嵌入式系统课程设计报告

课程设计题目：基于触摸屏控制的多功能电子钟

专业班级：XXXXXXXX

姓　　名：zoxiii

学　　号：XXXXXXX

指导教师：XXXX

2020 年 11 月

目录

一、 课程设计目的 7

二、 课程设计内容 7

三、 实验方案分析与设计 7

3.1 RTC 7

3.1.1 模块工作原理 7

3.1.2 硬件电路的连接 9

3.1.3 库函数配置方法 9

3.2 ADC 11

3.2.1 模块工作原理 11

3.2.2 硬件电路的连接 11

3.2.3 库函数配置方法 11

3.3 BEEP 12

3.3.1 模块工作原理 12

3.3.2 硬件电路的连接 12

3.3.3 库函数配置方法 12

3.4 I2S 和 WM8978 13

3.4.1 模块工作原理 13

3.4.2 硬件电路的连接 14

3.4.3 库函数配置方法 15

3.5 SPI 和 W25Q128(FLASH) 15

3.5.1 模块工作原理 15

3.5.2 硬件电路的连接 17

3.5.3 库函数配置方法 17

3.6 I2C 和 AT24C02(EEPROM) 18

3.6.1 模块工作原理 18

3.6.2 硬件电路的连接 19

3.6.3 库函数配置方法 19
3.7 SDIO 和 MALLOC 19
3.7.1 模块工作原理 19
3.7.2 硬件电路的连接 21
3.7.3 库函数配置方法 22
3.8 FATFS 和 SRAM 22
3.8.1 模块工作原理 22
3.8.2 硬件电路的连接 23
3.8.3 库函数配置方法 23
3.9 KEY 24
3.9.1 模块工作原理 24
3.9.2 硬件电路的连接 24
3.9.3 库函数配置方法 24
3.10 LED 24
3.10.1 模块工作原理 24
3.10.2 硬件电路的连接 25
3.10.3 库函数配置方法 25
3.11 LCD 25
3.11.1 模块工作原理 25
3.11.2 硬件电路的连接 26
3.11.3 库函数配置方法 26
3.12 TOUCH 27
3.12.1 模块工作原理 27
3.12.2 硬件电路的连接 28
3.12.3 库函数配置方法 28
3.13 TPAD 28
3.13.1 模块工作原理 28
3.13.2 硬件电路的连接 29
3.13.3 库函数配置方法 30
3.14 USART 30

3.14.1 模块工作原理 30
3.14.2 硬件电路的连接 30
3.14.3 库函数配置方法 30

四、 具体实现过程描述 31

4.1 CALENDAR ----- 日历和时钟 31
4.1.1 刷新 LCD 屏上的日历----calendar_date_refresh() 31
4.1.2 绘制钟表盘----calendar_circle_clock_drawpanel() 32
4.1.3 绘制时针、分针、秒针----calendar_circle_clock_showtime() 34
4.1.4 显示温度----void temperature_show() 36
4.1.5 整点报时响铃----calendar_ring() 37
4.1.6 运行时钟----calendar_play() 37
4.2 ALARM ----- 闹钟 40
4.2.1 闹钟初始化----alarm_init() 40
4.2.2 闹钟调度----alarm_schedule() 44
4.2.3 闹钟排序----alarm_sort() 46
4.2.4 闹钟比较得到下一个闹钟 id 号----alarm_compare() 47
4.2.5 设置闹钟信息----alarm_Set() 49
4.2.6 闹钟删除----alarm_delete() 50
4.2.7 初始化系统中的闹钟信息----alarm_system() 52
4.2.8 闹钟比较得到是否已有相同的闹钟存在----alarm_equal() 53
4.2.9 获取闹钟信息----alarm_Get() 54
4.2.10 读取闹钟从 EEPROM 中----alarm_read() 55
4.2.11 保存闹钟到 EEPROM 中----alarm_save() 55
4.2.12 闹钟响闹铃----alarm_ring() 56
4.3 FILE ----- FLASH 读取 57
4.3.1 获取特殊字体点阵----font_set() 57
4.4 MP3PLAY ----- MP3 解码播放 59
4.4.1 MP3 音频播放----mp3PlayerDemo() 59
4.5 MAIN ----- 主函数 65

4.6 组员 NOSTEGLIC 负责的 GUI 界面流程图.....66

五、 实现效果.....67

5.1 时钟界面.....67

5.2 修改日期、时间界面.....67

5.3 闹钟设置界面.....68

5.4 串口调试界面.....71

六、 总结.....71

6.1 最终实现的功能.....71

6.2 小组分工.....72

6.3 课程设计中遇到的问题及解决办法.....72

6.3.1 个人遇到的问题.....72

6.3.2 小组共同遇到的问题.....73

6.3.3 待改进的问题.....73

6.4 课程设计的收获与心得.....74

图表目录

图表 1:RTC 模块.....9

图表 2：ADC 模块.....11

图表 3：BEEP 模块.....12

图表 4：I2S 和 WM8978 模块.....14

图表 5：SPI 内部结构图.....16

图表 6：W25Q128 芯片引脚图.....16

图表 7：SPI 模块.....17

图表 8：AT24C02 芯片引脚图.....19

图表 9：IIC 模块.....19

图表 10：分块式内存管理原理 20
图表 11：SDIO 模块 21
图表 12：FATFS 层次结构图 23
图表 13：KEY 模块 24
图表 14：LED 与 STM32F4 连接图 25
图表 15：TFTLCD 使用流程 25
图表 16：4.3' TFTLCD 原理图 26
图表 17：投射式电容屏电极矩阵示意图 27
图表 18：触摸屏与 STM32F4 连接图 28
图表 19：TPAD 按键原理 29
图表 20：TPAD 和 STM32F4 连接图 29
图表 21：USART 连接图 30
图表 22：运行时钟流程图 40
图表 23：闹钟初始化流程图 43
图表 24：闹钟调度流程图 45
图表 25：闹钟排序流程图 47
图表 26：闹钟比较得到下一个闹钟 id 号流程图 48
图表 27：设置闹钟信息流程图 50
图表 28：闹钟删除流程图 51
图表 29：系统存储闹钟信息初始化流程图 53
图表 30：闹钟比较是否有相同闹钟存在流程图 54
图表 31：闹钟响铃流程图 57
图表 32：特殊字体设置流程图 59
图表 33：MP3 播放流程图 65
图表 34：主函数流程图 65
图表 35：GUI 界面流程图 66
图表 36：时钟/设置界面 67
图表 37：修改时间界面 67
图表 38：修改日期界面 68
图表 39：添加第一个闹钟界面 68

图表 40：闹钟信息设置界面 69
图表 41：相同闹钟提醒界面 69
图表 42：删除闹钟提醒界面 70
图表 43：闹钟已满提醒界面 70
图表 44：系统调试界面 71

一、 课程设计目的

1、进一步巩固掌握嵌入式系统课程所学 STM32F4 各功能模块的工作原理；

2、进一步熟练掌握 STM32F4 各功能模块的配置与使用方法；

3、进一步熟练掌握开发环境 Keil MDK5 的使用与程序调试技巧；

4、自学部分功能模块的原理、配置与使用方法，培养自学能力；

5、培养设计复杂嵌入式应用软、硬件系统的分析与设计能力。

二、 课程设计内容

根据所选题目列出具体内容要求：

1、查阅资料，自学 STM32F4 的 RTC 模块，完成 RTC 的配置；

2、查阅资料，学习 STM32F4 与 LCD 的接口设计，完成 LCD 液晶屏驱动程序的设计，将时间、日期、星期等日历信息显示在 LCD 上；

3、能进行正常的日期、时间、星期显示；

4、有校时、校分功能，可以使用按键校时、校分，也可以通过串口调试助手由主机传送时间参数进行校时、校分；

5、能进行整点报时并有闹钟功能，闹钟时间可以设置多个；

6、系统关机后时间能继续运行，下次开机时间应准确；

7、查阅资料，学习 STM32F4 内部温度传感器的配置，采集、计算片内温度并显示在 LCD 上；

8、其他功能，自由发挥扩展。

三、 实验方案分析与设计

3.1 RTC

3.1.1 模块工作原理

STM32F4 的 RTC 是一个独立的 BCD 定时器/计数器，它提供一个日历时钟（包含年月日时分秒信息）、两个可编程闹钟（ALARM A 和 ALARM B）中断和一个具有中断功能的周期性可编程唤醒标志。

RTC 模块和时钟配置是在后备区域，在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变，只要后备区域供电正常，RTC 就会一直运行下去。

1.时钟与分频

我们选择 LSE 作为时钟源（RTCCLK），设置 RTC 的可编程预分配器，提供 1Hz 的时钟。可编程预分配器（RTC_PRER）分为 2 个部分：

（1）7 位异步预分配器，通过 RTC_PRER 寄存器 PREDIV_A 位配置；

（2）15 位同步预分配器，通过 RTC_PRER 寄存器 PREDIV_S 位配置。

利用下面的公式计算得到 1Hz 的F_{ck_spre}：

$$F_{ck\_spre} = \frac{F_{rtcclk}}{(PREDIV\_S + 1) \times (PREDIV\_A + 1)}$$

其中，为了最大程度降低功耗，设置$PREDIV_A = 0X7F$，$PREDIV\_S = 0XFF$。

2.日历时间（RTC_TR）和日期（RTC_DR）寄存器

RTC_TR 和 RTC_DR 分别用来存储/设置时间和日期。每隔 2 个 RTCCLK 周期，当前日历值就会复制到影子寄存器，并置位 RTC_ISR 寄存器的 RSF 位。读取 RTC_TR 和 RTC_DR 可获得当前日历和日期的 BCD 码信息，需要进行相应转换得到十进制数据。

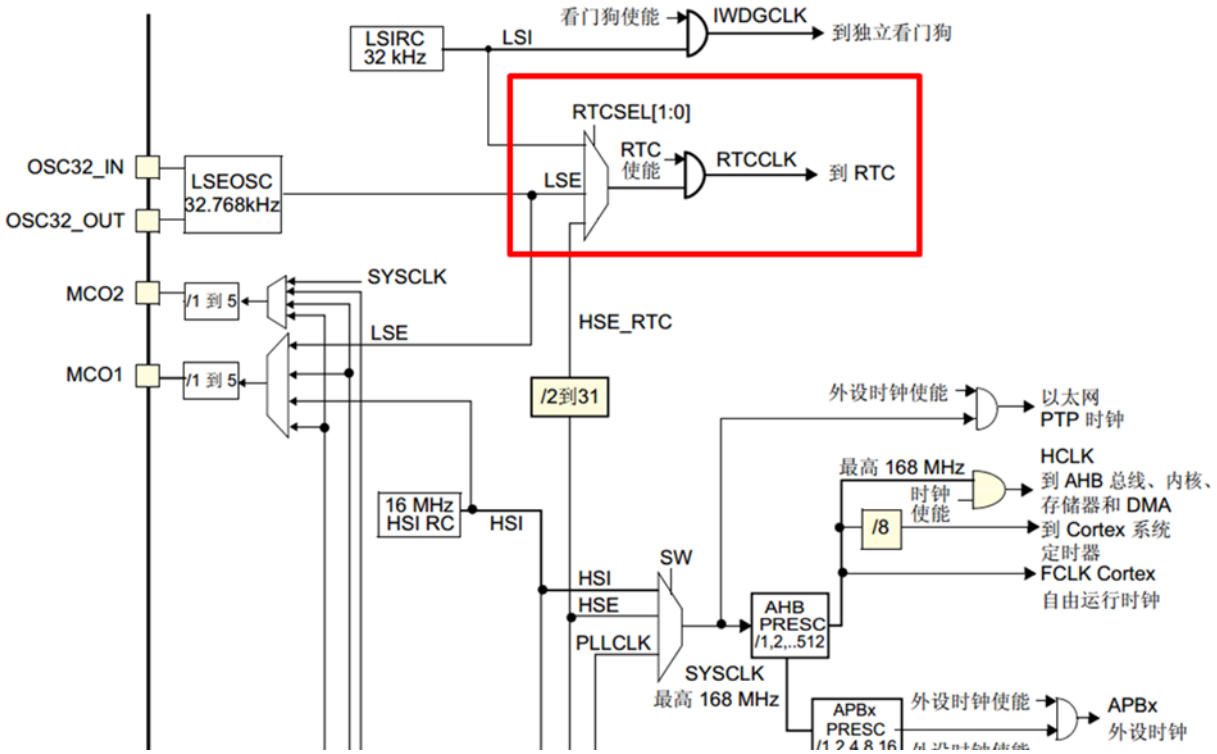
3.闹钟模块

RTC 单元提供两个可编程闹钟，即闹钟 A 和闹钟 B。通过 RTC_CR 寄存器的 ALRAE 和 ALRBE 位置 1 来使能可编程闹钟功能。当日历的秒、分、小时、日期分别与闹钟寄存器 RTC_ALRMASSR/RTC_ALRMAR 和 RTC_ALRMBSSR/RTC_ALRMBR 中的值匹配时，则可以产生闹钟（需要适当配置）。本课程设计我们将利用闹钟 A 产生闹铃中断。

4.周期性自动唤醒模块

周期性唤醒标志由一个 16 位可编程自动重载递减计数器生成。唤醒定时器范围可扩展至 17 位。通过将 RTC_CR 寄存器中的 WUTIE 位置 1 来使能周期性唤醒中断时，可以使 STM32F4 退出低功耗模式。系统复位以及低功耗模式（睡眠、停机和待机）对唤醒定时器没有任何影响，它仍然可以正常工作，所以唤醒定时器，可以用于周期性唤醒 STM32F4。

3.1.2 硬件电路的连接



图表 1:RTC 模块

3.1.3 库函数配置方法

（1）RTC 日历配置一般步骤

1° 使能 PWR 时钟

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);
```

2° 使能后备寄存器写访问

```
PWR_BackupAccessCmd(ENABLE);
```

3° 配置 RTC 时钟源，使能 RTC 时钟：

```
RCC_LSEConfig(RCC_LSE_ON);          //需要使用 LSE，所以必须开启 LSE
RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);
RCC_RTCCLKCmd(ENABLE);
```

4° 初始化 RTC

```
RTC_Init(&RTC_InitStructure);
//同步/异步分频系数和时钟格式
//RTC_InitStructure.RTC_AsynchPrediv = 0x7F;//RTC 异步分频系数(1~0X7F)
//RTC_InitStructure.RTC_SynchPrediv = 0xFF;//RTC 同步分频系数(0~7FFF)
//RTC_InitStructure.RTC_HourFormat = RTC_HourFormat_24;//RTC 设置为,24 小时格式
```

5° 设置时间

```
RTC_SetTime();
```

6° 设置日期

```
RTC_SetDate();
```

（2）RTC 闹钟配置步骤

1° RTC 初始化好相关参数；

2° 关闭闹钟；

```
RTC_AlarmCmd(RTC_Alarm_A,DISABLE);          //关闭 RTC 闹钟服务
```

3° 配置闹钟参数；

```
RTC_SetAlarm();                             //初始化闹钟参数
```

4° 开启闹钟；

```
RTC_AlarmCmd(RTC_Alarm_A,EABLE);            //开启 RTC 闹钟服务
```

5° 开启配置闹钟中断；

```
RTC_ITConfig();                             //开启 RTC 中断
EXTI_Init();                                //配置中断线等
NVIC_Init();                                //配置中断优先级等
```

6° 编写中断服务函数；

```
RTC_Alarm_IRQHandler();                     //RTC 闹钟中断服务函数
```

（3）RTC 周期性自动唤醒配置步骤

1° RTC 初始化好相关参数；

2° 关闭 WakeUp；

