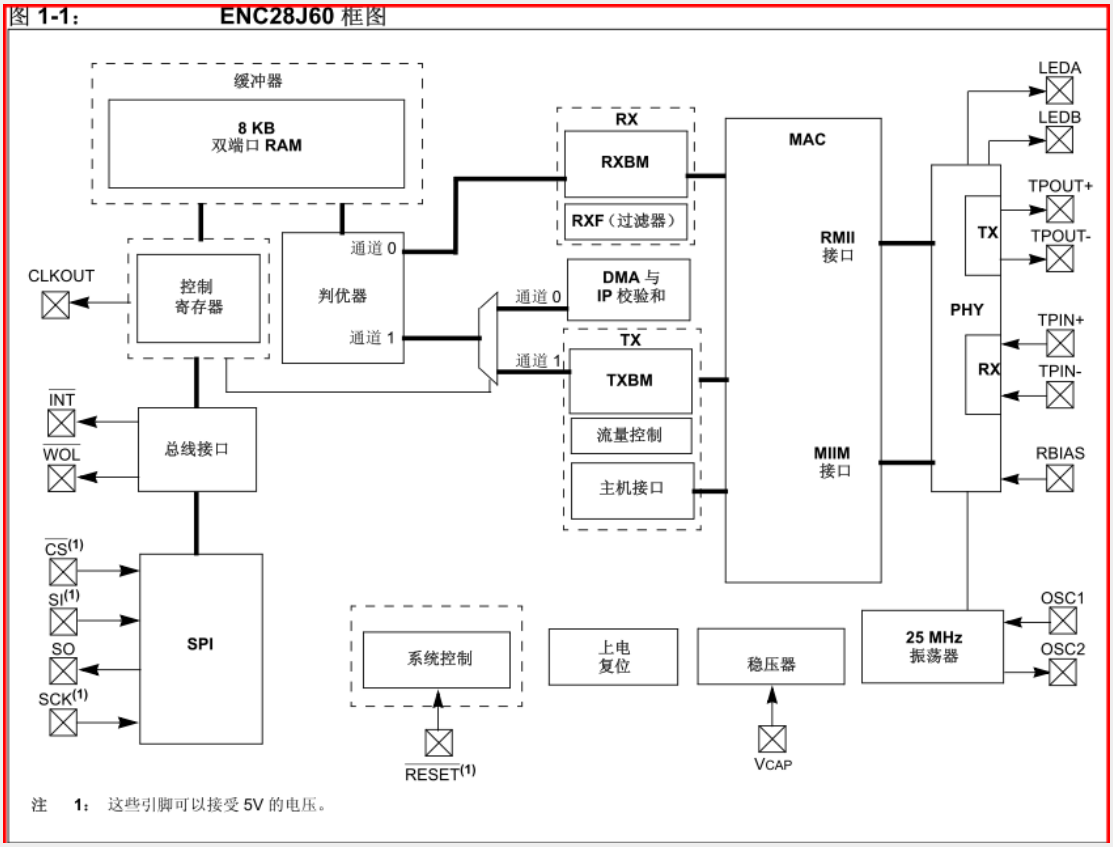


图 1-1： ENC28J60 框图



ENC28J60 由七个主要功能模块组成：

1. SPI 接口——充当主控制器和 ENC28J60 之间通信通道。
2. 控制寄存器——用于控制和监视 ENC28J60。
3. 双端口 RAM 缓冲器——用于接收和发送数据包。
4. 判优器——当 DMA、发送和接收模块发出请求时对 RAM 缓冲器的访问进行控制。
5. 总线接口——对通过 SPI 接收的数据和命令进行解析。
6. MAC （Medium Access Control）模块——实现符合 IEEE 802.3 标准的 MAC 逻辑。
7. PHY（物理层）模块——对双绞线上的模拟数据进行编码和译码。

该器件还包括其他支持模块，诸如振荡器、片内稳压器、电平变换器（提供可以接受 5V 电压的 I/O 引脚）和系统控制逻辑。

7.3 实验讲解

建议阅读程序的顺序为：spi_enc28j60.c -> enc28j60.c -> ip_arp_udp_tcp.c ->web_server.c 。

spi_enc28j60.c ： ENC28J60(以太网芯片) SPI 接口应用函数库。

enc28j60.c ： Microchip ENC28J60 Ethernet Interface Driver。

ip_arp_udp_tcp.c ： IP, Arp, UDP and TCP functions（这部分野火仍在学习）。

web_server.c ： web 服务程序应用函数库。

其中 enc28j60.c 、ip_arp_udp_tcp.c web_server.c 是从 国外的一个开源项目里面移植过来的，源文件基本上没有做修改。spi_enc28j60.c 是由我们用户实现的底层函数接口，还有我们修改了 web_server.c 这个文件中网页命令控制部分的服务程序。

在配置好需要用的库文件之后，下面我们从 main 函数开始讲解，有关库函数是如何添加的情参考前面的教程，这里不再赘述。

```
1.  /*
2.   * 函数名：main
3.   * 描述  ：主函数
4.   * 输入  ：无
5.   * 输出  ：无
6.   */
7.  int main (void)
8.  {
9.    /* 配置系统时钟为 72M */
10.     SystemInit();
11.
12.     /* 配置 LED */
13.   LED_GPIO_Config();
14.
15.     /* ENC28J60 SPI 接口初始化 */
16.   SPI_Enc28j60_Init();
17.
18.     /* ENC28J60 WEB 服务程序 */
19.   Web_Server();
20.
21.     //return 0;
22. }
```

在进入 main 函数代码段后，我们首先调用系统库函数 `SystemInit();`将我们的系统时钟配置为 72MHZ，如果用的是 3.5.0 版本的库则不需要，因为已在启动文件里面调用了。

`LED_GPIO_Config();`用于初始化 LED，因为我们我们在我们的 web 服务器中要控制的就是 LED，所以在这里要先把 LED 配置好，好让它接下来能工作。

`SPI_Enc28j60_Init();`用于配置以太网芯片 ENC28J60 所用到的数据通信口 SPI2 和其他控制 I/O。这是我们用户在 spi_enc28j60.c 中实现的底层程序。

```
/*
 * 函数名: SPI1_Init
 * 描述  : ENC28J60 SPI 接口初始化
 * 输入  : 无
 * 输出  : 无
 * 返回  : 无
 */
void SPI_Enc28j60_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef  SPI_InitStructure;

    /* 使能 SPI1 时钟 */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_SPI1, ENABLE);

   /*
    * PA5-SPI1-SCK :ENC28J60_SCK
    * PA6-SPI1-MISO:ENC28J60_SO
    * PA7-SPI1-MOSI:ENC28J60_SI
    */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;       // 复用输出
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* PA4-SPI1-NSS:ENC28J60_CS */ // 片选

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;        // 推免输出
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_SetBits(GPIOA, GPIO_Pin_4);

    /* PB13:ENC28J60_INT */ // 中断引脚没用到
    /* PE1:ENC28J60_RST*/   // 复位似乎不用也可以


    /* SPI1 配置 */
    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_Low;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;

    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStructure.SPI_CRCPolynomial = 7;
    SPI_Init(SPI1, &SPI_InitStructure);

    /* 使能 SPI1  */
```

```
51.     SPI_Cmd(SPI1, ENABLE);
52. }
```

在这个函数中不知大家有没注意到没有这两条注释：

```
1.      /* PB13:ENC28J60_INT */ // 中断引脚没用到
2.
3.      /* PE1:ENC28J60_RST*/    // 复位似乎不用也可以
```

enc28j60 的中断引脚没用到很正常，但是复位引脚也没用到，这我就很纳闷了。我想原因可能是 enc28j60 有个上电自动复位的功能，这里它的复位引脚只能暂时没有用到而已，也或许是我们的开发板中引脚 PE1（接 enc28j60 的复位脚）收到什么信号的干扰，产生了类似复位的信号。这里我们先把这问题搁一边先，毕竟程序还是工作了。至于具体的原因我以后有时间再深究下。

Web_Server();函数实现的功能是创建一个网页服务器，在这个网页服务器上我可以点击我们设定好的命令按钮来控制我们开发板上 LED 的亮灭。其实，从这里面我们就可以看到有点智能家居的味道了，所谓智能家居就是通过网络来控制我们家电的状态，如开和断。举个例子：我们可以在遥远的地方可以通过网络来控制家里的电视、电冰箱等，是不是很神奇哩？只要你学会了这个实验，再经过自己的再深入学习，这些都是小菜一碟。^_^，本实验旨在引导大家入门。

Web_Server();在 web_server.c 中实现。在 web_server.c 的开头包含了头文件：

```
1. #include "enc28j60.h"
2. #include "ip_arp_udp_tcp.h"
3. #include "net.h"
```

enc28j60.h ： Microchip ENC28J60 Ethernet Interface Driver Header file

ip_arp_udp_tcp.h： IP, Arp, UDP and TCP functions Header file . For more Infermation Please See http://www.gnu.org/licenses/gpl.html

net.h : Based on the net.h file from the AVRlib library by Pascal Stang.

For AVRlib See http://www.procyonengineering.com/

Used with explicit permission of Pascal Stang.

有关这三个头文件对应的 C 文件的功能，请大家阅读源码。

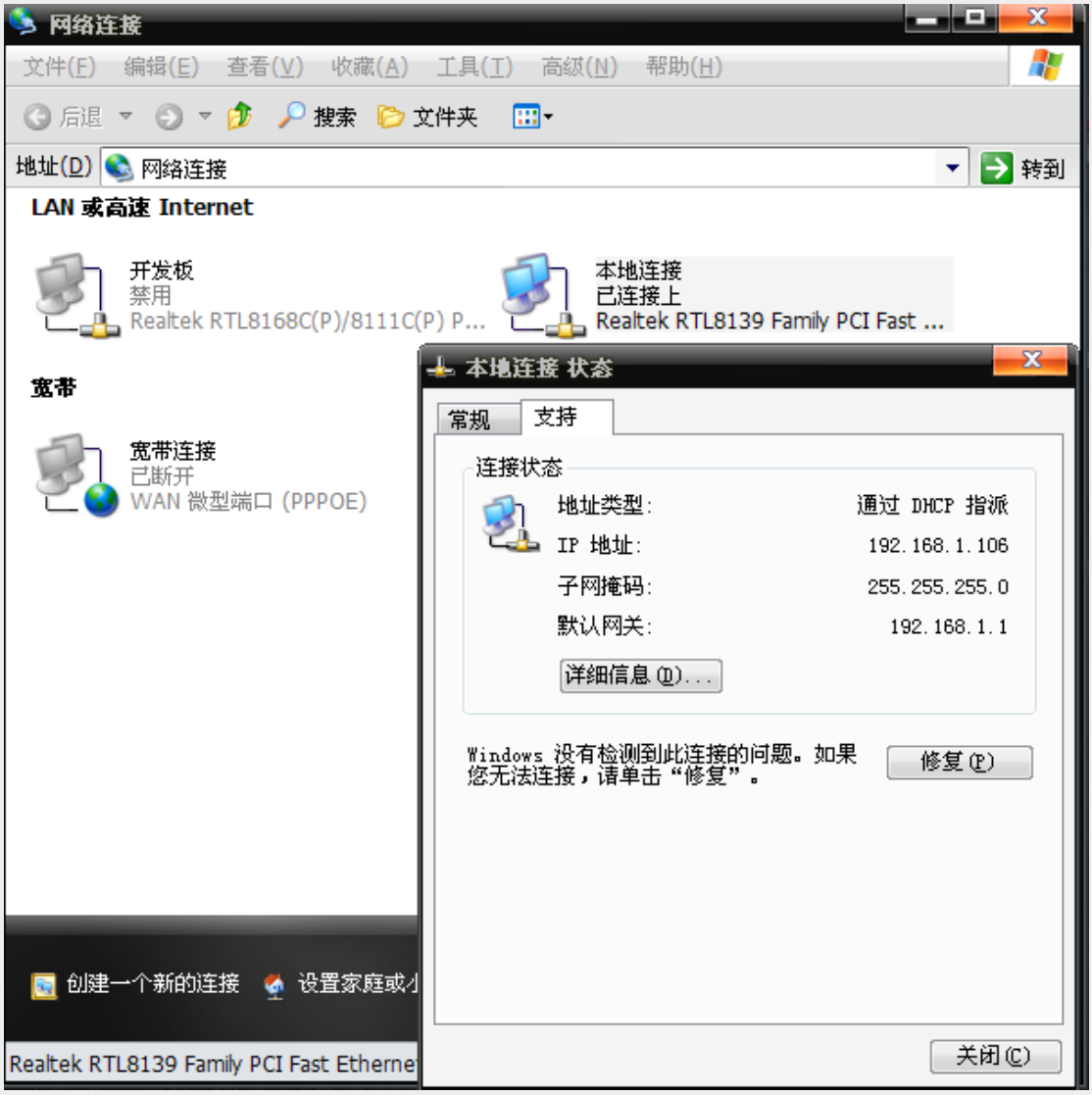
接下来我们具体看看 `Web_Server();` 这个函数具体做了什么，由于这个函数的源码较多，在这里我就不贴出来了。

enc28j60Init(mymac); 这个函数用来初始化 enc28j60 的 MAC 地址(物理地址),这个函数必须要调用一次。mymac 在 web_server.c 的开头定义：

static unsigned char mymac[6] = {0x54,0x55,0x58,0x10,0x00,0x24}; 这里要注意的是：mac 地址在局域网内必须唯一，否则将与其他主机冲突，导致连接不成功。mymac 数组里面的数值可随便初始化，但万万不可与局域网内的 mac 地址冲突，否则当另外一部主机在上网时，你是上不了网的。