```
RTC_WakeUpCmd(DISABLE);                     //关闭 RTC 周期性自动唤醒服务
```

3° 配置 WakeUp 时钟分频系数/来源

```
RTC_WakeUpClockConfig();                    //配置 RTC 的 WakeUp 参数
```

4° 设置 WakeUp 自动装载寄存器；

```
RTC_SetWakeUpCounter();                     //配置 WakeUp 自动装载寄存器
```

5° 使能 WakeUp；

```
RTC_WakeUpCmd( ENABLE);                     //开启 RTC 周期性自动唤醒服务
```

6° 开启配置闹钟中断：

```
RTC_ITConfig();                             //开启 RTC 中断
EXTI_Init();                                //配置中断线等
NVIC_Init();                                //配置中断优先级等
```

7° 编写中断服务函数；

```
RTC_WKUP_IRQHandler();                      //RTC 的 WakeUp 中断服务函数
```

3.2 ADC

3.2.1 模块工作原理

STM32F4 有一个内部的温度传感器，可以用来测量 CPU 及周围的温度(TA)。该温度传感器在内部和 ADC1_IN16（STM32F40xx/F41xx 系列）或 ADC1_IN18（STM32F42xx/F43xx 系列）输入通道相连接，此通道把传感器输出的电压转换成数字值。STM32F4 的内部温度传感器支持的温度范围为：-40~125 度。精度为±1.5℃左右。

STM32F407ZGT6 的内部温度传感器固定的连接在 ADC1 的通道 16 上，所以在设置好 ADC1 之后只要读取通道 16 的值，就是温度传感器返回来的电压值了。根据这个值，我们就可以按照以下公式计算出当前温度。

$$T(℃) = (V_{sense} - V_{25})/Avg_{Slope} + 25$$

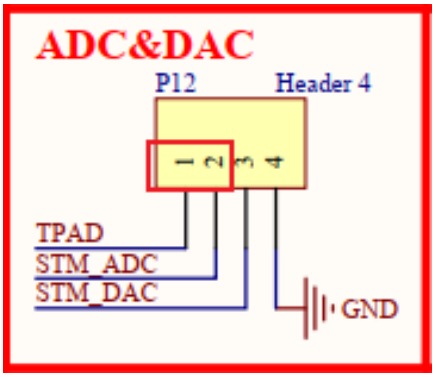
其中：

（1）$V_{25} = V_{sense}$在 25 度时的数值（典型值为：0.76）；

（2）Avg_{Slope} =温度与V_{sense}曲线的平均斜率（单位为 mv/℃或 uv/℃）；

3.2.2 硬件电路的连接

ADC 属于 STM32F4 内部资源，实际上我们只需要软件设置就可以正常工作。



图表 2：ADC 模块

3.2.3 库函数配置方法

1° 开启 PA 口时钟和 ADC1 时钟，设置 PA1（ADC1 通道 1） 为模拟输入。

```
RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOA, ENABLE);      //使能 GPIOA 时钟
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);        //使能 ADC1 时钟
GPIO_Init();                                                //初始化 IO 口
```

2° 复位 ADC1。

```
RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC1,ENABLE);          //ADC1 复位
RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC1,DISABLE);         //ADC1 复位结束
```

3° 初始化 ADC_CCR 寄存器，同时设置 ADC1 分频因子

```
ADC_CommonInit();                                            //设置 ADC1 分频数
```

4° 初始化 ADC1 参数，设置 ADC1 的工作模式以及规则序列的相关信息。

```
void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct)；//初始化参数
```

5° 使能 ADC

```
ADC_Cmd(ADC1, ENABLE);                                       // 开启 AD 转换器
```

6° 配置规则通道参数：

```
ADC_RegularChannelConfig();                                  //配置通道参数
```

7° 软件开启转换

```
ADC_SoftwareStartConv(ADC1);                                 //开启 ADC1 转换
```

8° 等待转换完成， 读取 ADC 值

```
ADC_GetConversionValue(ADC1);                                //读取 ADC 值
```

9° 启动内部温度传感器

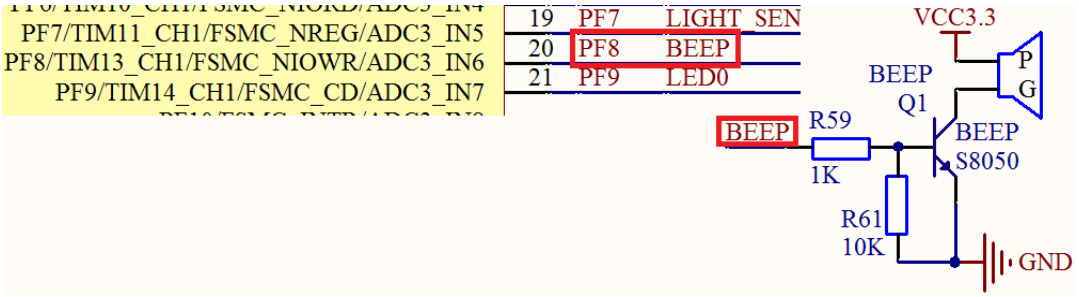
```
ADC_TempSensorVrefintCmd(ENABLE);                            //使能内部温度传感器
```

3.3 BEEP

3.3.1 模块工作原理

蜂鸣器是一种一体化结构的电子讯响器。探索者 STM32F4 开发板板载的蜂鸣器是电磁式的有源蜂鸣器。我们是通过三极管扩流后再驱动蜂鸣器，这样 STM32F4 的 IO 只需要提供不到 1mA 的电流就足够了。

3.3.2 硬件电路的连接



图表 3：BEEP 模块

3.3.3 库函数配置方法

1° 使能 GPIO 时钟

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE);          //使能 GPIOF 时钟
```

2° 蜂鸣器 GPIO 引脚初始化

```
GPIO_Init(GPIOF, &GPIO_InitStructure);                          //初始化 GPIO
```

3° 初始关闭蜂鸣器

```
GPIO_ResetBits(GPIOF,GPIO_Pin_8);                               //BEEP 对应引脚拉低
```

3.4 I2S 和 WM8978

3.4.1 模块工作原理

（1）I2S 模块

I2S(Inter IC Sound)总线，又称集成电路内置音频总线，是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专责于音频设备之间的数据传输，广泛应用于各种多媒体系统。它采用了沿独立的导线传输时钟与数据信号的设计，通过将数据和时钟信号分离，避免了因时差诱发的失真，为用户节省了购买抵抗音频抖动的专业设备的费用。

STM32F4 自带了 2 个全双工 I2S 接口，其特点包括：

① 支持全双工/半双工通信

② 主持主/从模式设置

③ 8 位可编程线性预分频器，可实现精确的音频采样频率(8~192Khz)

④ 支持 16 位/24 位/32 位数据格式

⑤ 数据包帧固定为 16 位（仅 16 位数据帧）或 32 位（可容纳 16/24/32 位数据帧）

⑥ 可编程时钟极性

⑦ 支持 MSB 对齐（左对齐）、 LSB 对齐（右对齐）、飞利浦标准和 PCM 标准等 I2S 协议

⑧ 支持 DMA 数据传输（16 位宽）

⑨ 数据方向固定位 MSB 在前

⑩ 支持主时钟输出（固定为 256*fs， fs 即音频采样率）

（2）WM8978 模块

WM8978 是欧胜（Wolfson） 推出的一款全功能音频处理器。它带有一个 HI-FI 级数字信号处理内核，支持增强 3D 硬件环绕音效，以及 5 频段的硬件均衡器，可以有

效改善音质；并有一个可编程的陷波滤波器，用以去除屏幕开、切换等噪音。

WM8978 的主要特性有：

① I2S 接口，支持最高 192K,24bit 音频播放

② DAC 信噪比 98dB； ADC 信噪比 90dB

③ 支持无电容耳机驱动（提供 40mW@16Ω的输出能力）

④ 支持扬声器输出（提供 0.9W@8Ω的驱动能力）

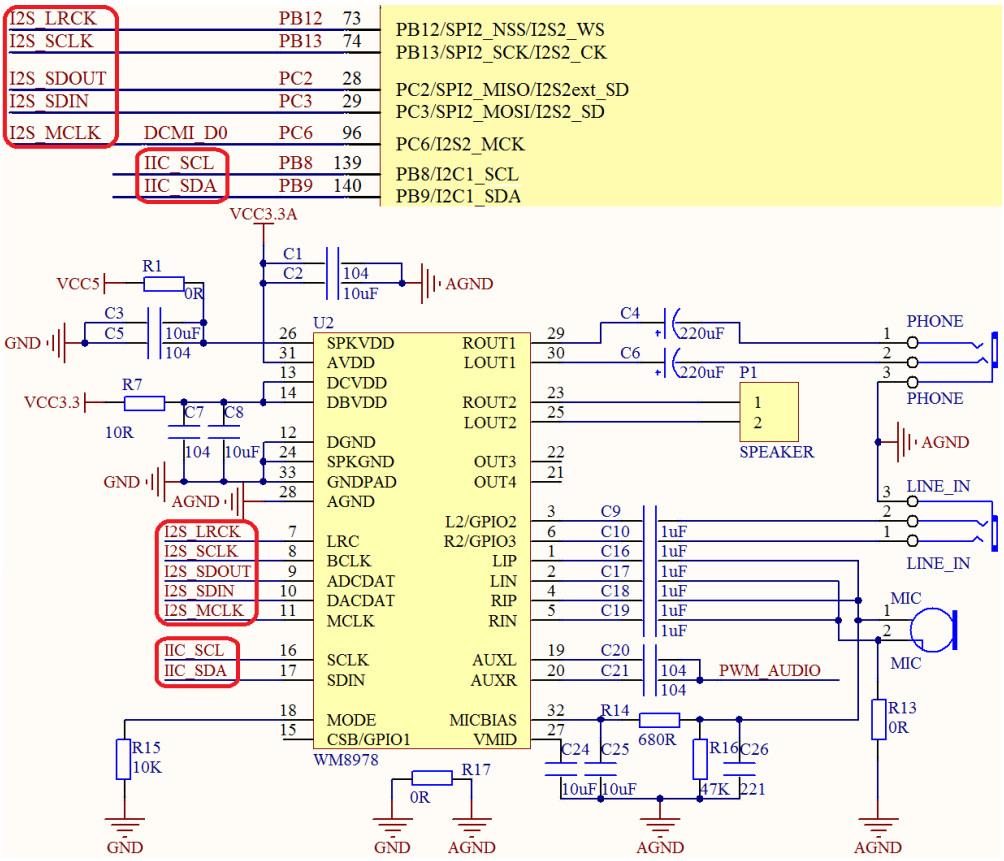
⑤ 支持立体声差分输入/麦克风输入

⑥ 支持左右声道音量独立调节

⑦ 支持 3D 效果，支持 5 路 EQ 调节

WM8978 的控制通过 I2S 接口（即数字音频接口）同 MCU 进行音频数据传输（支持音频接收和发送），通过两线（MODE=0，即 IIC 接口）或三线（MODE=1）接口进行配置。 WM8978 的 I2S 接口，由 4 个引脚组成：①ADCDAT:ADC 数据输出②DACDAT： DAC 数据输入③LRC：数据左/右对齐时钟④BCLK：位时钟，用于同步

3.4.2 硬件电路的连接



图表 4：I2S 和 WM8978 模块

3.4.3 库函数配置方法

1° 使能时钟

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC, ENABLE);
//使能外设 GPIOB,GPIOC 时钟
```

2° PB12/13 复用功能输出、PC2/PC3/PC6 复用功能输出

```
GPIO_Init(GPIOB, &GPIO_InitStructure);          //初始化
GPIO_Init(GPIOC, &GPIO_InitStructure);          //初始化
GPIO_PinAFConfig( );                            //复用
```

3° 初始化 IIC

```
IIC_Init();                                     //初始化 IIC 接口
```

4° 软复位

```
WM8978_Write_Reg(0,0);                          //软复位 WM8978
```

5° WM8978 通用设置

```
WM8978_Write_Reg();                             //WM8978 配置
```

6° 音量设置

```
WM8978_HPvol_Set(40,40);                        //耳机音量设置
WM8978_SPKvol_Set(50);                          //喇叭音量设置
```

3.5 SPI 和 W25Q128(FLASH)

3.5.1 模块工作原理

（1）SPI 模块

SPI （Serial Peripheral interface）即串行外围设备接口。 是 Motorola 首先在其 MC68HCXX 系列处理器上定义的。

SPI 是一种高速、 全双工、 同步串行通信总线，并且在芯片的管脚上只占用四根线，节约了芯片的管脚，同时为 PCB 的布局上节省空间，提供方便。

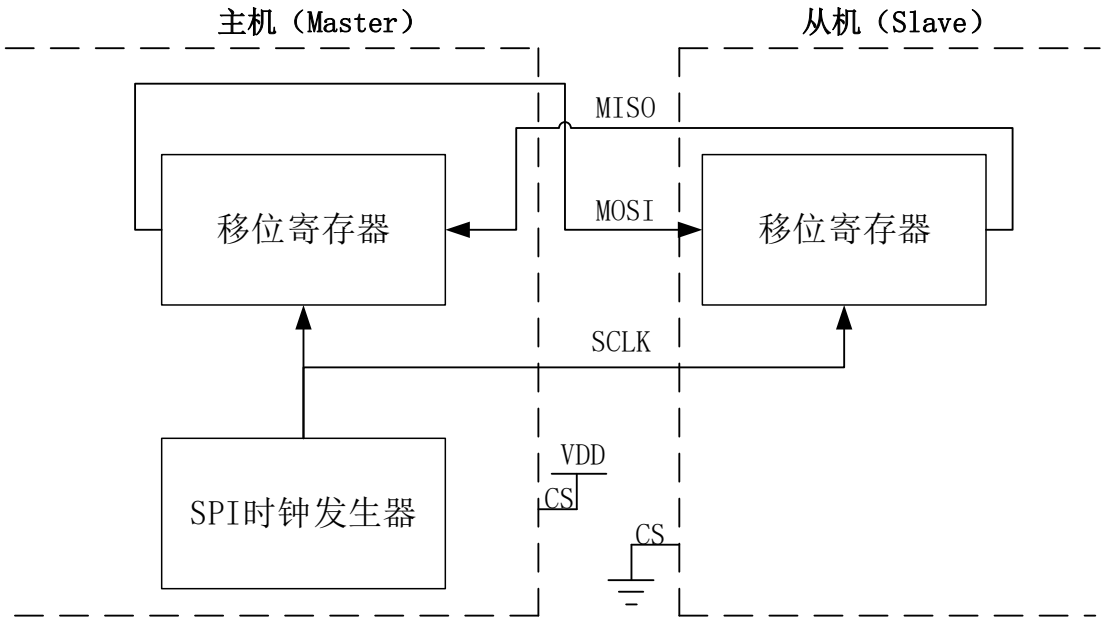
SPI 接口一般使用 4 条线通信：

1、MISO 主设备数据输入，从设备数据输出。

2、MOSI 主设备数据输出，从设备数据输入。

3、SCLK 时钟信号，由主设备产生。

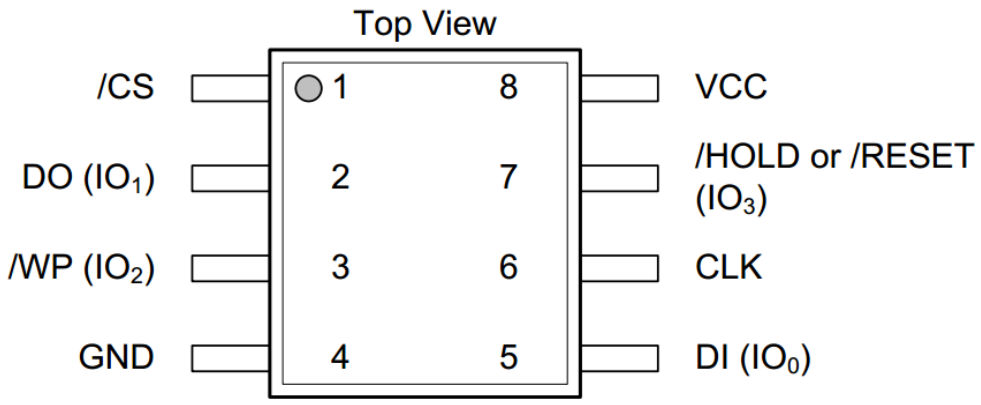
4、CS 从设备片选信号，由主设备控制。



图表 5：SPI 内部结构图

（2）W25Q128 模块

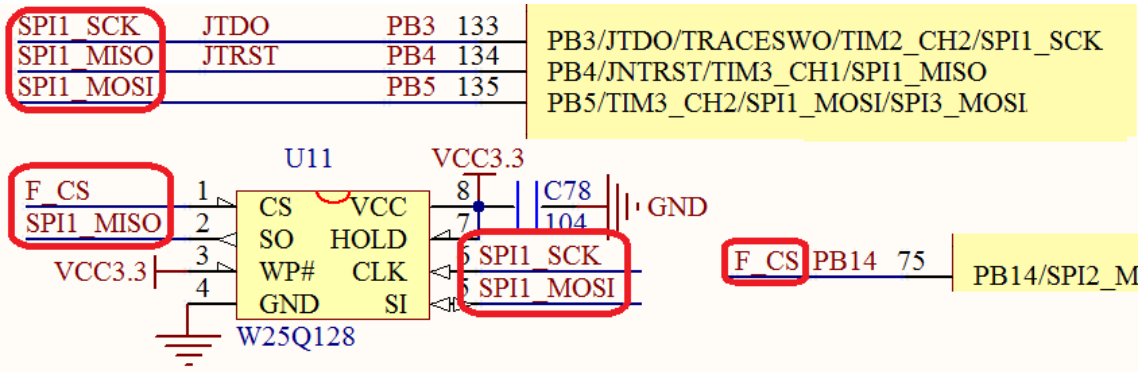
在本课程设计中，我们将使用 STM32F4 的 SPI 来读取外部 SPI FLASH 芯片 W25Q128。



图表 6：W25Q128 芯片引脚图

FLASH W25QXX 的页写是可以跨页的，与擦除指令不一样（整页擦除），页写入指令的地址并不要求按 256 字节对齐，只要确认目标存储单元是擦除状态即可(即被擦除后没有被写入过)。所以，若对“地址 x”执行页写入指令后，发送了 200 个字节数据后终止通讯，下一次再执行页写入指令，从“地址(x+200)”开始写入 200 个字节也是没有问题的(小于 256 均可)。只是在实际应用中由于基本擦除单元是 4KB，一般都以扇区为单位进行读写。

3.5.2 硬件电路的连接



图表 7：SPI 模块

3.5.3 库函数配置方法

（1）SPI 模块配置的步骤

1° 使能 SPIx 和 IO 口时钟

```
RCC_AHBxPeriphClockCmd();                    //使能 GPIOA 时钟
RCC_APBxPeriphClockCmd();                    //使能 SPI1 时钟
```

2° 初始化 IO 口为复用功能

```
void GPIO_Init();                            //GPIOF9,F10 初始化设置
```

3° 设置引脚复用映射：

```
GPIO_PinAFConfig();                          //引脚复用
```

4° 初始化 SPIx，设置 SPIx 工作模式

```
void SPI_Init(SPI_TypeDef* SPIx, SPI_InitTypeDef* SPI_InitStruct);//SPI 初始化
```

5° SPI 口初始化

```
RCC_APB2PeriphResetCmd(RCC_APB2Periph_SPI1,ENABLE);   //复位 SPI1
RCC_APB2PeriphResetCmd(RCC_APB2Periph_SPI1,DISABLE);  //停止复位 SPI1
```

6° 使能 SPIx

```
void SPI_Cmd(SPI_TypeDef* SPIx, FunctionalState NewState);      //SPI 使能
```

7° SPI 传输数据

```
void SPI_I2S_SendData(SPI_TypeDef* SPIx, uint16_t Data);        //发送数据
uint16_t SPI_I2S_ReceiveData(SPI_TypeDef* SPIx) ;               //接收数据
```

（2）W25Q128 芯片模块配置步骤

1° 使能时钟；

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);      //使能 GPIOB 时钟
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);      //使能 GPIOG 时钟
```

2° GPIO 初始化；

```
GPIO_Init(GPIOB, &GPIO_InitStructure);                    //初始化
```

3°防止 WIRELESS 影响我们芯片数据的读写；

```
GPIO_SetBits(GPIOG,GPIO_Pin_7);          //PG7 输出 1,防止 NRF 干扰 SPI FLASH 的通信
```

4°初始化 SPI；

```
W25QXX_CS=1;                             //SPI FLASH 不选中
SPI1_Init();                             //初始化 SPI
SPI1_SetSpeed(SPI_BaudRatePrescaler_2);  //设置为 42M 时钟,高速模式
W25QXX_TYPE=W25QXX_ReadID();             //读取 FLASH ID.
```

3.6 I2C 和 AT24C02(EEPROM)

3.6.1 模块工作原理

（1）IIC 模块

IIC(Inter－Integrated Circuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。它是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。在 CPU 与被控 IC 之间、 IC 与 IC 之间进行双向传送， 高速 IIC 总线一般可达 400kbps 以上。

I2C 总线在传送数据过程中共有三种类型信号， 它们分别是：开始信号、结束信号和应答信号。

1、开始信号： SCL 为高电平时， SDA 由高电平向低电平跳变，开始传送数据。

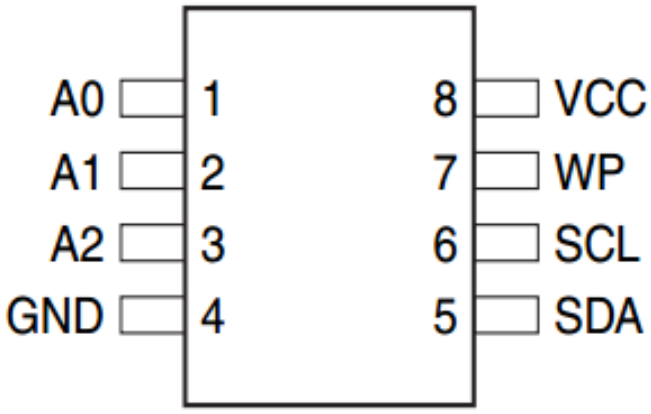
2、结束信号： SCL 为高电平时， SDA 由低电平向高电平跳变，结束传送数据。

3、应答信号：接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。 CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号， CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

（2）AT24C02 模块

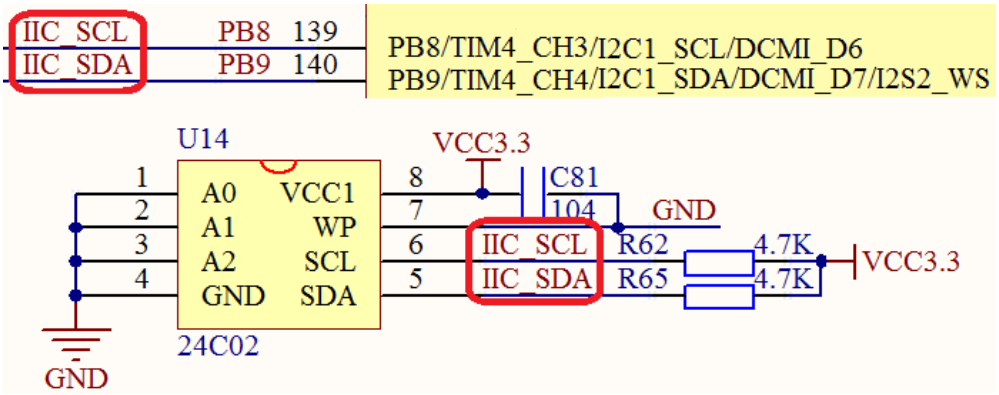
ALIENTEK 探索者 STM32F4 开发板板载的 EEPROM 芯片型号为 24C02。该芯片的总容量是 256 个字节，该芯片通过 IIC 总线与外部连接。

I2C 的 AC24CXX 是不可以跨页写入的，地址是要对齐 8 能整除的。



图表 8：AT24C02 芯片引脚图

3.6.2 硬件电路的连接



图表 9：IIC 模块

3.6.3 库函数配置方法

1° 使能时钟

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);          //使能 GPIOB 时钟
```

2° GPIOB8,B9 初始化设置

```
GPIO_Init(GPIOB, &GPIO_InitStructure);                          //初始化
```

3° 空闲状态

```
IIC_SCL=1;                                                      //空闲状态
IIC_SDA=1;
```

3.7 SDIO 和 MALLOC

3.7.1 模块工作原理

（1）SDIO 模块

ALIENTEK 探索者 STM32F4 开发板自带 SDIO 接口。

STM32F4 的 SDIO 控制器支持多媒体卡（MMC 卡）、SD 存储卡、SD I/O 卡和 CE-ATA 设备等。SDIO 的主要功能如下：

①与多媒体卡系统规格书版本 4.2 全兼容。支持三种不同的数据总线模式：1 位(默认)、4 位和 8 位。

②与较早的多媒体卡系统规格版本全兼容(向前兼容)。

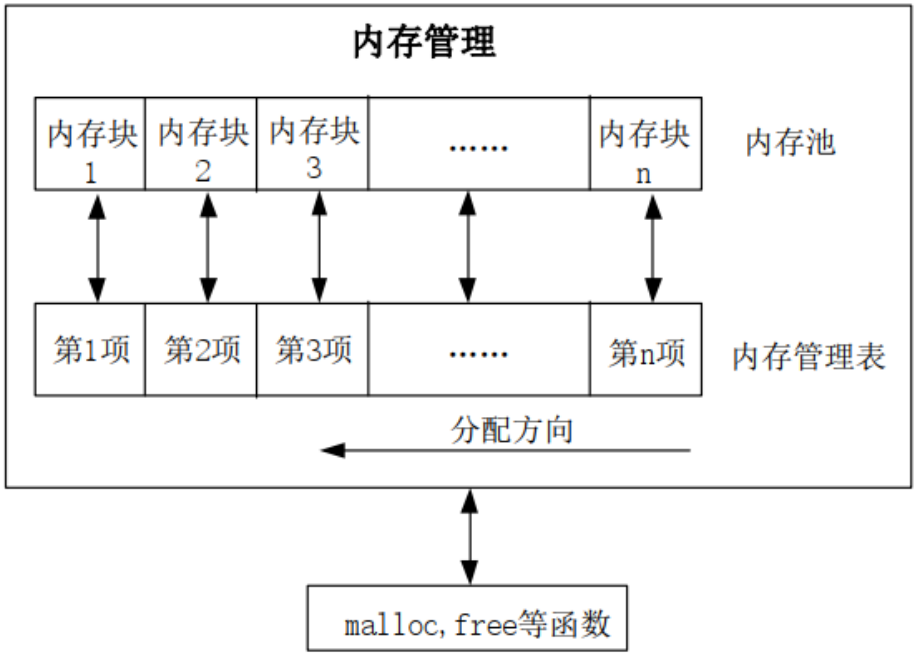
③与 SD 存储卡规格版本 2.0 全兼容。

④与 SD I/O 卡规格版本 2.0 全兼容：支持良种不同的数据总线模式：1 位(默认)和 4 位。

⑤完全支持 CE-ATA 功能(与 CE-ATA 数字协议版本 1.1 全兼容)。8 位总线模式下数据传输速率可达 48MHz（分频器旁路时）。

⑥数据和命令输出使能信号，用于控制外部双向驱动器。

（2）MALLOC 模块



图表 10：分块式内存管理原理

从上图可以看出，分块式内存管理由内存池和内存管理表两部分组成。内存池被等分为 n 块，对应的内存管理表，大小也为 n，内存管理表的每一个项对应内存池的一块内存。

内存管理表的项值代表的意义为：当该项值为 0 的时候，代表对应的内存块未被占

用，当该项值非零的时候，代表该项对应的内存块已经被占用，其数值则代表被连续占用的内存块数。比如某项值为 10，那么说明包括本项对应的内存块在内，总共分配了 10 个内存块给外部的某个指针。

内寸分配方向如图所示，是从顶→底的分配方向。即首先从最末端开始找空内存。当内存管理刚初始化的时候，内存表全部清零，表示没有任何内存块被占用。

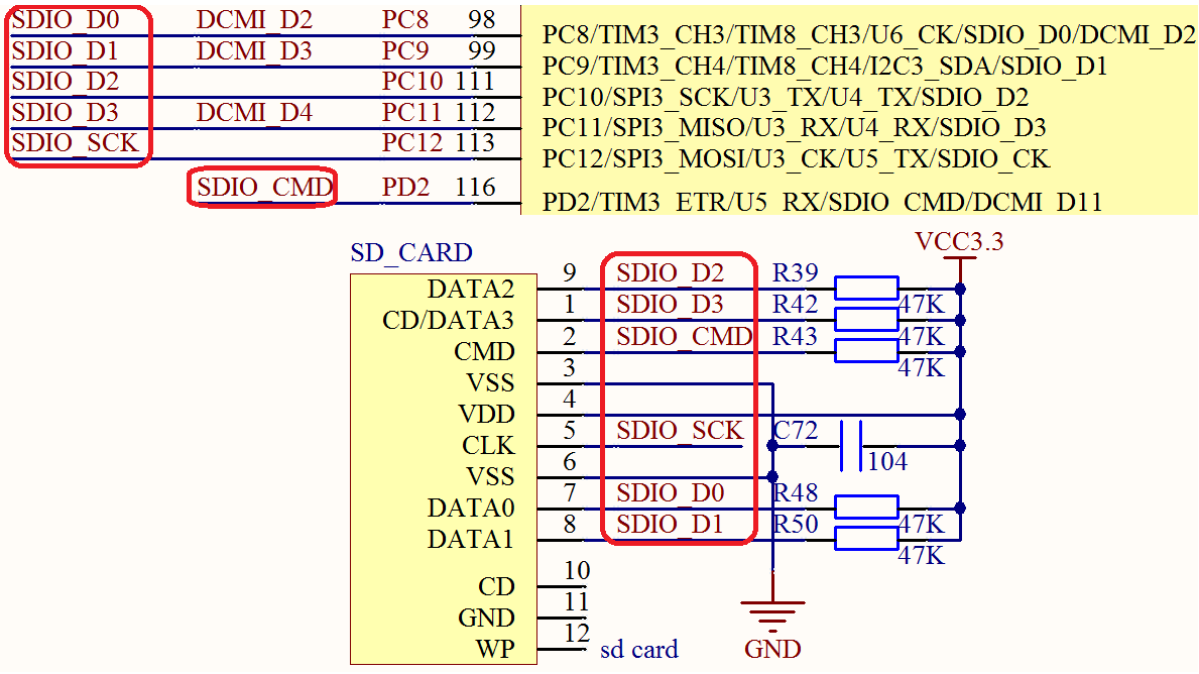
分配原理：

当指针 p 调用 malloc 申请内存的时候，先判断 p 要分配的内存块数（m），然后从第 n 项开始，向下查找，直到找到 m 块连续的空内存块（即对应内存管理表项为 0），然后将这 m 个内存管理表项的值都设置为 m（标记被占用），最后，把最后的这个空内存块的地址返回指针 p，完成一次分配。注意，如果当内存不够的时候（找到最后也没找到连续的 m 块空闲内存），则返回 NULL 给 p，表示分配失败。

释放原理：

当 p 申请的内存用完，需要释放的时候，调用 free 函数实现。free 函数先判断 p 指向的内存地址所对应的内存块，然后找到对应的内存管理表项目，得到 p 所占用的内存块数目 m（内存管理表项目的值就是所分配内存块的数目），将这 m 个内存管理表项目的值都清零，标记释放，完成一次内存释放。

3.7.2 硬件电路的连接



图表 11：SDIO 模块

3.7.3 库函数配置方法

（1）SD 卡模块配置

1° 使能时钟

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC|RCC_AHB1Periph_GPIOD|RCC_AHB1Periph_DMA2, ENABLE);  //使能 GPIOC,GPIOD DMA2 时钟
```