enc28j60PhyWrite(PHLCON,0x476);这个函数用于设定网络变压器中 LED 的颜色，不同的颜色指示不同的状态。网络变压器在没有工作的情况下，这两个 LED 是不会被点亮的，当网络变压器工作正常时，绿色 LED 表示 link 状态，黄色 LED 表示通信状态。本实验中，我们的程序工作正常时，绿色 LED 常亮，黄色 LED 是一闪一闪的。

init_ip_arp_udp_tcp(mymac,myip,mywwwport); 这个函数用于初始化以太网的 IP 层。这里面涉及到三个参数：mymac、myip、mywwwport。其中 mymac 在前面已经讲解过。ip 指的是我们开发板的 ip 地址，要想通过网页来访问我们的服务器(即开发板)，则必须要有 ip。ip 地址跟 mac 地址一样，在局域网能要保持唯一，不能够与其他主机的 ip 地址产生冲突。还要注意的一点就是：开发板的 ip 与我们电脑的 ip 必须保持在同一个网段，即 ip 地址的前三段要保持一致，后面一段不同。在本地链接中可查看到我们电脑的 ip 地址。我的电脑的 ip 地址是：192.168.1.106，如下截图所示：



所以，在此设置我们开发板的 ip 为 static unsigned char myip[4] = {192,168,1,15}; ip 的最后一段的值 15 可改为其他值，但要注意不要产生冲突。

接下来的是一个无限循环 while （ 1 ），在这里面我们创建了一个网页，在网页中注入了我们自己的信息，并随时监控我们命令状态的改变，好实时地控制我们的 LED，有关这些功能的实现请大家自行阅读源码，这里不再详述。我们仅来看看 LED 控制的部分：

```
1.  if (cmd==1)       // 用户程序
2.  {
3.      LED1(ON);
4.      i=1;          // 命令 = 1
5.  }
```

```
6.  if (cmd==0)      // 用户程序
7.  {
8.      LED1(OFF);
9.      i=0;           // 命令 = 0
10. }
```

当我们在网页上点击 点亮 LED 这个按钮时，网页发送命令 1 给开发板，这是开发板中的 LED 被点亮，反之，LED 则被关闭。

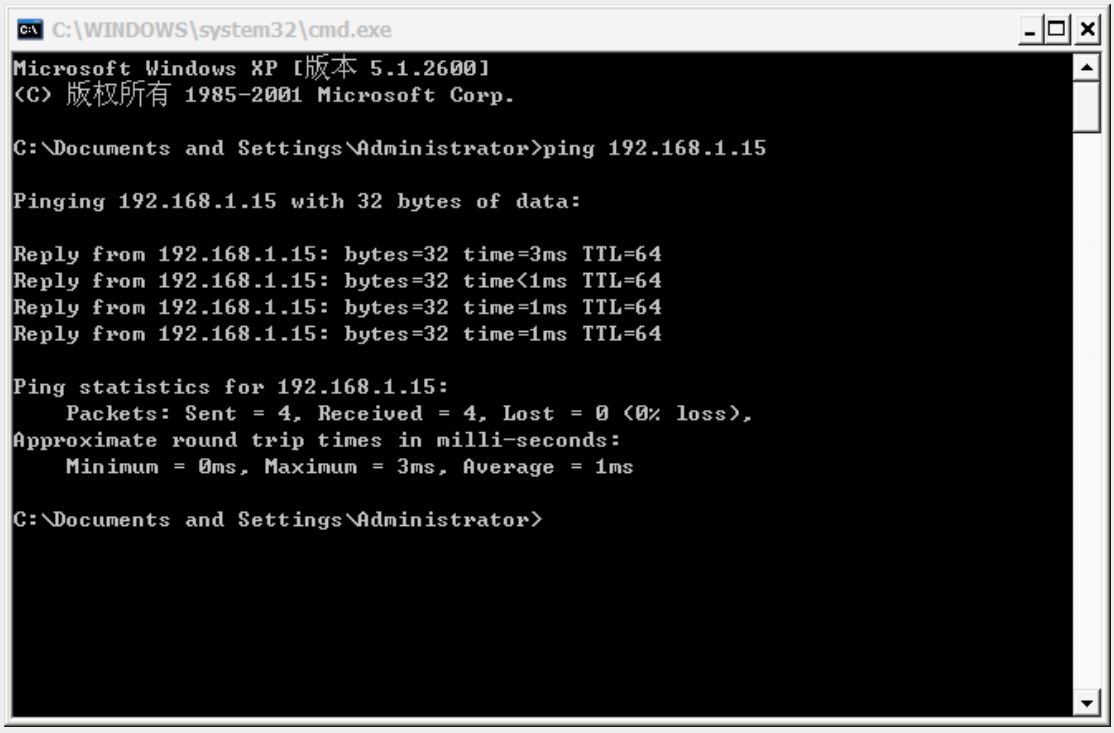
这里我们仅提供简单的 web 服务程序，如果要实现更加复杂的功能，仍需大家的努力。当然，野火也会跟大家一起奋斗，开发出更多的应用程序跟大家分享。

这里面的操作涉及到很多 enc28j60 的知识，特别是寄存器的操作，具体的请大家参考 enc28j60 的 datasheet，大家一定要看看，而且是要认认真真、仔仔细细地看，直到弄懂为止。

7.4 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上网口线，网口线一端连接路由器，一端连开发板。注意：电脑跟开发板的网线连接的路由器要同在一个局域网中。将编译好的程序下载到开发板。

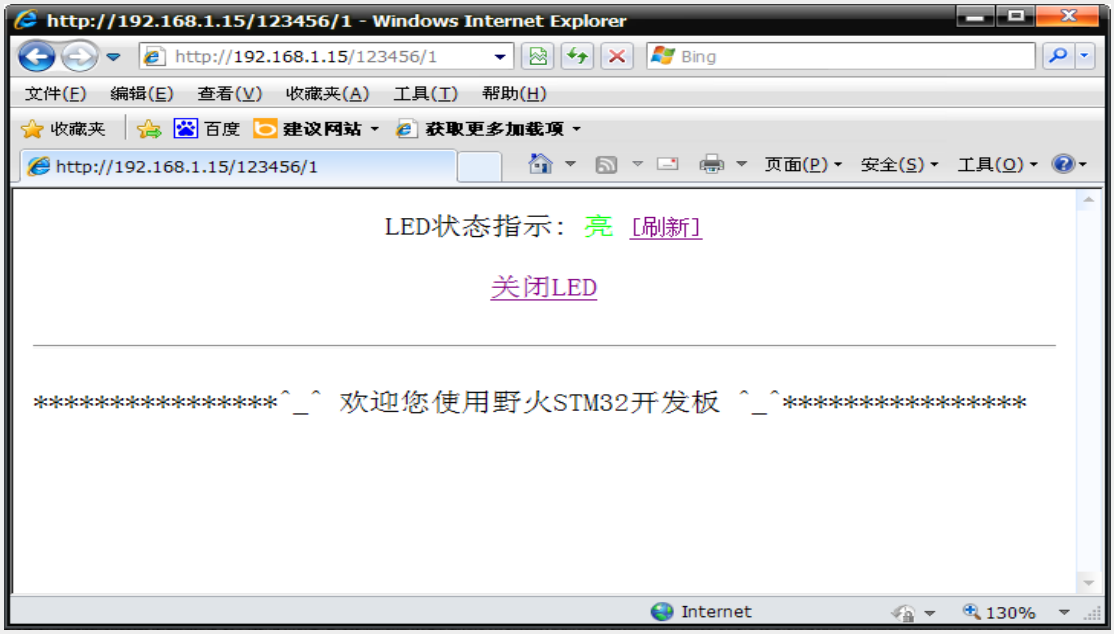
1：打开电脑的 DOS 界面，输入： ping 192.168.1.15 ，看看能否 ping 通。打印出如下信息则表示 ping 通，其中 192.168.1.15 是我们开发板的 ip。