2° SDIO 复位

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SDIO, ENABLE);    //SDIO 时钟使能
RCC_APB2PeriphResetCmd(RCC_APB2Periph_SDIO, ENABLE);    //SDIO 复位
```

3° GPIO 初始化

```
GPIO_Init(GPIOC, &GPIO_InitStructure);    // PC8,9,10,11,12 复用功能输出
GPIO_PinAFConfig();                       //引脚复用
```

4° SDIO 结束复位

```
RCC_APB2PeriphResetCmd(RCC_APB2Periph_SDIO, DISABLE);//SDIO 结束复位
SDIO_Register_Deinit();                   //SDIO 外设寄存器设置为默认值
```

5° NVIC 初始化

```
NVIC_Init(&NVIC_InitStructure);           //根据指定的参数初始化 NVIC 寄存器
```

6° SD 卡上电

```
errorstatus=SD_PowerON();                 //SD 卡上电
```

（2）MALLOC 内存初始化

```
my_mem_init(SRAMIN);                      //初始化内部内存池
my_mem_init(SRAMEX);                      //初始化外部内存池
my_mem_init(SRAMCCM);                     //初始化 CCM 内存池
```

3.8 FATFS 和 SRAM

3.8.1 模块工作原理

FATFS 是一个完全免费开源的 FAT 文件系统模块，专门为小型的嵌入式系统而设计。它完全用标准 C 语言编写，所以具有良好的硬件平台独立性，可以移植到 8051、PIC、 AVR、 SH、Z80、 H8、 ARM 等系列单片机上而只需做简单的修改。它支持 FATl2、 FATl6 和 FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读 / 写，并特别对 8 位单片机和 16 位单片机做了优化。

FATFS 的特点有：

① Windows 兼容的 FAT 文件系统（支持 FAT12/FAT16/FAT32）

② 与平台无关，移植简单

③ 代码量少、效率高

④ 多种配置选项：

a) 支持多卷（物理驱动器或分区，最多 10 个卷）

b) 多个 ANSI/OEM 代码页包括 DBCS

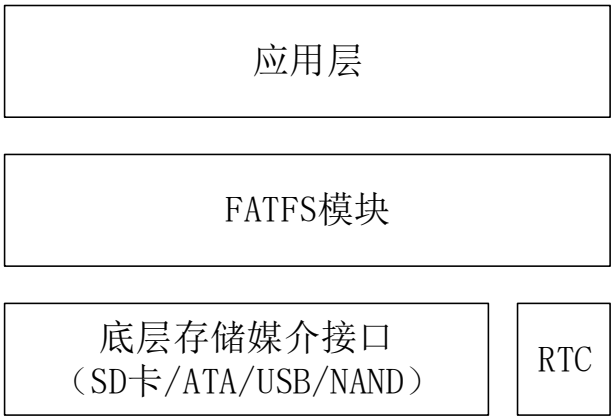
c) 支持长文件名、 ANSI/OEM 或 Unicode

d) 支持 RTOS

e) 支持多种扇区大小

f) 只读、最小化的 API 和 I/O 缓冲区等

FATFS 的这些特点，加上免费、开源的原则，使得 FATFS 应用非常广泛。 FATFS 模块的层次结构如下图所示：



图表 12：FATFS 层次结构图

3.8.2 硬件电路的连接

该文件系统主要是与 SD 卡、SPI FLASH 相关的，所以除 SPI 外无其他硬件连接。

3.8.3 库函数配置方法

（1）FATFS 申请内存

```
u8 exfuns_init(void);                                    //为 exfuns 申请内存
```

（2）初始化外部 SRAM

1° 使能时钟

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOD|RCC_AHB1Periph_GPIOE|RCC_AHB1Periph_GPIOF|RCC_AHB1Periph_GPIOG, ENABLE);
//使能 PD,PE,PF,PG 时钟
RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC,ENABLE);          //使能 FSMC 时钟
```

2° GPIO 初始化

```
GPIO_Init();                                    //初始化
GPIO_PinAFConfig();                             //复用
```

3° 初始化 FSMC

```
FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);          //初始化 FSMC 配置
FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM3, ENABLE);          //使能 BANK3
```

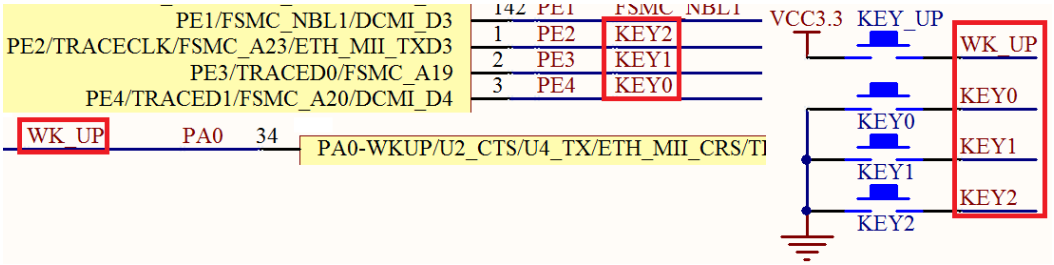
3.9 KEY

3.9.1 模块工作原理

我们可以通过查询或中断方式来使用实验板上的 4 个按键，分别为 KEY0、KEY1、KEY2、KEY_UP。

本课程设计我们将使用查询方式使用按键。

3.9.2 硬件电路的连接



图表 13：KEY 模块

3.9.3 库函数配置方法

1° 使能时钟

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOE, ENABLE);
//使能 GPIOA,GPIOE 时钟
```

2° GPIO 初始化

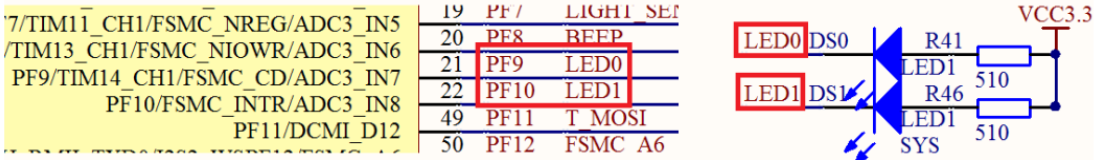
```
GPIO_Init(GPIOE, &GPIO_InitStructure);                 //初始化 GPIOE2,3,4
```

3.10 LED

3.10.1 模块工作原理

LED0 和 LED1 分别对应 IO 口 F9 和 F10。将 F9、F10 位设置为 0，灯就亮；设置为 1，灯就灭。

3.10.2 硬件电路的连接



图表 14：LED 与 STM32F4 连接图

3.10.3 库函数配置方法

1° 使能 GPIOF 时钟

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE);
```

2° 初始化 GPIO

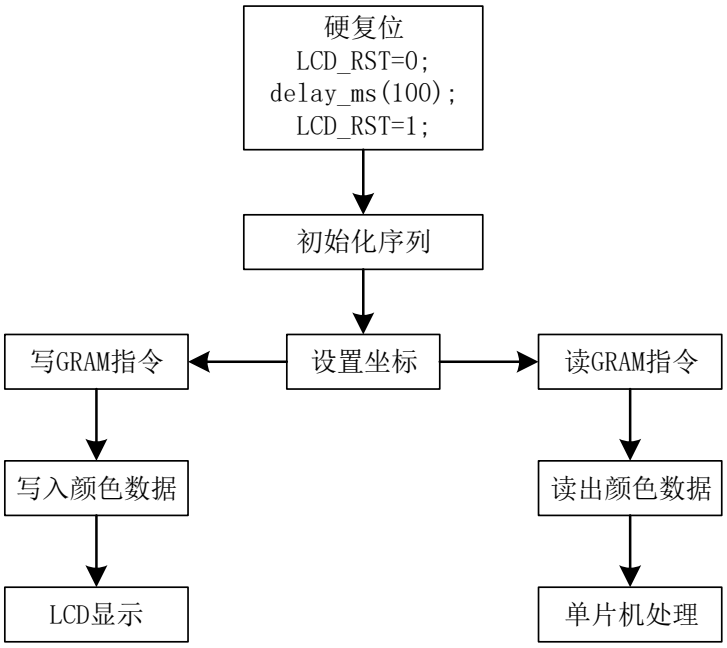
```
GPIO_Init(GPIOF, &GPIO_InitStructure);
```

3.11 LCD

3.11.1 模块工作原理

TFT-LCD 即薄膜晶体管液晶显示器，它在液晶显示屏的每一个像素上都设置有一个薄膜晶体管（TFT），可有效地克服非选通时的串扰，使显示液晶屏的静态特性与扫描线数无关，因此大大提高了图像质量。所用 LCD 的颜色数据为 16 位，最低 5 位代表蓝色，中间 6 位为绿色，最高 5 位为红色。数值越大，表示该颜色越深。

TFTLCD 的通用使用流程如下：



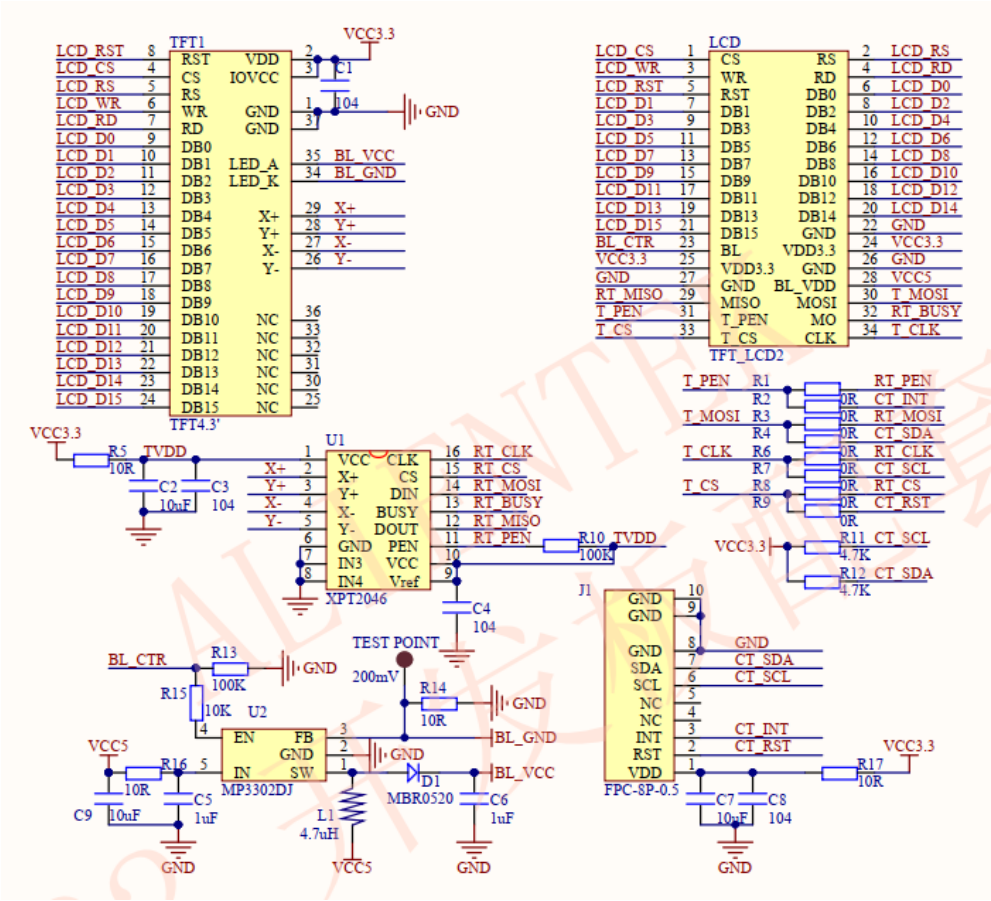
图表 15：TFTLCD 使用流程

其中，硬复位和初始化序列只需要执行一次即可。

画点流程为：设置坐标→写 GRAM 指令→写入颜色数据。

读点流程为：设置坐标→读 GRAM 指令→读取颜色数据。

3.11.2 硬件电路的连接



图表 16：4.3' TFTLCD 原理图

3.11.3 库函数配置方法

1° GPIO，FSMC 时钟使能

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOD|RCC_AHB1Periph_GPIOE|RCC_AHB1Periph_GPIOF|RCC_AHB1Periph_GPIOG, ENABLE);
//使能 PD,PE,PF,PG 时钟
RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC,ENABLE);//使能 FSMC 时钟
```

2° GPIO 初始化设置

```
GPIO_Init();
```

3° 引脚复用映射设置

```
GPIO_PinAFConfig();
```

4° FSMC 初始化

```
FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
```

5° 使能 FSMC

```
FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM4, ENABLE);
```

6° 不同的 LCD 驱动器不同的初始化设置

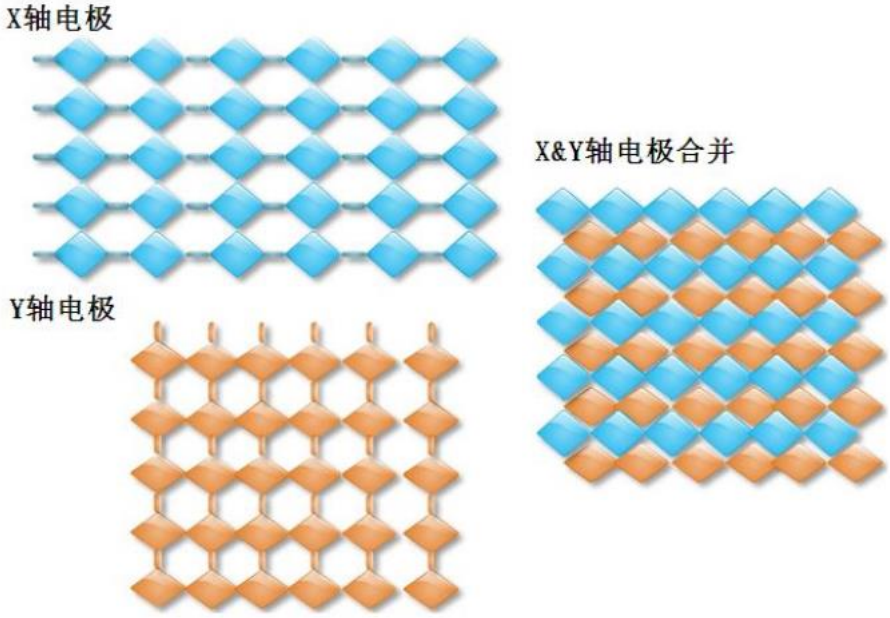
```
根据读到的 LCD ID，对不同的驱动器执行不同的初始化代码。
```

3.12 TOUCH

3.12.1 模块工作原理

本次实验使用的 4.3 寸 TFTLCD 模块自带的触摸屏，采用的是投射式电容触摸屏（交互电容类型）。它是在玻璃表面的横向和纵向的 ITO 电极的交叉处形成电容。交互电容的扫描方式就是扫描每个交叉处的电容变化，来判定触摸点的位置。当触摸的时候就会影响到相邻电极的耦合，从而改变交叉处的电容量，交互电容的扫面方法可以侦测到每个交叉点的电容值和触摸后电容变化，因而它需要的扫描时间与自我电容的扫描方式相比要长一些，需要扫描检测 X×Y 根电极。

电容触摸屏对工作环境的要求是比较高的，在潮湿、多尘、高低温环境下面，都是不适合使用电容屏的。



图表 17：投射式电容屏电极矩阵示意图

其中，图中的电极实际是透明的，X、Y 轴的透明电极电容屏的精度、分辨率与 X、

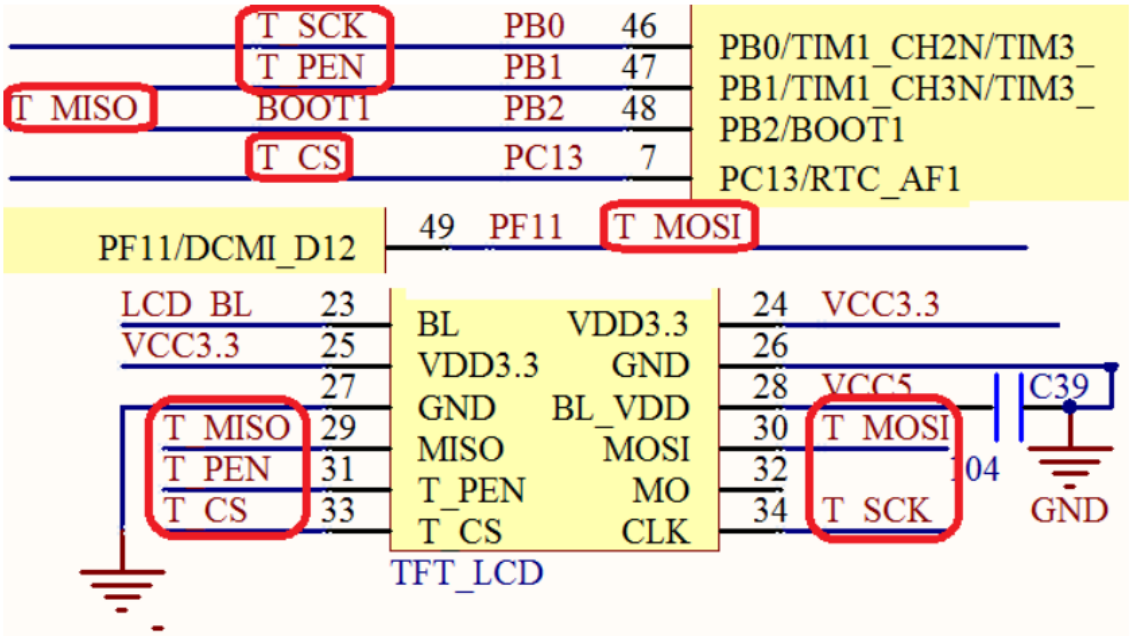
Y 轴的通道数有关。通道数越多，精度越高。

电容触摸屏的优点： 手感好、无需校准、支持多点触摸、透光性好。

电容触摸屏的缺点：成本高、精度不高、抗干扰能力差。

本实验为了简化编程复杂度，设置单点触摸。

3.12.2 硬件电路的连接



图表 18：触摸屏与 STM32F4 连接图

3.12.3 库函数配置方法

1° GPIO 时钟使能

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC|RCC_AHB1Periph_GPIOF, ENABLE);//使能 GPIOB,C,F 时钟
```

2° GPIO 初始化

```
GPIO_Init();
```

3° 第一次读取初始化

```
TP_Read_XY(&tp_dev.x[0],&tp_dev.y[0]);
```

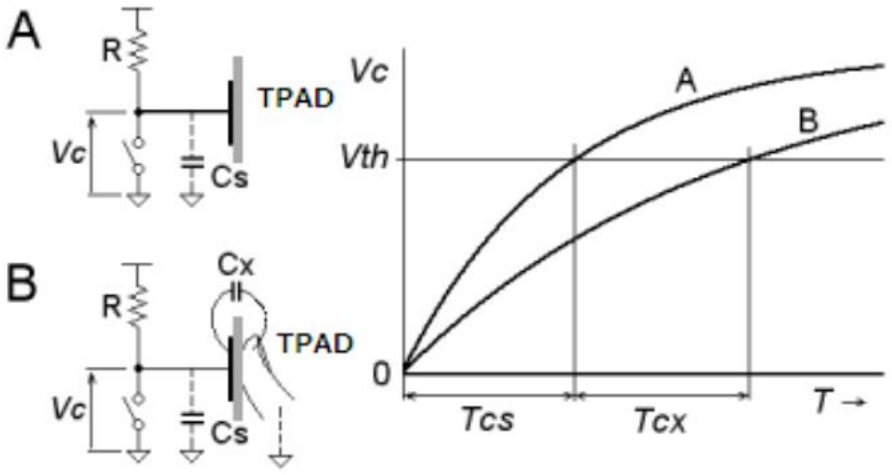
4° 初始化 24CXX

```
AT24CXX_Init();
```

3.13 TPAD

3.13.1 模块工作原理

通过检测电容放电时间的方法来判断是否对 TPAD 有触摸行为。

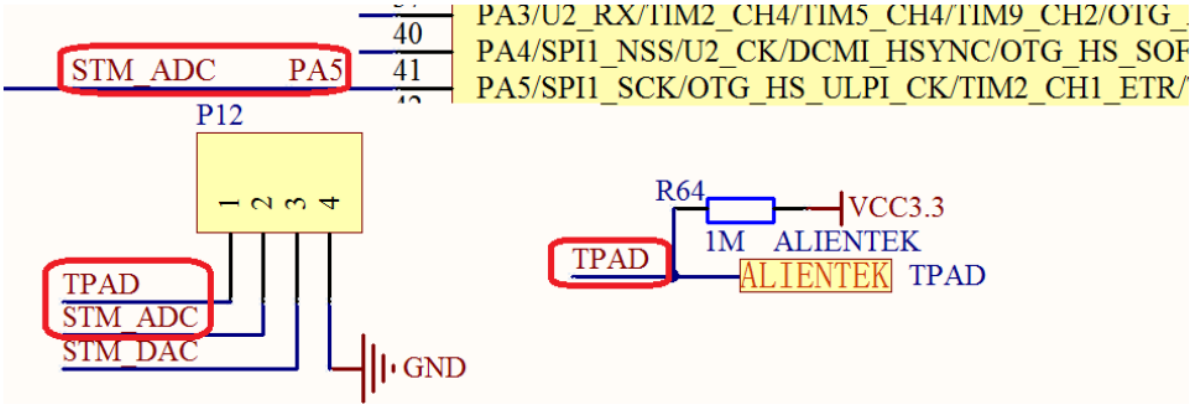


图表 19：TPAD 按键原理

其中，R是外接的电容充电电阻，C_S是没有触摸按下时 TPAD 和 PCB 之间的杂散电容，C_X是有手指按下时手指与 TPAD 之间形成的电容。先用开关将C_S上的电放尽，然后断开开关，让R给C_S充电。当无手指触摸时，C_S的充电曲线如图中的 A 曲线；当有手指触摸时，C_S的充电曲线如图中的 B 曲线。在 A、B 两种情况下，V_C达到V_{th}的时间分别为T_{CS}和$T_{CS} + T_{CX}$。当充电时间在T_{CS}附近，就可以认为没有触摸，而重带你时间大于$T_{CS} + T_{CX}$时，就认为有触摸按下。

实验中，使用定时器 TIM2 的通道 1 进行输入捕获，在 MCU 每次复位重启的时候，执行一次捕获检测，记录此时的值为 tpad_default_val 作为判断依据。在后续的捕获检测，通过与 tpad_default_val 的对比来判断是不是有触摸发生。

3.13.2 硬件电路的连接



图表 20：TPAD 和 STM32F4 连接图

3.13.3 库函数配置方法

1° TIM2 定时器、GPIOA 时钟初始化

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);        //TIM2 时钟使能
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);    //使能 PORTA 时钟
```

2° 引脚复用映射设置

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource5,GPIO_AF_TIM2);//GPIOA 复用为定时器 2
```

3° GPIO 初始化

```
GPIO_Init(GPIOA,&GPIO_InitStructure);
```

4° TIM2 初始化

```
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
TIM_ICInit(TIM2, &TIM2_ICInitStructure);
```

5° TIM2 使能

```
TIM_Cmd(TIM2,ENABLE);
```

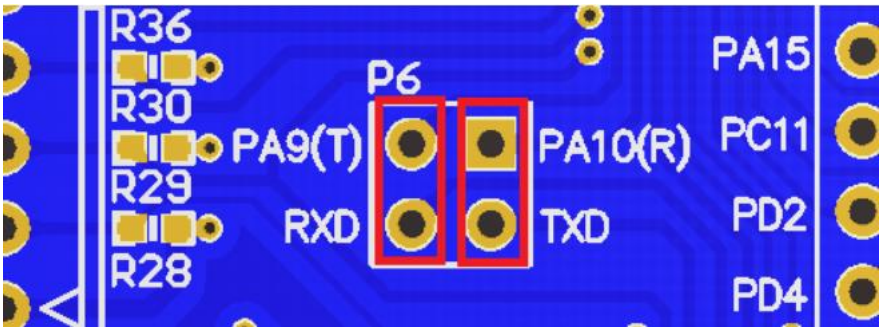
3.14 USART

3.14.1 模块工作原理

串口作为 MCU 的重要外部接口，同时也是软件开发重要的调试手段。本次实验我们利用 USART 进行 printf，输出调试信息帮助编程。

调试时，打开串口调试助手，选择相应的 COM 端口，将波特率设置为 115200。

3.14.2 硬件电路的连接



图表 21：USART 连接图

把 P6 的 RXD 和 TXD 用跳线帽与 PA9 和 PA10 连接起来。

3.14.3 库函数配置方法

1° GPIO、USART 时钟使能

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE); //使能 GPIOA 时钟
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE);//使能 USART1 时钟
```

2°GPIO 引脚复用映射

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource9,GPIO_AF_USART1);
//GPIOA9 复用为 USART1
GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1);
//GPIOA10 复用为 USART1
```

3°GPIO 端口初始化

```
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA9，PA10
```

4°串口参数初始化

```
USART_Init(USART1, &USART_InitStructure);
//设置波特率、字长、资产校验等参数
```

5°使能串口

```
USART_Cmd(USART1, ENABLE);
```

6°串口数据收发

```
void USART_SendData();                  //发数据，写 DR
uint16_t USART_ReceiveData();           //接受数据，读 DR
```

7°串口传输状态获取

```
FlagStatus USART_GetFlagStatus();       //状态标志
void USART_ClearITPendingBit();         //中断标志
```

四、 具体实现过程描述

4.1 calendar ----- 日历和时钟

4.1.1 刷新 LCD 屏上的日历----calendar_date_refresh()

1、函数介绍

RTC 自身定义了一个日历结构体，其中包括了当前的年、月、日、星期信息，所以可以直接调用该结构体中的变量转换格式后显示到 LCD 屏上。

```
RTC_DateTypeDef RTC_DateStruct;         //日期结构体：年、月、日、星期
```

为了界面的美观，在不断的测试后设置了日历显示在屏幕上的位置如下。

```
static u16 DATE_TOPX=50;                //日期 x 坐标位置
static u16 DATE_TOPY=10;                //日期 y 坐标位置
```

系统导入了中文字体，所以在显示“星期几”时，定义了一个中文字符数组，用日历结构体的星期变量作为数组的下标来索引到对应的星期显示到 LCD 屏幕上。

```
u8*const week_table[8]={(u8 *)" ",(u8 *)"一",(u8 *)"二",(u8 *)"三",(u8 *)"四",(u8 *)"五",(u8 *)"六",(u8 *)"日"};                    //星期几数组
```

当调用 calendar_play()时，调用该函数，不断刷新显示当前的日历信息；当当前日期信息不对时，进入设置界面校正日历信息，再返回时钟界面时，日历信息即被刷新成功。

2、函数主体

```
//*********************************************************************
//根据当前的日期,更新显示到屏幕上的日历信息
//返回类型：void
//使用参数：无
//*********************************************************************
void calendar_date_refresh(void)
{
    //显示阳历年月日
    POINT_COLOR=WHITE;//白色画笔
    gui_show_num(DATE_TOPX,DATE_TOPY,2,WHITE,36,((RTC_DateStruct.RTC_Year+2000)/100)%100,0X80);          //显示年  20/19
    gui_show_num(DATE_TOPX+36,DATE_TOPY,2,WHITE,36,(RTC_DateStruct.RTC_Year+2000)%100,0X80);          //显示年
    gui_show_ptchar(DATE_TOPX+72,DATE_TOPY,lcddev.width,lcddev.height,0,WHITE,36,'-',0);//"-"
    gui_show_num(DATE_TOPX+90,DATE_TOPY,2,WHITE,36,RTC_DateStruct.RTC_Month,0X80);
                    //显示月
    gui_show_ptchar(DATE_TOPX+126,DATE_TOPY,lcddev.width,lcddev.height,0,WHITE,36,'-',0);//"-"
    gui_show_num(DATE_TOPX+144,DATE_TOPY,2,WHITE,36,RTC_DateStruct.RTC_Date,0X80);
                    //显示日
    //显示星期几
    POINT_COLOR=RED;
    Show_Str(DATE_TOPX+24,DATE_TOPY+50,lcddev.width,lcddev.height,(u8 *)"星期",24,0);
                    //显示汉字“星期”
    Show_Str(DATE_TOPX+72,DATE_TOPY+50,lcddev.width,lcddev.height,week_table[RTC_DateStruct.RTC_WeekDay],24,0);//显示汉字
}
```

4.1.2 绘制钟表盘----calendar_circle_clock_drawpanel()

1、函数功能介绍

该函数用于绘制钟表盘，参照“综合测试实验”中绘制钟表盘的方法，调用如下函数来实现：

首先调用画实心圆函数绘制钟表盘的外圈、内圈、中心圆圈即得到表盘框架。

```
void gui_fill_circle(u16 x0,u16 y0,u16 r,u16 color);                    //画实心圆
```

然后再调用画粗线函数绘制分钟、小时格，按照数学的 sin()和 cos()方法来获取需要绘制的方格的起点和终点坐标；在绘制小时格的同时将表盘数字绘制在小时格旁边即可。

```
void gui_draw_bline1(u16 x0,u16 y0,u16 x1,u16 y1,u8 size,u16 color);    //画一条粗线,方法 1
```

调用该函数时，需要设置表盘的相关参数如下：

```
static u16 center_x=240;                                    //表盘中心 x 坐标
static u16 center_y=300;                                    //表盘中心 y 坐标
static u16 size=340;                                        //表盘直径大小
static u16 d=12;                                            //表盘外圈宽度
```