2：打开 IE 浏览器，在地址栏输入：http://192.168.1.15/123456 ，其中 123456 是我们自己设置的密码。然后按 Enter 键进入，如果成功则会出现一个如下网页，通过网页中的命令按钮 点亮 LED /关闭 LED 就可以控制我们 开发板中 LED 的亮灭，这里控制的是开发板中的 LED1。

LED1 灭

LED1 亮


http://192.168.1.15/123456/1 - Windows Internet Explorer
http://192.168.1.15/123456/1
Bing
文件(F) 编辑(E) 查看(V) 收藏夹(A) 工具(T) 帮助(H)
收藏夹 百度 建议网站 获取更多加载项
http://192.168.1.15/123456/1
页面(P) 安全(S) 工具(O)
LED状态指示： 亮 [刷新]
关闭LED
**********^_^ 欢迎您使用野火STM32开发板 ^_^**********
Internet
130%

8、LWIP

8.1 友情提醒

Lwip 的教程非常庞大且深入，是《stm32 库开发实战指南》里面的内容，鉴于跟出版社的合约的关系，这里不能够开源，但野火 STM32 开发板里面提供了源码。欲知更详细内容，请关注机械工业出版社将于 10 月份出版的《stm32 库开发实战指南》。

LWIP 实验需要用到 ENC28J60 这个以太网模块，野火 STM32 开发板已经板载了这个模块，可直接做 LWIP 实验。

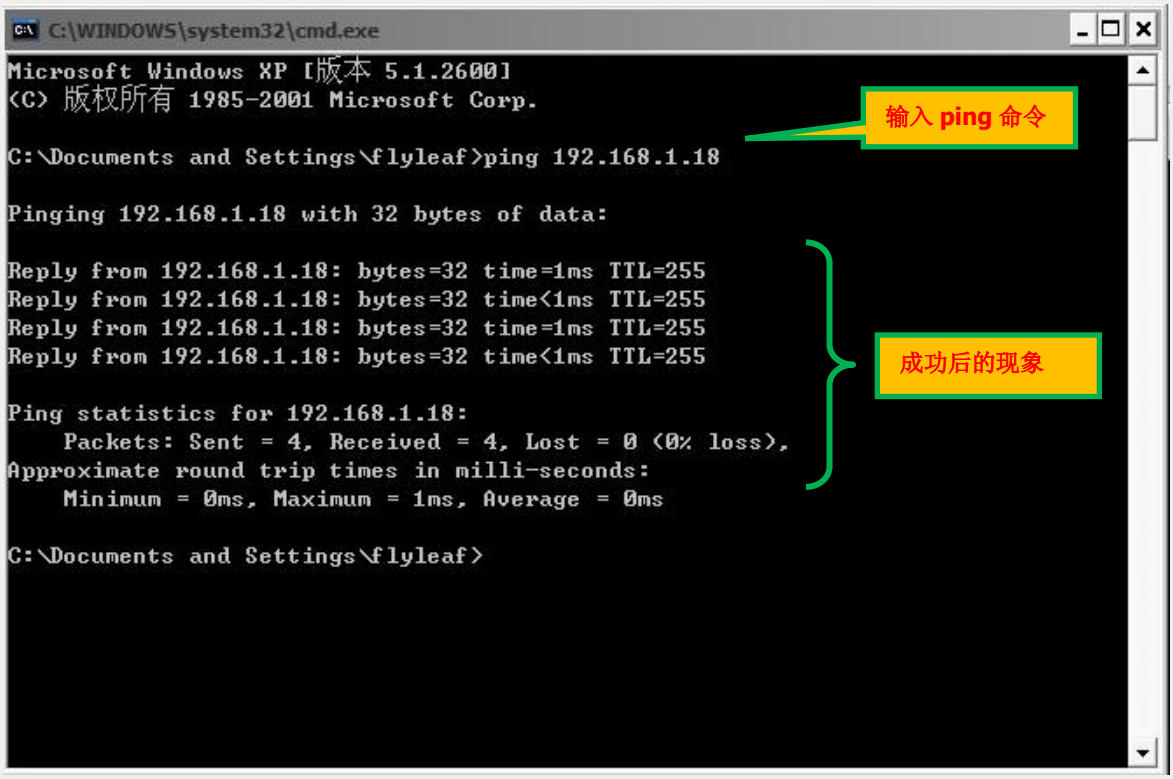
8.2 实验步骤

下面简单介绍下 LWIP 的操作方法和能够达到的效果。

野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，利用网线把 STM32 开发板接入与 PC 相同的路由，也可以直接利用网线把开发板和 PC 相连，其实验的操作是相同的(这样可以排除路由的问题)，但在进行浏览网页实验时，图片可能无法正常显示。把本工程文件编译后烧录到开发板上，*在程序运行框输入 cmd 命令进入 dos 模式*。