该函数在运行 calendar_play()时调用。

2、函数主体

```
//*********************************************************************
//画圆形指针表盘
//返回类型：void
//使用参数：x,y:坐标中心点
//          size:表盘大小(直径)
//          d:表盘分割,秒钟的高度
//*********************************************************************
void calendar_circle_clock_drawpanel(u16 x,u16 y,u16 size,u16 d)
{
    u16 r=size/2;                   //得到半径
    u16 sx=x-r;                     //左边界
    u16 sy=y-r;                     //上边界
    u16 px0,px1;                    //秒钟、时钟格起点的坐标
    u16 py0,py1;                    //秒钟、时钟格终点的坐标
    u16 i;                          //循环变量
    gui_fill_circle(x,y,r,RED);           //画外圈
    gui_fill_circle(x,y,r-6,BLACK);   //画内圈
    for(i=0;i<60;i++)//画秒钟格
    {
        px0=sx+r+(r-7)*sin((PI/30)*i);          //获取起点 x 坐标
        py0=sy+r-(r-7)*cos((PI/30)*i);          //获取起点 y 坐标
        px1=sx+r+(r-d)*sin((PI/30)*i);          //获取终点 x 坐标
        py1=sy+r-(r-d)*cos((PI/30)*i);          //获取终点 y 坐标
        gui_draw_bline1(px0,py0,px1,py1,0,WHITE);          //画一条粗线
    }
    for(i=0;i<12;i++)//画小时格
    {
        px0=sx+r+(r-7)*sin((PI/6)*i);           //获取起点 x 坐标
        py0=sy+r-(r-7)*cos((PI/6)*i);           //获取起点 y 坐标
        px1=sx+r+(r-d-2)*sin((PI/6)*i);         //获取终点 x 坐标
        py1=sy+r-(r-d-2)*cos((PI/6)*i);         //获取终点 y 坐标
```

```
        gui_draw_bline1(px0,py0,px1,py1,2,YELLOW); //画一条粗线
        //显示表盘刻度
        px1=sx+r+(r-2*d-8)*sin((PI/6)*i)-d-2;
        py1=sy+r-(r-2*d-8)*cos((PI/6)*i)-d-2;
        if(i==0) LCD_ShowNum(px1,py1,12,2,24);
        else LCD_ShowNum(px1,py1,i,2,24);
    }
    for(i=0;i<4;i++)//画 3 小时格
    {
        px0=sx+r+(r-7)*sin((PI/2)*i);           //获取起点 x 坐标
        py0=sy+r-(r-7)*cos((PI/2)*i);           //获取起点 y 坐标
        px1=sx+r+(r-d-7)*sin((PI/2)*i);         //获取终点 x 坐标
        py1=sy+r-(r-d-7)*cos((PI/2)*i);         //获取终点 y 坐标
        gui_draw_bline1(px0,py0,px1,py1,2,YELLOW); //画一条粗线
    }
    gui_fill_circle(x,y,d/2,RED);       //画中心圈
}
```

4.1.3 绘制时针、分针、秒针----calendar_circle_clock_showtime()

1、函数功能介绍

该函数用于绘制钟表盘上的时针、分针、秒针，在 calendar_play()函数中调用。每当时钟时间变化时，钟表盘上的指针也应该随之变化。

在该函数中首先清除上一次显示的指针，再根据当前的时间绘制新的指针，根据人的视觉暂留作用，看上去就是指针的走动效果。

2、函数主体

```
//*********************************************************************
//显示钟表盘上的时针、分针、秒针
//返回类型：void
//使用参数：x,y:坐标中心点
//          size:表盘大小(直径)
//          d:表盘分割,秒钟的高度
//          hour:时钟、min:分钟、sec:秒钟
//*********************************************************************
void calendar_circle_clock_showtime(u16 x,u16 y,u16 size,u16 d,u8 hour,u8 min,u8 sec)
{
    static u8 oldhour=0;    //最近一次进入该函数的时分秒信息
    static u8 oldmin=0;
    static u8 oldsec=0;
    float temp;
    u16 r=size/2;       //得到半径
    u16 sx=x-r;         //左边界
```

```
    u16 sy=y-r;          //上边界
    u16 px0,px1;         //起点
    u16 py0,py1;         //终点
    u8 r1=d/2+3;         //指针终点
    if(hour>11)hour-=12;//超过 12 小时时重新循环
  ////////////////清除上一次的数据//////////////////
    //清除小时
    temp=(float)oldmin/60;
    temp+=oldhour;
    px0=sx+r+(r-6*d)*sin((PI/6)*temp);
    py0=sy+r-(r-6*d)*cos((PI/6)*temp);
    px1=sx+r+r1*sin((PI/6)*temp);
    py1=sy+r-r1*cos((PI/6)*temp);
    gui_draw_bline1(px0,py0,px1,py1,2,BLACK);
    //清除分钟
    temp=(float)oldsec/60;
    temp+=oldmin;
    px0=sx+r+(r-5*d)*sin((PI/30)*temp);
    py0=sy+r-(r-5*d)*cos((PI/30)*temp);
    px1=sx+r+r1*sin((PI/30)*temp);
    py1=sy+r-r1*cos((PI/30)*temp);
    gui_draw_bline1(px0,py0,px1,py1,1,BLACK);
    //清除秒钟
    px0=sx+r+(r-4*d)*sin((PI/30)*oldsec);
    py0=sy+r-(r-4*d)*cos((PI/30)*oldsec);
    px1=sx+r+r1*sin((PI/30)*oldsec);
    py1=sy+r-r1*cos((PI/30)*oldsec);
    gui_draw_bline1(px0,py0,px1,py1,0,BLACK);
  ////////////////////显示//////////////////////
    //显示新的时钟
    temp=(float)min/60;
    temp+=hour;
    px0=sx+r+(r-6*d)*sin((PI/6)*temp);
    py0=sy+r-(r-6*d)*cos((PI/6)*temp);
    px1=sx+r+r1*sin((PI/6)*temp);
    py1=sy+r-r1*cos((PI/6)*temp);
    gui_draw_bline1(px0,py0,px1,py1,2,YELLOW);
    //显示新的分钟
    temp=(float)sec/60;
    temp+=min;
    px0=sx+r+(r-5*d)*sin((PI/30)*temp);
    py0=sy+r-(r-5*d)*cos((PI/30)*temp);
    px1=sx+r+r1*sin((PI/30)*temp);
    py1=sy+r-r1*cos((PI/30)*temp);
```

```
    gui_draw_bline1(px0,py0,px1,py1,1,GREEN);
    //显示新的秒钟
    px0=sx+r+(r-4*d)*sin((PI/30)*sec);
    py0=sy+r-(r-4*d)*cos((PI/30)*sec);
    px1=sx+r+r1*sin((PI/30)*sec);
    py1=sy+r-r1*cos((PI/30)*sec);
    gui_draw_bline1(px0,py0,px1,py1,0,GBLUE);
    oldhour=hour;       //保存时
    oldmin=min;         //保存分
    oldsec=sec;         //保存秒
}
```

4.1.4 显示温度----void temperature_show()

1、函数功能介绍

该函数用于获取 ADC 内部温度通道的温度，在计算过后显示在 LCD 屏幕上。

为了界面的美观，在不断的测试后设置了温度显示在屏幕上的位置如下。

```
    static u16 TEMP_TOPX=70;            //温度 x 坐标位置
    static u16 TEMP_TOPY=600;           //温度 y 坐标位置
```

2、函数主体

```
//*********************************************************************
//显示温度到屏幕指定位置
//返回类型：void
//使用参数：无
//*********************************************************************
void temperature_show(void)
{
    POINT_COLOR=WHITE;                  //白色画笔
    Show_Str(TEMP_TOPX,TEMP_TOPY+20,lcddev.width,lcddev.height,(u8 *)"温度",24,0);
                                        //显示汉字“温度”
    gui_show_ptchar(TEMP_TOPX+48,TEMP_TOPY+20,lcddev.width,lcddev.height,0,WHITE,24,':',0);
                                        //":"
    temp=Get_Temprate();                //得到温度值
        if(temp<0)                      //当温度为负时
        {
        temp=-temp;                     //取反
        gui_show_ptchar(TEMP_TOPX+72,TEMP_TOPY,lcddev.width,lcddev.height,0,GBLUE,60,'-
',0);                                   //显示负号"-"
        }
        else
    gui_show_ptchar(TEMP_TOPX+72,TEMP_TOPY,lcddev.width,lcddev.height,0,BRRED,60,' ',0);
                                        //无符号
    gui_show_num(TEMP_TOPX+92,TEMP_TOPY,2,BRRED,60,temp/100,0X80);          //整数部分
```

```
    gui_show_ptchar(TEMP_TOPX+152,TEMP_TOPY,lcddev.width,lcddev.height,0,BRRED,60,'.',0);
                                                    //"."
    gui_show_num(TEMP_TOPX+182,TEMP_TOPY,2,BRRED,60,temp%100,0X80);       //小数部分
    gui_show_ptchar(TEMP_TOPX+252,TEMP_TOPY,lcddev.width,lcddev.height,0,BRRED,60,95+'
',0);                                               //单位“摄氏度”
}
```

4.1.5 整点报时响铃----calendar_ring()

1、函数功能介绍

该函数用于蜂鸣器响，在设置整点报时、半小时报时时，调用该函数。

根据输入参数 TYPE 的不同，蜂鸣器响的方式也不同。

2、函数主体

```
//**********************************************************************
//整点报时
//返回类型：void
//使用参数：type:闹铃类型
//0,滴.
//1,滴.滴.
//2,滴.滴.滴
//3,滴.滴.滴.滴
//**********************************************************************
void calendar_ring(u8 type)
{
    u8 i;                                           //临时变量
    for(i=0;i<(type+1);i++)                         //循环
    {
        BEEP=1;                                     //蜂鸣器响
        delay_ms(50);
        BEEP=0;                                     //蜂鸣器停
        delay_ms(70);
    }
}
```

4.1.6 运行时钟----calendar_play()

1、函数功能介绍

该函数为时钟运行的函数，是整个时钟的主体运行函数。

我负责的部分主要是时钟的运行和其中闹钟的后台调度，中间跳转到可视化的 GUI 设置界面则是由我们小组的另一个成员负责的，这里只描述我负责的部分。

为了界面的美观，在不断的测试后设置了数码管字体的时间显示在屏幕上的位置。

```
static u16 TIME_TOPX=100;                //时间 x 坐标位置
static u16 TIME_TOPY=500;                    //时间 y 坐标位置
```

时钟获取时间使用的是 RTC 定义的时间结构体。

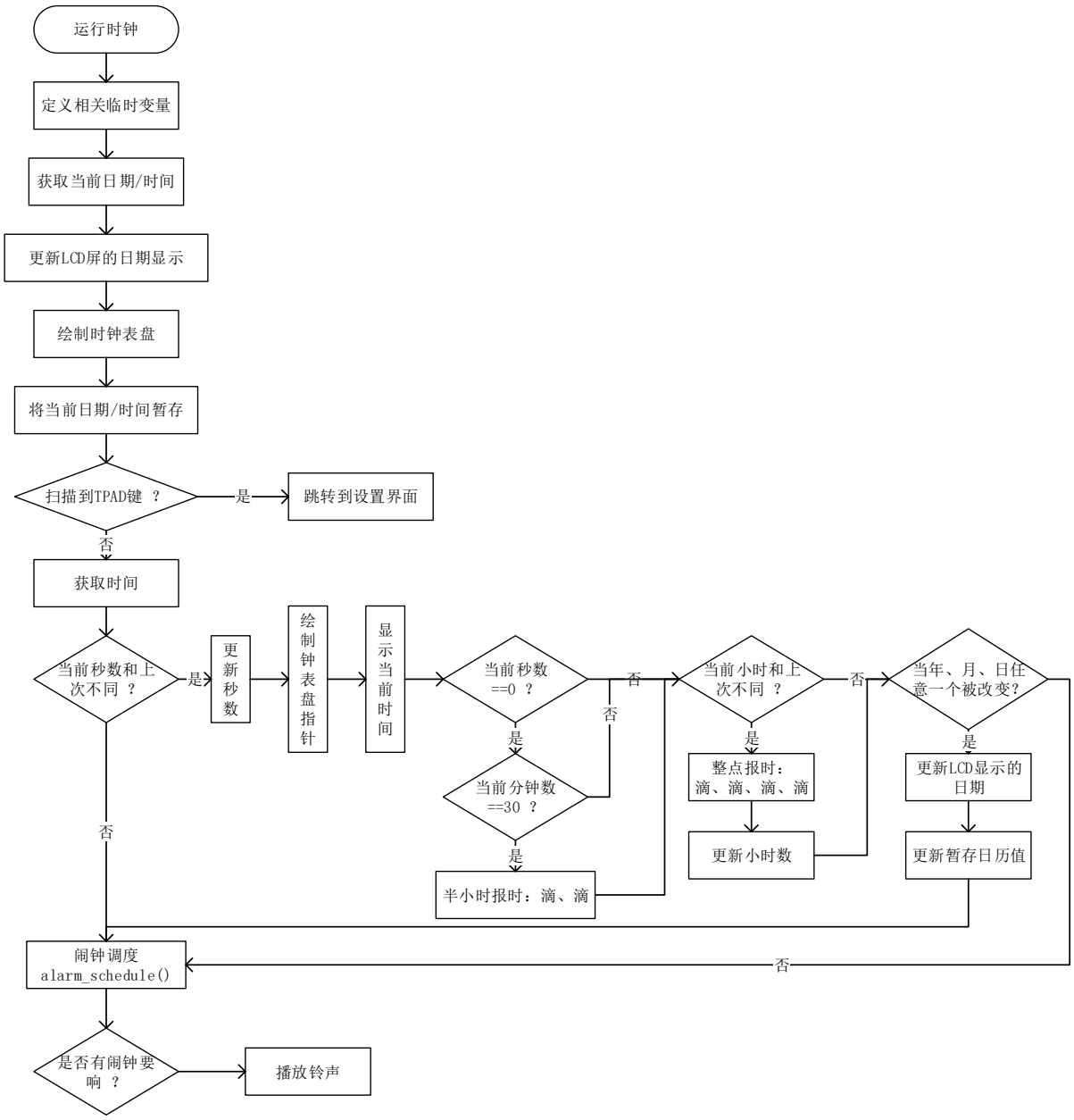
```
RTC_TimeTypeDef RTC_TimeStruct;       //时间结构体：时、分、秒、ampm
```