8.2.1 ping 实验

在命令提示符窗口输入命令并回车： *ping 192.168.1.18*



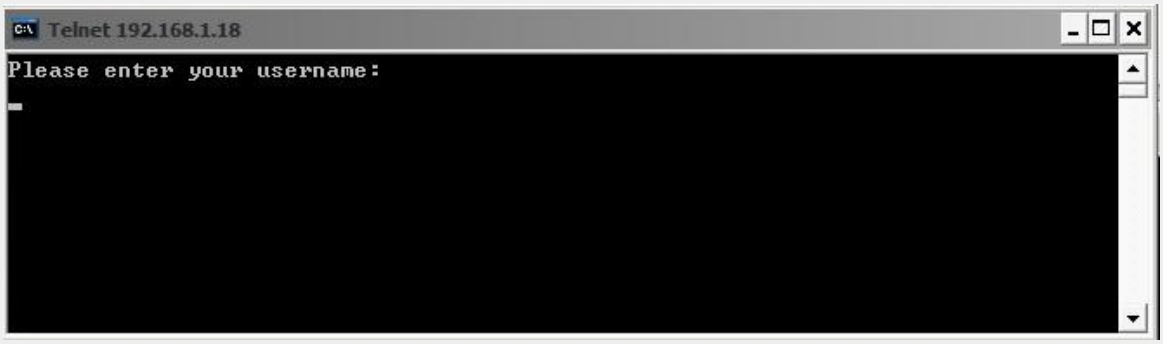
ping 192.168.1.18

8.2.2 telnet 实验

1. 如果使用 windows 7 系统，系统没有 telnet 程序，需要自行下载安装。使用 xp 系统的用户，在命令提示符窗口输入命令并回车：

   telnet 192.168.1.18

输入命令后弹出如下窗口：



进入 telnet 程序

2. 见**错误！未找到引用源。**，在弹出的窗口下输入用户名并回车：

   wildfire

3. 若用户名正确，程序提示输入密码，键入密码并回车：*123456*

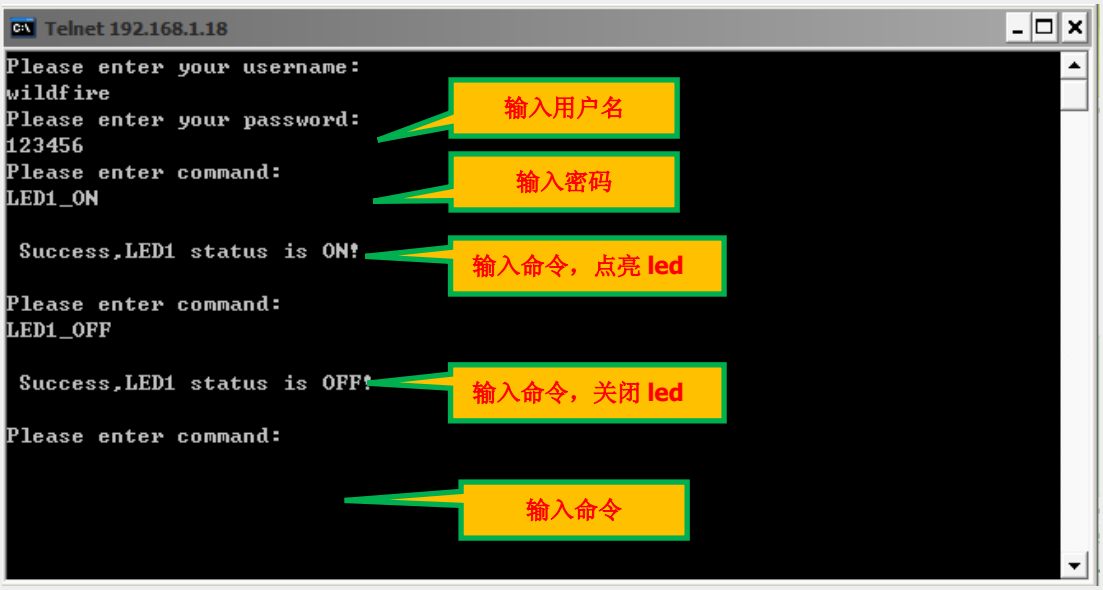
4. 若密码正确，提示输入命令，本工程只允许两条命令，分别为 LED1_ON 和 LED1_OFF，用于控制 LED1 的亮和灭。

输入命令：*LED1_ON*

板上的 LED1 灯会被点亮，窗口会弹出控制成功的信息，并且提示输入命令。

输入命令：*LED1_OFF*

板上的 LED1 会被关灭，窗口弹出控制成功信息，再次提示输入命令。



telnet 控制流程

若用户输入的用户名、密码不正确或不存在的命令，会出现各种提示，并可以重新输入。

8.2.3 网页浏览实验

若 PC 没有接入互联网，图片可能没法正常显示。

1.打开浏览器，在地址栏输入 IP 并回车：*192.168.1.18*

在弹出的网页中输入用户名和密码：*wildfire 123456*


输入地址并回车
192.168.1.18
Wild Fire Team
野火不熄 开源不止
野火
官方旗舰店
单片机 开发板
Cortex M3 M4
STM32 Kinetis
野火STM32开发平台
**********野火STM32开发板STM32F103-ENC28J60-移植LWIP，web服务器测试**********
用户名:
密码:
登录
输入用户名和密码点击登录
野火官方旗舰店 amoBBS论坛野火专区
新品 NEW 上市 敬请期待!
全新外观 酷黑神秘 给力升级 让人爱不释手
野火STM32 V3开发板!
WiFi、Camera、G-sensor、2.4Gwileless
MP3、TF Card、3.2 inch TFT、CAN、485.....
一切尽在野火STM32 v3开发板!
开发板
¥368

网页登录

2.点击登录后，出现如下界面，且开发板上的 LED 被点亮

登录后的页面

3.点选关闭 LED1，并点击控制按钮，网页显示的 LED 状态改变，板上的 LED1 也被关灭。


LWIP移植实验 - Mozilla Firefox
File Edit View History Bookmarks Cocoon Tools Help
LWIP移植实验
192.168.1.18/html/html_form_action.asp
Getting Started Bing135 - 时尚生活 ... 建议网站 网页快讯库
Wild Fire Team
野火不熄 开源不止
野火
官方旗舰店
单片机 开发板
Cortex M3 M4
STM32 Kinetis
野火STM32开发平台
**********野火STM32开发板STM32F103-ENC28J60-移植LWIP，web服务器测试**********
**********！！！登录成功！！！**********
LED状态： 灭
点亮LED1
关闭LED1
控制
野火官方旗舰店 amoBBS论坛野火专区

关闭 LED

9、WIFI

9.1 资料与工具下载

为了更好地理解和使用 Wi-Fi 模块的各项功能，您首先需要下载和学习以下资料，这些文档资料需要通过 Internet 下载：

提供下载地址：

《AN0003_EMW_DataTransferExample.pdf》:透明传输模块使用范例，详细描述了模块在各种模式下的透明传输的使用方法

《RM0001_EMW3280》:EMW 模块使用说明，详细描述了模块的各项功能

《RM0002_EMWToolBox》:EMW 模块配置软件使用说明，详细描述了如何配置模块的各项参数

《DS001_EMW3280_V2.pdf》：EMW 模块的电气特性及引脚定义封装说明。

《RM0001_EMW3280_V02060288》:EMW 模块的工作模式及命令控制集

需要的工具软件：

EMW Tool Box: PC 端配置 Wi-Fi 模块参数工具软件

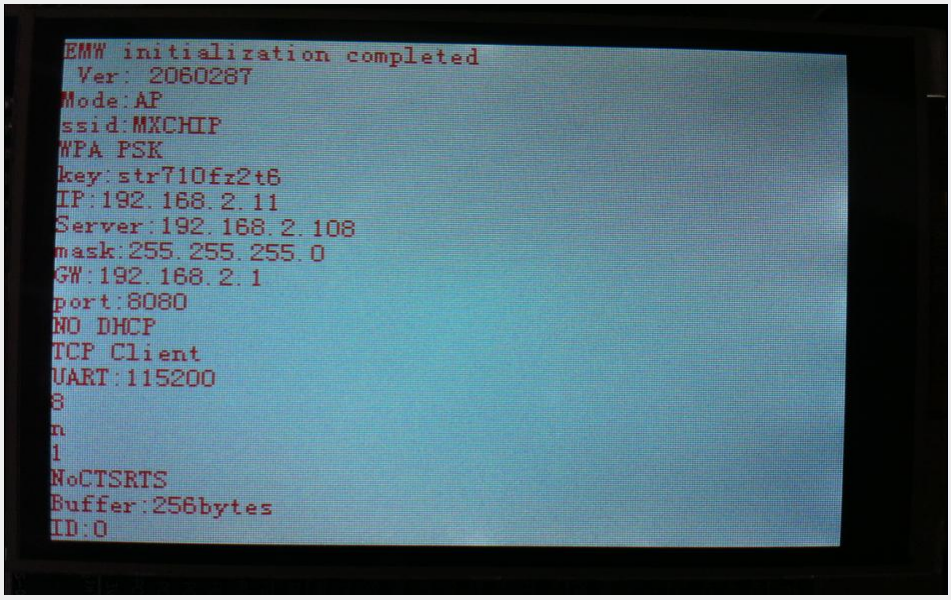
TCP/UDP 测试工具：用于在 PC 端与 WiFi 模块建立 TCP/UDP 连接，实现数据收发串口调试助手或者超级终端：用于配合 WiFi 数据收发调试工具软件

9.2 实验描述

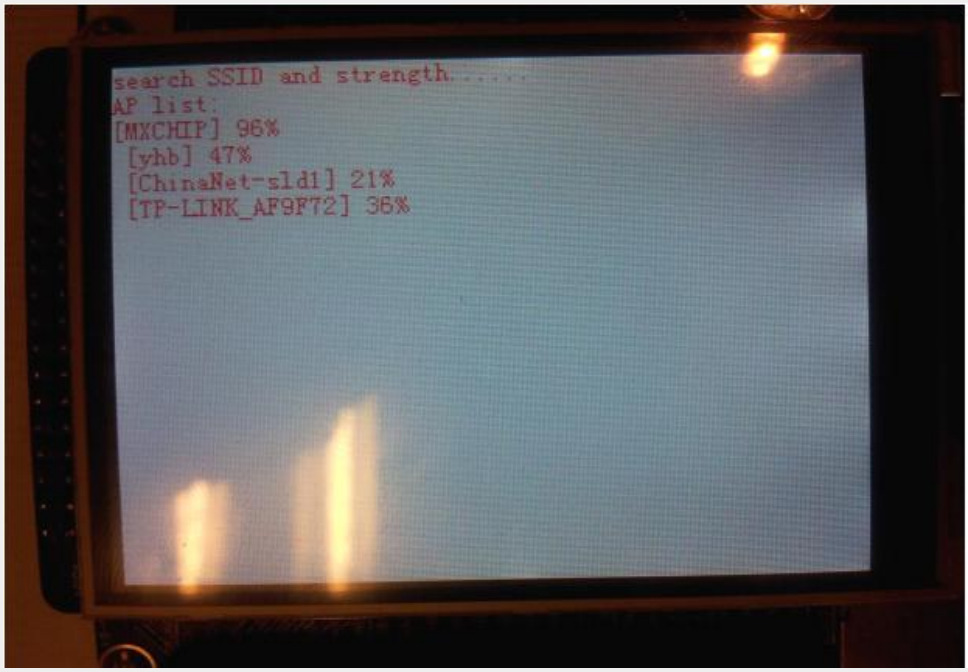
该实验讲解了如何运用 Mxchip 提供的 EMSP_API 函数来配置 Wi-Fi 模块的参数，连接无线网络，与同网段中的 PC 建立 TCP 连接，并打开 PC 端安装 TCP/UDP 测试工具。TCP/UDP 测试工具发送的数据，野火 STM32 开发板通过 Wi-Fi 将接收到的数据返回给 PC，达到回显的功能。