2、函数主体

```
//*********************************************************************
//运行时钟：不断更新时间(包括 闹钟调度/校时校分/闹钟设置 的调用)
//返回类型：void
//使用参数：无
//*********************************************************************
u8 calendar_play(u8 tag)
{
    u8 rval=HOLD_CLOCK_SETTING;         //返回值
    u8 second=0;                        //临时秒数
    u8 minute=0;                        //临时分钟数
    u8 hour=0;                          //临时小时数
    u8 tempdate=0;                      //临时天数值
    u8 tempmonth=0;                     //临时月份
    u8 tempyear=0;                      //临时年份
    RTC_GetTime(RTC_Format_BIN,&RTC_TimeStruct);          //获取时间
    RTC_GetDate(RTC_Format_BIN,&RTC_DateStruct);          //获取日期
    calendar_date_refresh();                              //更新日期
    POINT_COLOR=WHITE;                                    //白色画笔
    calendar_circle_clock_drawpanel(center_x,center_y,size,d);      //显示指针时钟表盘
    gui_show_ptchar(TIME_TOPX+80,TIME_TOPY,lcddev.width,lcddev.height,0,GBLUE,60,':',0);//":"
    gui_show_ptchar(TIME_TOPX+170,TIME_TOPY,lcddev.width,lcddev.height,0,GBLUE,60,':',0);//":"
    second=RTC_TimeStruct.RTC_Seconds;              //暂存当前秒数
    minute=30;                                      //用于设置 30min 时报时
    hour=RTC_TimeStruct.RTC_Hours;                  //暂存小时数
    tempdate=RTC_DateStruct.RTC_Date;               //暂存当前日期
    tempmonth=RTC_DateStruct.RTC_Month;             //暂存当前月份
    tempyear=RTC_DateStruct.RTC_Year;               //暂存当前年份
    while(1)
    {
        //判断 TPAD 返回按键
        if(TPAD_Scan(0)){
            rval=SWITCH_CLOCK_SETTING;
            break;
        }
        ////////////////////时钟设置//////////////////////////////
        RTC_GetTime(RTC_Format_BIN,&RTC_TimeStruct);//获取时间
        if(second!=RTC_TimeStruct.RTC_Seconds||tag==2)//上一次秒数和当前不同(在这里判断 tag
==2 达到设置界面时间也显示的效果)
```

```
        {
            second=RTC_TimeStruct.RTC_Seconds;//更新秒数
    calendar_circle_clock_showtime(center_x,center_y,size,d,RTC_TimeStruct.RTC_Hours,RTC_TimeStruct.RTC_Minutes,RTC_TimeStruct.RTC_Seconds);//显示时分秒指针
    gui_show_num(TIME_TOPX+20,TIME_TOPY,2,GBLUE,60,RTC_TimeStruct.RTC_Hours,0X80);
    //显示时
    gui_show_num(TIME_TOPX+110,TIME_TOPY,2,GBLUE,60,RTC_TimeStruct.RTC_Minutes,0X80);
    //显示分
    gui_show_num(TIME_TOPX+200,TIME_TOPY,2,GBLUE,60,RTC_TimeStruct.RTC_Seconds,0X80);
    //显示秒
    temperature_show();
    RTC_GetTime(RTC_Format_BIN,&RTC_TimeStruct);      //更新时间结构体
    RTC_GetDate(RTC_Format_BIN,&RTC_DateStruct);      //更新日期结构体
    //半小时报时
    if(RTC_TimeStruct.RTC_Seconds == 0)
        if(minute == RTC_TimeStruct.RTC_Minutes)          //是 30 分
        {
            calendar_ring(1);
        }
        //整点报时
        if(hour!=RTC_TimeStruct.RTC_Hours)                //上一次小时和现在不同
        {
            calendar_ring(3);
            hour=RTC_TimeStruct.RTC_Hours;                //防止重复进入
        }
        //当年、月、日任意一个被改变都立即更新屏幕上的日期
    if((RTC_DateStruct.RTC_Date!=tempdate)|(tempmonth!=RTC_DateStruct.RTC_Month)|(tempyear!=RTC_DateStruct.RTC_Year))
    {
        calendar_date_refresh();                  //更新日期
        tempdate=RTC_DateStruct.RTC_Date; //防止重复进入
        tempmonth=RTC_DateStruct.RTC_Date;
        tempyear=RTC_DateStruct.RTC_Date;
    }
}
    if(tag==2)
        break;
    ////////////////闹钟的相关设置//////////////////////////
    alarm_schedule();             //闹钟调度
    if(Alarm_FLAG == 1)           //当有闹钟要响时
        break;
 }
return rval;
}
```

3、函数配置流程



图表 22：运行时钟流程图

4.2 alarm ----- 闹钟

4.2.1 闹钟初始化----alarm_init()

1、函数功能介绍

该函数用于初始化闹钟信息。包括系统闹钟信息存储的地址、闹钟等待数组、系统已存在的闹钟信息等等。

虽然 RTC 已定义过一个闹钟结构体，但是为了实现多个闹钟，我们需要更多的参

数才行，所以定义了如下的闹钟结构体。

```
__packed typedef struct
{
    u8   hour;                  //闹铃小时
    u8   min;                   //闹铃分钟
    u8   state;                 //闹铃状态,0X80，开状态;其他，为关闭状态
    u8   weekmask;              //闹钟响铃掩码 bit1~bit7,代表周一~周日.
    u8   saveflag;              //保存标志,0X0A,保存过了;其他,还从未保存
}_alarm_obj;
```

另外，要想设置多个闹钟，还闹钟的调度中还需要以下这些变量存储相关的设置信息。

```
_alarm_obj Alarm[RTC_Alarm_MAX_NUM];        //多个闹钟结构体
u8 SYSTEM_Alarm_NUM;                        //系统中的闹钟个数
u8 wait_alarm[RTC_Alarm_MAX_NUM];           //等待闹钟的顺序（以在数组中的序号为索引）
```

为了使实验板关闭电源后，在下一次打开时仍可以读取到之前设置的闹钟信息，所以使用了系统的可擦除存储器 EEPROM 来存储闹钟信息，对闹钟的各种增删改查操作都同步到系统中，所以定义了闹钟在系统中存储的地址如下：

```
u8 SYSTEM_Alarm_SAVE_BASE;                  //闹钟存储起始地址
u8 SYSTEM_Alarm_SAVE_END;                   //闹钟存储结束地址
```

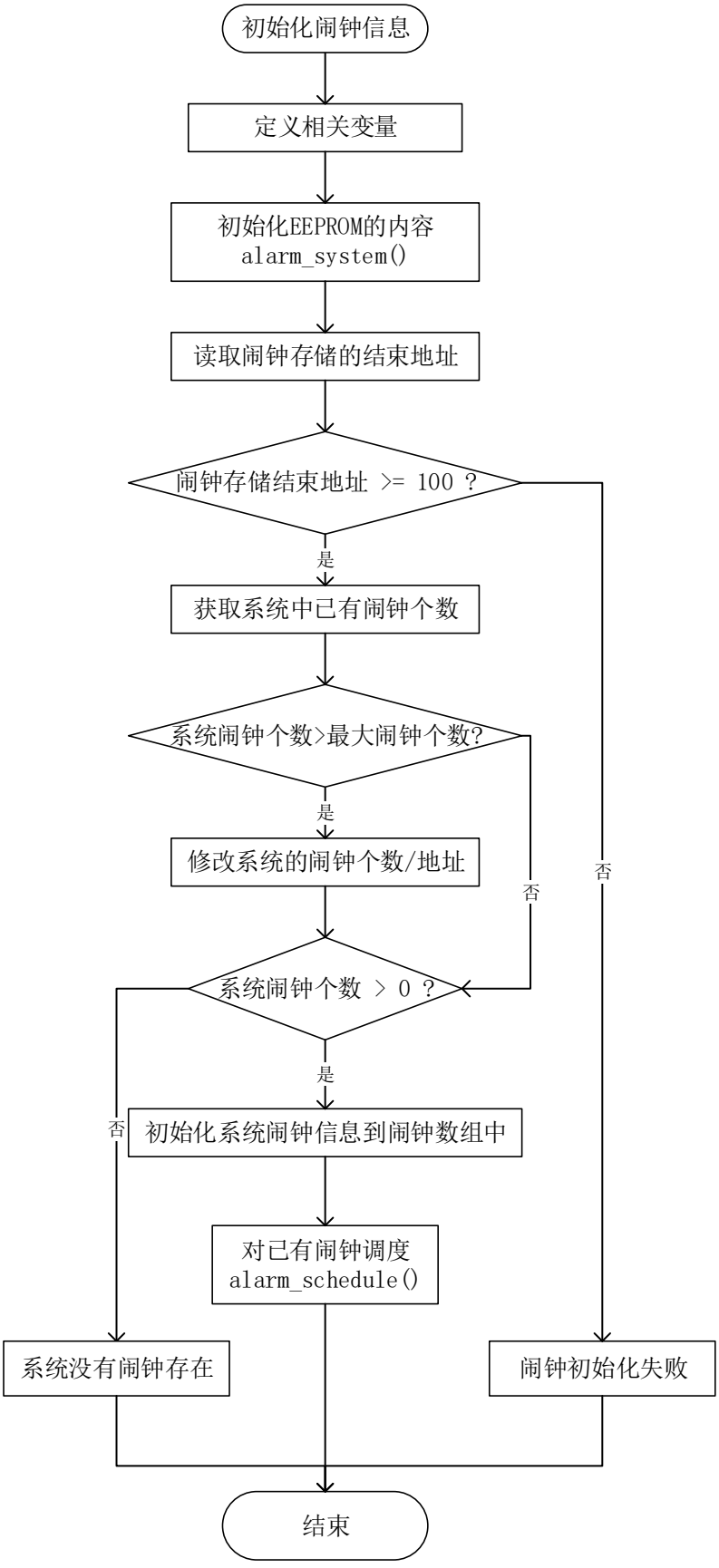
初始化闹钟成功之后，说明闹钟模块就可以使用了，接下来就可以使用触摸屏的闹钟设置界面设置闹钟的相关信息，传递参数到闹钟系统中，实现闹钟的运行。

2、函数主体

```
//**********************************************************************
//初始化闹钟：读取系统中的闹钟信息
//返回类型：void
//使用参数：无
//**********************************************************************
void alarm_init(void)
{
    u8 i;                   //循环变量
    u16 addr;       //某个闹钟在系统中的起始地址
    //初始化系统 AT24CXX 内容
    alarm_system();
    //初始化读取闹钟的地址/闹钟个数
    SYSTEM_Alarm_SAVE_BASE = SYSTEM_PARA_SAVE_BASE
+sizeof(SYSTEM_Alarm_SAVE_END);                 //闹钟存储起始地址
    SYSTEM_Alarm_SAVE_END = AT24CXX_ReadOneByte(SYSTEM_PARA_SAVE_BASE);//获取系统存储结束地址
```

```
    //初始化闹钟等待数组的序号
    for(i=0;i<RTC_Alarm_MAX_NUM;i++)
        wait_alarm[i] = i;
    //当闹钟存储的结束地址符合要求时
    printf("\r\nAlarm_endaddr: %d\r\n",SYSTEM_Alarm_SAVE_END);
    if(SYSTEM_Alarm_SAVE_END >= 0X64)
    {
        SYSTEM_Alarm_NUM = (SYSTEM_Alarm_SAVE_END - SYSTEM_Alarm_SAVE_BASE) /
sizeof(_alarm_obj);      //获取系统中已有的闹钟个数
        //当系统闹钟个数比（最大闹钟个数）大时
        if(SYSTEM_Alarm_NUM>RTC_Alarm_MAX_NUM)
        {
            SYSTEM_Alarm_NUM = RTC_Alarm_MAX_NUM;       //修改系统闹钟个数为最大闹
钟个数
            SYSTEM_Alarm_SAVE_END = RTC_Alarm_MAX_NUM * sizeof(_alarm_obj);  //修改
闹钟存储结束地址
        AT24CXX_WriteOneByte(SYSTEM_PARA_SAVE_BASE,SYSTEM_Alarm_SAVE_END);//存
入 AT24CXX
        }
        //随机初始化几个闹钟用于测试
        printf("\r\nAlarm_num: %d\r\n",SYSTEM_Alarm_NUM);//测试输出闹钟个数
        //当系统中有闹钟时
        if (SYSTEM_Alarm_NUM > 0)
        {
            //读取闹钟信息到数组中
            for(i=0;i<SYSTEM_Alarm_NUM;i++)
            {
                addr=SYSTEM_Alarm_SAVE_BASE + i * sizeof(_alarm_obj); //即将读取的闹钟的
起始地址
                alarm_read(addr,i);        //读取闹钟
                //测试：输出系统中现有的闹钟
    printf("\r\nAlarm_AT24CXX[%d]:%d,%d,%d,%d,%d,\r\n",i,Alarm[i].hour,Alarm[i].min,Alarm[i].stat
e,Alarm[i].weekmask,Alarm[i].saveflag);//测试从系统中读出来的闹钟信息
            }
            //闹钟调度
            alarm_schedule();
        }
        else
            printf("\r\n----------------------目前系统中并没有任何闹钟信息！请在闹钟设置界面设
置！----------------------\r\n");
    }
    else
        printf("\r\nAlarm 初始化失败\r\n");
}
```

3、函数配置流程



图表 23：闹钟初始化流程图

4.2.2 闹钟调度----alarm_schedule()

1、函数功能介绍

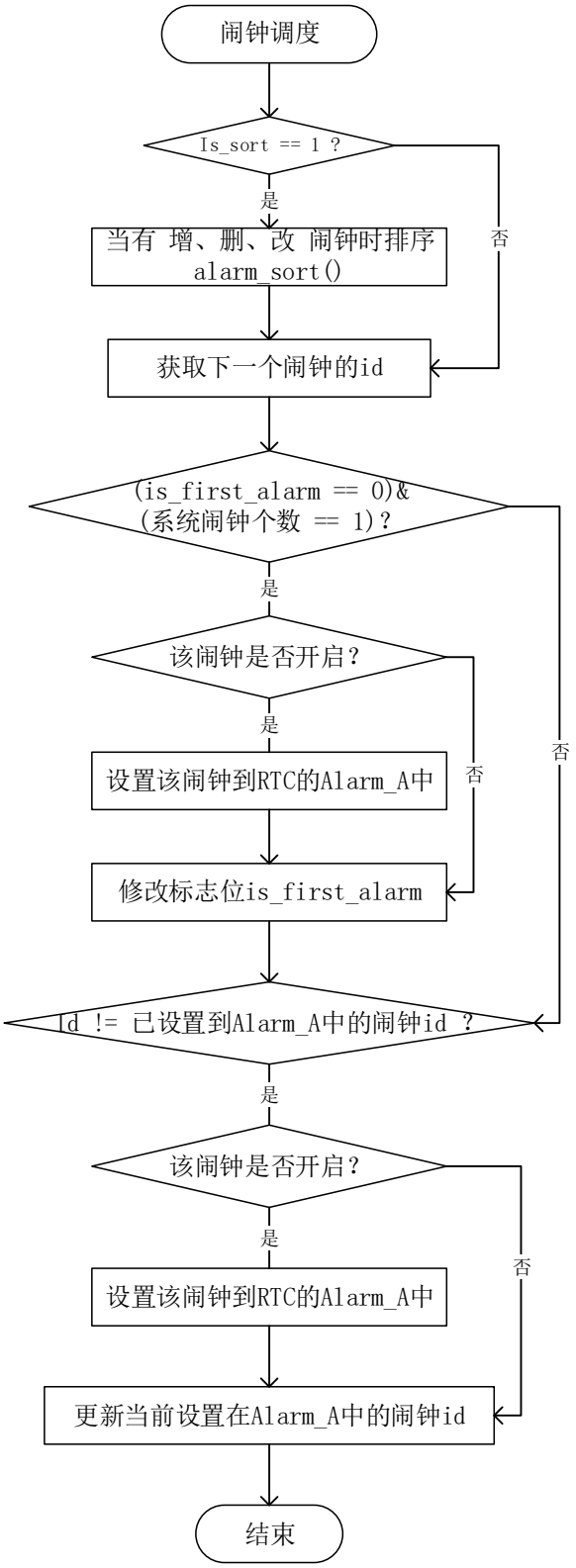
该函数使对系统中已存在的闹钟进行调度。由于 RTC 中只有两个可编程闹钟 Alarm_A 和 Alarm_B，而我们需要实现多闹钟，所以将所有的闹钟进行排序然后不断地将离当前时间最近的闹钟设置到 Alarm_A 中，当该闹钟执行过后，立即将下一个闹钟设置进去，这样就可以实现多个闹钟的功能。所以在系统的时钟运行时，或者当对闹钟有了增、删、改操作时，要一直不断的运行、调度闹钟。