9.2.1 实验现象

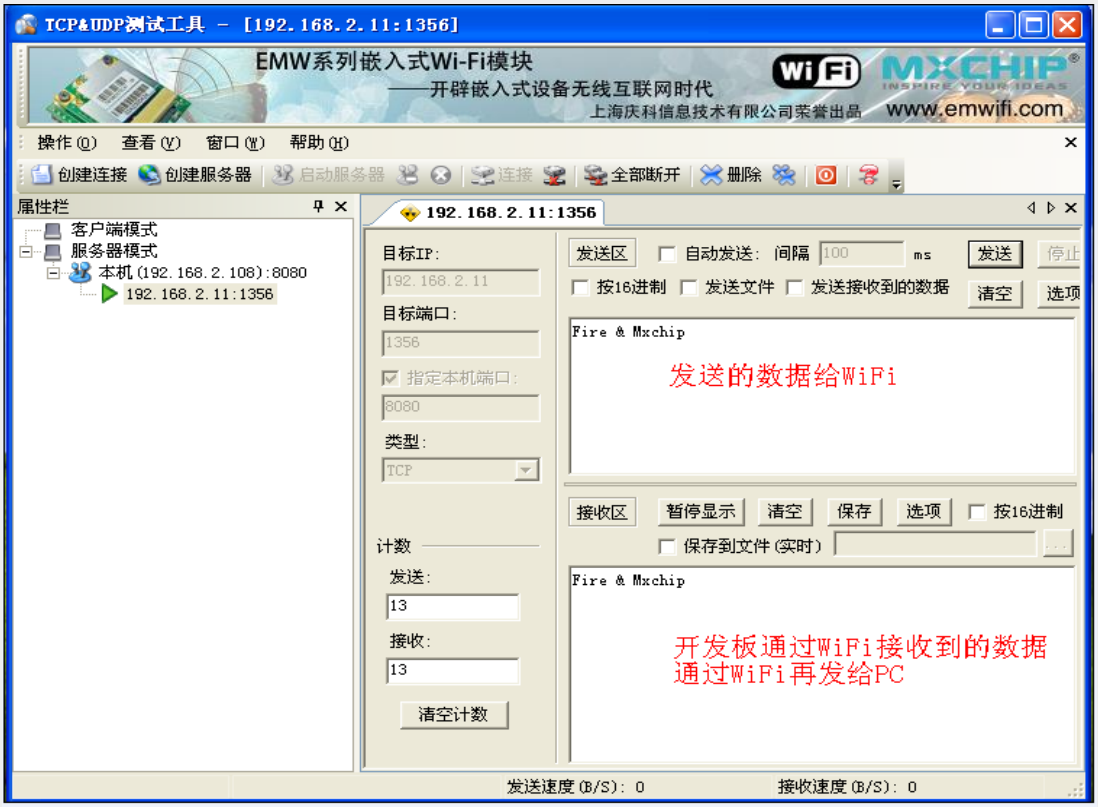
读取到 Wi-Fi 模块的配置参数，并且显示到 LCD 屏



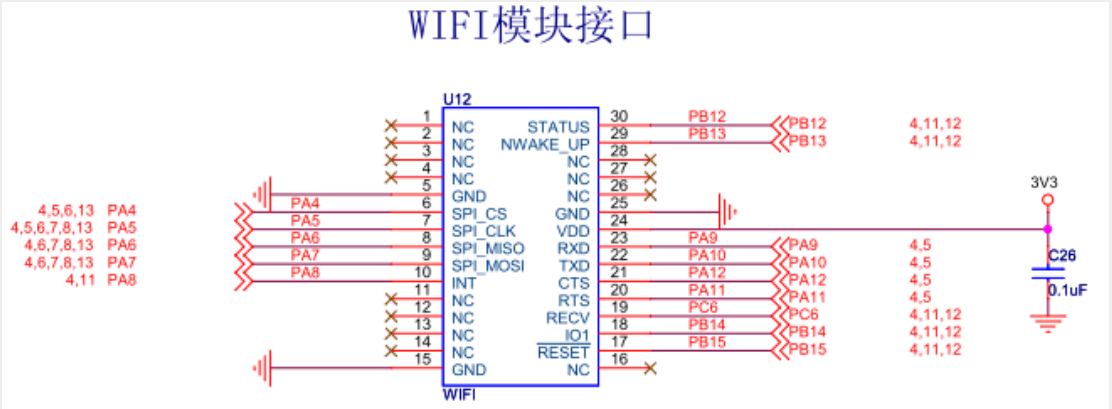
读取搜索到的周围的无线网络和信号强度



野火 STM32 开发板接收到 PC 端 TCP/UDP 测试工具发送的数据，通过 Wi-Fi 模块发送给 PC，达到回显的功能



9.2.2 硬件连接图



- 串口连接说明

RXD - PA9

TXD - PA10

CTS - PA12

RTS - PA11

STATUS - PB12　状态脚

Wake up - PB13　唤醒脚

IO1 - PB14　帧控制

- SPI 接口说明

PA4 - SPI_CS

PA5 - SPI_CLK

PA6 - SPI_MISO

PA7 - SPI_MOSI

PC7 - INT

PC6 - RECV

PB12 - STATUS

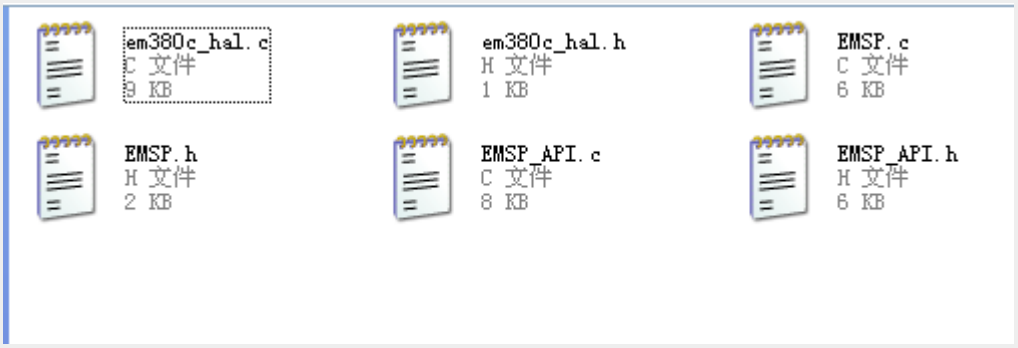
PB13 - Wakeup

SPI 接口模块的 WiFi 例程暂时没有，等到以后做了之后会发布

PS：引脚的具体定义及功能请参考“datasheet”文件夹中的 EMW3280_V2.pdf 文档

9.2.3 EMSP_API 函数

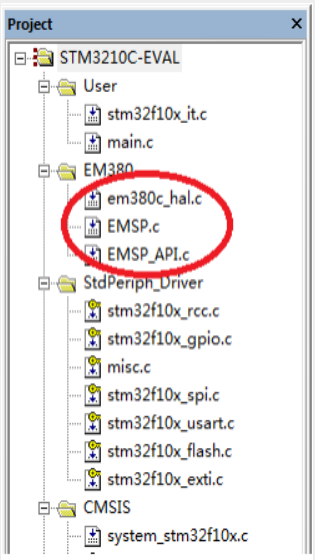
EMSP_API 接口函数提供了一系列 API 函数，用户通过调用这些函数可以轻松地在各种嵌入式设备上实现对 EMW 系列 Wi-Fi 模块的控制和数据传输。现在该接口函数随和 WiFi 资料随野火 STM32 开发板例程一并提供：



如果大家购买了野火 STM32 开发板和 Mxchip EMW 系列 Wi-Fi 模块，就可以在开发板上调试这些例程。

EMSP_API 函数库由标准 C 编写而成，可以直接加入到常用的嵌入式开发环境，如 KEIL，IAR 等。

EMSP_API 函数由以下三个 C 语言文件及其对应的头文件构成。



- emw38x_hal.c

该代码实现了 EMW 系列模块和嵌入式设备之间的硬件接口。用户需要根据自己的硬件环境实现相应的函数

- EMSP.c

该代码实现了 EMSP 命令的协议处理。

- EMSP_API.c

该代码提供给用户用于操控模块的 API 函数，用户只需调用这些函数，就可以实现对模块的配置和操作。

9.2.4 API 函数一览

函数名	vs8 EM380C_Init(void)
功能	用于初始化模块，和与模块通讯的 UART 接口，并使模块处于能够响。
返回值	-1 ： 执行命令失败
	0 ： 执行命令成功

函数名	vs8 EM380C_Get_ver(u32* version)
功能	用于获得 EMW 系列模块的固件版本号。
返回值	-1 ： 执行命令失败
	0 ： 执行命令成功

函数名	vs8 EMSP_Get_status(EM380C_status_TypeDef* EM380C_status)
功能	用于获得 Wi-Fi 模块的网络连接状态。
输入	用于存放 Wi-Fi 的网络连接状态结构体地址 typedef struct { EM380C_TCPstatus_TypeDef TCPstatus; EM380C_WiFistatus_TypeDef WiFistatus; } EM380C_status_TypeDef;
返回值	-1 ： 执行命令失败
	0 ： 执行命令成功

函数名	vs8 EM380C_Get_APList(EM380C_APLst_TypeDef* EM380C_APLst)
功能	用于获得区域内无线 AP 的 SSID 号和相应的信号强度。
输入	用于存放无线的 AP 的 SSID 号和相应的信号强度的线性表的起始地址 typedef struct { char AP_NAME[20]; float AP_signal; } EM380C_APLst_TypeDef;

返回值	-1 ： 执行命令失败
	>=0：执行命令成功，获得的 AP 信息的数量

函数名	vs8 EM380C_Startup(void)
功能	启动 Wi-Fi 模块的 TCP/IP 网络连接。
返回值	-1 ： 执行命令失败
	0 ： 执行命令成功

函数名	vs8 EM380C_Get_RF_POWER(EM380C_RF_POWER_TypeDef*RF_POWER)
功能	用于获得 Wi-Fi 模块当前的配置参数。

输入	参数结构体的地址，成功执行命令后，模块当前的参数会写入这个地址。参数结构体如下。
	typedef struct
	{
	// WIFI
	u8 wifi_mode; //Wlan802_11IBSS(0), Wlan802_11Infrastructure(1)
	u8 wifi_ssid[32]; //
	u8 wifi_wepkey[16]; // 40bit and 104 bit
	u8 wifi_wepkeylen;// 5, 13
	
	// TCP/IP
	u8 local_ip_addr[16];
	u8 remote_ip_addr[16]; // if em380 is server, it is NOT used;if em380 is client, it is server's IP

	u8 net_mask[16]; // 255.255.255.0
	u8 gateway_ip_addr[16]; // gateway ip address
	u8 portH; // High Byte of 16 bit
	u8 portL; // Low Byte of 16 bit
	u8 connect_mode; // 0:server 1:client
	u8 use_dhcp; // 0:disale, 1:enable
	u8 use_udp; // 0:use TCP,1:use UDP
	
	// COM
	u8 UART_buadrate; // 0:9600, 1:19200, 2:38400, 3:57600, 4:115200
	u8 DMA_buffersize; // 0:2, 1:16, 2:32, 3:64, 4:128, 5:256, 6:512
	u8 use_CTS_RTS; // 0:disale, 1:enable
	u8 parity; // 0:none, 1:even parity, 2:odd parity
	u8 data_length; // 0:8, 1:9
	u8 stop_bits; // 0:1, 1:0.5, 2:2, 3:1.5
	
	// DEVICE
	// u8 device_num; // 0 - 255
	u8 IO_Control; // 0 - 255
	u8 sec_mode; // 0 = wep, 1=wpa psk, 2=none
	u8 wpa_psk[32];
	} EM380C_parm_TypeDef;

返回值	-1： 执行命令失败
	0 ： 执行命令成功

函数名	vs8 EM380C_Set_Config(EM380C_parm_TypeDef* EM380C_Parm)
功能	用于设置 Wi-Fi 的配置参数。
输出	参数结构体的地址，成功执行命令后，会将该地址上的数据写入到 WiFi 模块里面去。结构体与上面 GetConfig 参数一致。
返回值	-1 ： 执行命令失败
	0 ： 执行命令成功

函数名	u32 EM380C_Send_Data(u8* Data,u32 len)
功能	用于通过 Wi-Fi 模块发送数据
输出 1	保存发送数据的内存空间的起始地址
输出 2	发送的数据长度
返回值	>0：执行命令成功，返回发送的数据长度
	0 ： 执行命令成功

函数名	vs8 EM380C_Reset(void)
功能	重启 Wi-Fi 模块，配置参数后，需重启模块，参数才能生效
返回值	-1：执行命令成功，返回发送的数据长度
	0 ： 执行命令成功

函数名	vs8 EM380C_Set_Mode(EM380C_mode_TypeDef mode)
功能	设置 Wi-Fi 模块模式，命令模式和透传模式
输入	用于存放 Wi-Fi 模块的模式结构体 typedef enum { config_mode = 0x0, //命令模式

	DTU_mode = 0x1, //透传模式
	} EM380C_mode_TypeDef;

返回值	-1：执行命令成功
	0：执行命令失败

9.2.5 MAIN 函数讲解

第一步，初始化硬件接口(其实这一部拉低 STATUS 引脚，初始化号相应硬件接口即可)

```
064 while(EM380C_Init(BaudRate_115200,WordLength_8b,StopBits_1,Parity_No,HardwareFlowControl_None,buffer_512bytes)==EM380ERROR);
065
066 printf("EMW initialization completed\n");
```

第二步，设置 Wi-Fi 模块参数

```
069
070 //******Config the Wi-Fi moudel parameter******//
071   parm.wifi_mode = AP;
072   strcpy((char*)parm.wifi_ssid,"MXCHIP");
073   strcpy((char*)parm.wifi_wepkey,"");
074   parm.wifi_wepkeylen = 0;
075   strcpy((char*)parm.local_ip_addr,"192.168.2.11");
076   strcpy((char*)parm.remote_ip_addr,"192.168.2.108");
077   strcpy((char*)parm.net_mask,"255.255.255.0");
078   strcpy((char*)parm.gateway_ip_addr,"192.168.2.1");
079   parm.portH =  8080>>8;
080   parm.portL =  8080;
081   parm.connect_mode = TCP_Client;
082   parm.use_dhcp =   DHCP_Disable;
083   parm.use_udp = TCP_mode;
084   parm.UART_buadrate = BaudRate_115200;
085   parm.DMA_buffersize = buffer_256bytes;
086   parm.use_CTS_RTS = HardwareFlowControl_None;
087   parm.parity = Parity_No;
088   parm.data_length = WordLength_8b;
089   parm.stop_bits = StopBits_1;
090   parm.IO_Control = IO1_Normal;
091 #ifdef EMW_FIRMWARE_UART
092   parm.sec_mode = Secure_WPA_WPA2_PSK;
093   strcpy((char*)parm.wpa_psk,"str710fz2t6");
094 #endif
095 while(EM380C_Set_Config(&parm)==EM380ERROR);
```

设置模块参数，并通过 EMSP_SET_CONFIG 命令发送给 Wi-Fi 模块。

模块详细的功能可参考"Wi-Fi 模块 datasheet"文件夹 EMW_DataTransferExamples.pdf，里面详细介绍了各种模式的数据透传。

第三步，重启 Wi-Fi 模块，模块的参数配置好之后，需要重启 Wi-Fi 模块才能生效

```
096
097  while(EM380C_Reset()==EM380ERROR);                    //Reset WiFi module , enable parameter
098
```

第四步，启动 Wi-Fi 模块，通过发送 EMSP_CMD_START 命令，此时模块内部 TCP/IP 协议栈已经跑启，模块上红灯常亮

```
098
099
100  EM380C_Startup();                                     // startup WiFi , connect to AP
101
102
```

第五步，拉高 STATUS 引脚，模块进入透传模式

```
105
106
107     while(EM380C_Set_Mode(DTU_mode)==EM380ERROR);      //EM380C config mode -> DTU mode
108
109
110
```

PS：直接拉高 STATUS 引脚可直接启动 Wi-Fi，跳过第四步

9.3 其他说明：

EMSP 详细命令可参考“datasheet”文件夹中 EMW3280.pdf 文档。

模块还可通过 PC 端的配置工具 EMW Tool Box 配置模块参数,具体使用可参考“datasheet”文件夹中 EMWToolBox2.pdf 文档。

9.4 技术支持

整个 WiFi 的例程讲解到这里就结束了，大家如果想要更详细的资料可以跟 Fire 或者 Mxchip 公司联系。技术问题请到论坛发帖。

10、摄像头

野火 STM32 开发板自带了摄像头接口，用的模块是野火自家生产的 OV7725，跟其他淘宝上卖的 OV7725 模块管脚不兼容。

摄像头部分暂时没有教程，只提供源码，如有什么技术问题请到论坛发帖讨论。