2、函数主体

```
//*********************************************************************
//闹钟调度：将排序后的闹钟设置到 RTC 的 Alarm_A 中(前提是已经有闹钟信息)
//返回类型：void
//使用参数：无
//*********************************************************************
void alarm_schedule(void)
{
    u8 id;                          //id 号
    //闹钟排序:当有 增、删、改 闹钟时排序
    if(is_sort == 1)
    {   alarm_sort();       //闹钟排序
        is_sort =0;         //修改标志位
    }
    id = alarm_compare();  //判断排序过的闹钟中下一个需要响的闹钟的 id 号
    //当是第一个闹钟时
    if((SYSTEM_Alarm_NUM==1)&(is_first_alarm==0))
    {   if(Alarm[id].state == 0X80)             //开启状态
        {
            printf("\r\n 进入 Alarm_A 的设置，要设置的闹钟是：[%d],时间
为：%d:%d\r\n",id,Alarm[id].hour,Alarm[id].min);
            RTC_Set_AlarmA(Alarm[id].weekmask,Alarm[id].hour,Alarm[id].min,0);
        }
        is_first_alarm = 1;//已执行过一次，防止重复进入
    }
    //当需要设置的闹钟与已经在 Alarm_A 中的闹钟重复时，不需要重新设置
    if(id != ring_id)
    {   //当闹钟是开启状态时，将其设置到 Alarm_A 中，等待响铃
        if(Alarm[id].state == 0X80)
        {
            printf("\r\n 进入 Alarm_A 的设置，要设置的闹钟是：[%d],时间
为：%d:%d\r\n",id,Alarm[id].hour,Alarm[id].min);
            RTC_Set_AlarmA(Alarm[id].weekmask,Alarm[id].hour,Alarm[id].min,0);
```

```
        }
        ring_id = id;   //更新当前的闹钟标号
    }
}
```

3、函数配置流程



图表 24：闹钟调度流程图

4.2.3 闹钟排序----alarm_sort()

1、函数功能介绍

该函数对系统中已有的闹钟进行排序。

因为对于定义的闹钟结构体数组，数组下标就代表了每个闹钟的 id 号，对闹钟排序时不能直接对闹钟信息交换，这样在闹钟列表看到的闹钟的顺序就被修改了，所以我们引入一个闹钟等待数组 wait_alarm 来存储闹钟的 id 号，对闹钟排序后，将排序好的闹钟 id 号存在闹钟等待数组中。

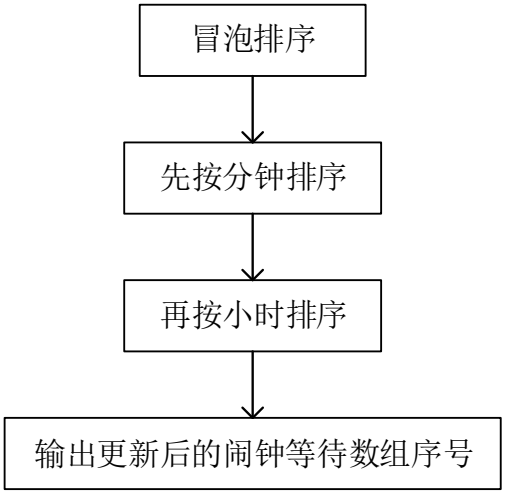
对闹钟的排序选用的是冒泡排序，闹钟排序有两个关键字：hour 和 min 都需要排序，所以应该先对 min 排序，再对 hour 排序，将所有闹钟从小到大排序。

2、函数主体

```
//*********************************************************************
//闹钟排序：将现有的闹钟按时间排序从早到晚
//返回类型：void
//使用参数：无
//*********************************************************************
void alarm_sort(void)
{
    u8 i,j,temp;                                    //临时循环变量
    //先排 min
    for(i=0;i<SYSTEM_Alarm_NUM-1;i++)
    {
        for(j=0;j<SYSTEM_Alarm_NUM-1-i;j++)
        {
            if(Alarm[wait_alarm[j]].min > Alarm[wait_alarm[j+1]].min)
            {
                temp = wait_alarm[j];
                wait_alarm[j] = wait_alarm[j+1];
                wait_alarm[j+1] = temp;
            }
        }
    }
    //再排 hour
    for(i=0;i<SYSTEM_Alarm_NUM-1;i++)
    {
        for(j=0;j<SYSTEM_Alarm_NUM-1-i;j++)
        {
            if(Alarm[wait_alarm[j]].hour > Alarm[wait_alarm[j+1]].hour)
            {
                temp = wait_alarm[j];
```

```
                    wait_alarm[j] = wait_alarm[j+1];
                    wait_alarm[j+1] = temp;
                }
            }
        }
    //测试：输出排序的结果
    printf("\r\n 系统中已存在的闹钟的顺序 wait_alarm[%d]:\r",SYSTEM_Alarm_NUM);
    for(i=0;i<SYSTEM_Alarm_NUM;i++)
        printf("\r%d,\r",wait_alarm[i]+1);//输出闹钟排序顺序
    printf("\r\n");
}
```

3、函数配置流程



图表 25：闹钟排序流程图

4.2.4 闹钟比较得到下一个闹钟 id 号----alarm_compare()

1、函数功能介绍

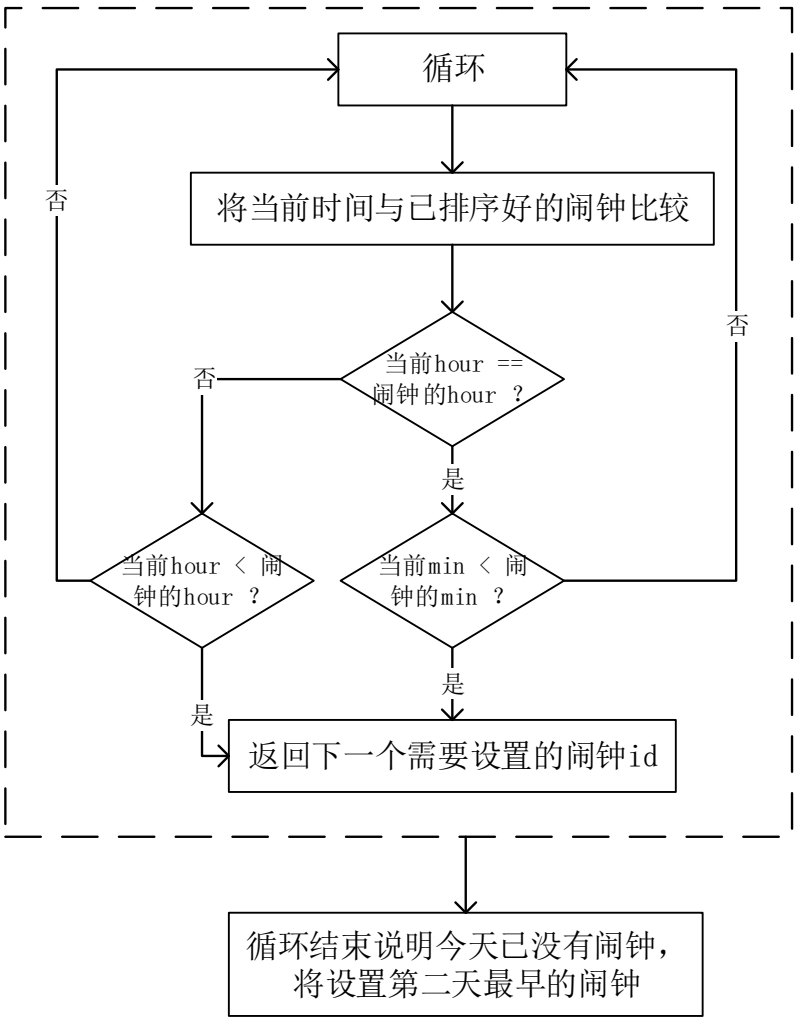
该函数用于返回下一个将要设置的闹钟的 id 号。当对所有已存在的闹钟排序之后，要确定离当前时间最近的闹钟是哪一个？类似于将当前时间插入到排好序的闹钟中，循环比较当前时间和闹钟时间，得到需要设置的闹钟的 id 号。

2、函数主体

```
//*********************************************************************
//闹钟比较：当前 Time 与已排好序的闹钟信息比较，得到下一个要放入的闹钟
//返回类型：u8:返回下一个该设置的闹钟的 id 号
//使用参数：无
//*********************************************************************
u8    alarm_compare(void)
{
    u8 i;           //临时变量
```

```
    //循环已排序好的闹钟与当前时间比较，
    for(i=0;i<SYSTEM_Alarm_NUM;i++)
    {
        if(RTC_TimeStruct.RTC_Hours == Alarm[wait_alarm[i]].hour)//同小时
        {
            if(RTC_TimeStruct.RTC_Minutes < Alarm[wait_alarm[i]].min)//比较分钟
                return wait_alarm[i];
        }
        else if(RTC_TimeStruct.RTC_Hours < Alarm[wait_alarm[i]].hour)
            return wait_alarm[i];
    }
    //循环结束，没有，说明今天已没有闹钟需要设置
    //当下一个闹钟应该为第二天的时,设置为第一个闹钟
    return wait_alarm[0];
}
```

3、函数配置流程



图表 26：闹钟比较得到下一个闹钟 id 号流程图

4.2.5 设置闹钟信息----alarm_Set()

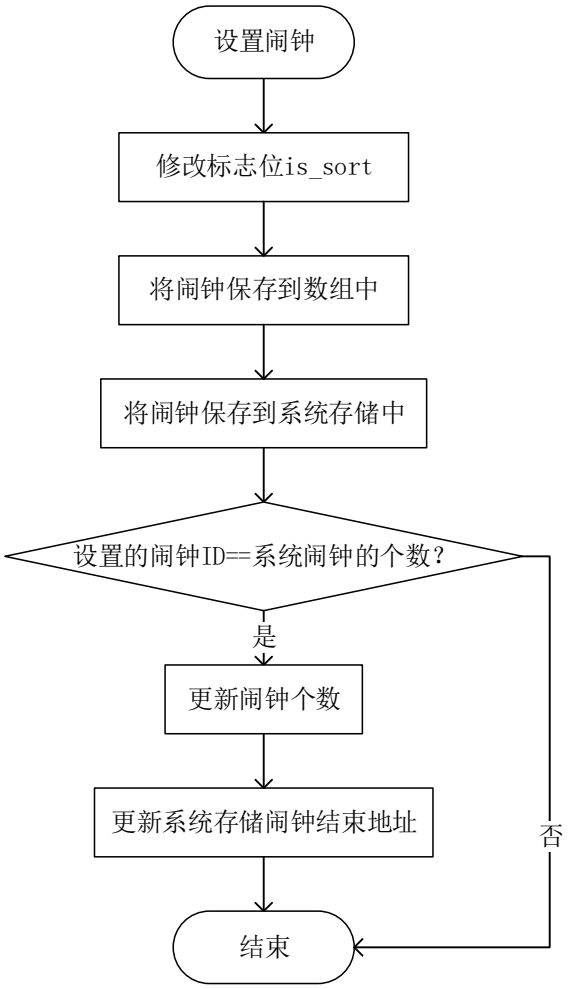
1、函数功能介绍

该函数用于设置闹钟的信息，包括添加闹钟、修改已存在的闹钟。根据传递给函数的 id 号判断该闹钟是要增加还是修改，再对闹钟数组和系统存储的闹钟信息处理。

2、函数主体

```
//*********************************************************************
//闹钟初始化：设置闹钟信息(包括 add/modify)
//返回类型：void
//使用参数：id：闹钟数组的标号
//                hour,min,state,weekmask,saveflag:闹钟结构体的信息
//*********************************************************************
void alarm_Set(u8 id,u8 hour,u8 min,u8 state,u8 weekmask,u8 saveflag)
{
    is_sort = 1;          //需要对闹钟重新排序
    //保存到数组中
    Alarm[id].hour = hour;
    Alarm[id].min = min;
    Alarm[id].state = state;
    Alarm[id].weekmask = weekmask;
    Alarm[id].saveflag = saveflag;
    printf("\r\nAlarm[%d]:%d,%d,%d,%d,%d,\r\n",id,Alarm[id].hour,Alarm[id].min,Alarm[id].state,Alar
m[id].weekmask,Alarm[id].saveflag);//输出设置的闹钟信息
    //保存到 AT24CXX 中
    alarm_save(SYSTEM_Alarm_SAVE_BASE + id * sizeof(_alarm_obj),id);
    //当添加新的闹钟时：修改相关信息
    if(id == SYSTEM_Alarm_NUM)
    {
        SYSTEM_Alarm_NUM+=1;            //增加已设置闹钟个数
        SYSTEM_Alarm_SAVE_END += sizeof(_alarm_obj);          //更新系统闹钟结束地址
        AT24CXX_WriteOneByte( SYSTEM_PARA_SAVE_BASE,SYSTEM_Alarm_SAVE_END);//
存入 AT24CXX
    }
}
```

3、函数配置流程



图表 27：设置闹钟信息流程图

4.2.6 闹钟删除----alarm_delete()

1、函数功能介绍

该函数用于删除闹钟。当在闹钟界面删除某一个闹钟时，调用该函数，对剩余闹钟信息处理，同时更新系统闹钟信息、闹钟个数、闹钟结束地址等等。

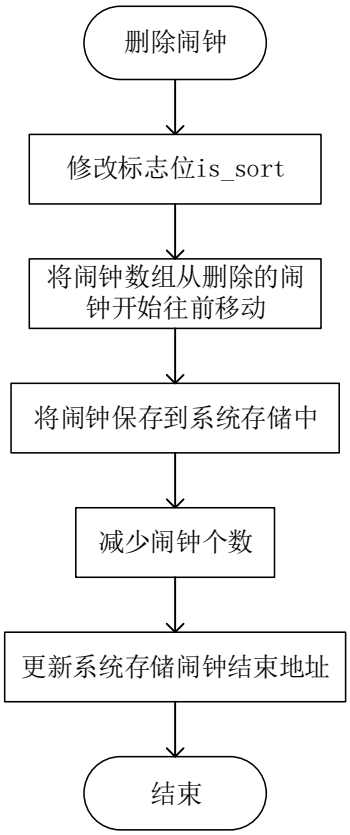
同时在串口调试助手输出一些调试信息，用于验证函数的功能是否正确。

2、函数主体

```
//*****************************************************************
//闹钟删除：更新 ALarm[],AT24CXX,闹钟个数等信息
//返回类型：void
//使用参数：id：闹钟数组的标号
//*****************************************************************
void alarm_delete(u8 id)
{
    u8 i;                   //临时变量
    is_sort = 1;            //需要对闹钟重新排序
```

```
    //修改数组
    for(i = id;i < SYSTEM_Alarm_NUM-1;i++)
    {
        Alarm[i] = Alarm[i+1];
    }
    //修改系统存储
    for(i = id;i< SYSTEM_Alarm_NUM-1;i++)
        alarm_save(SYSTEM_Alarm_SAVE_BASE + i * sizeof(_alarm_obj),i);   //修改系统中闹钟的
信息
    //修改闹钟存储结束地址/闹钟个数
    SYSTEM_Alarm_NUM -= 1;                    //减少已设置闹钟个数
    SYSTEM_Alarm_SAVE_END -= sizeof(_alarm_obj);      //更新系统闹钟结束地址
    AT24CXX_WriteOneByte( SYSTEM_PARA_SAVE_BASE,SYSTEM_Alarm_SAVE_END);//存入
AT24CXX
    //输出调试信息
    printf("\r\n 删除闹钟：Alarm[%d]---------\r\n",id);
    printf("\r\n-----------------此时系统中还剩余  %d  个闹钟如下：----------------
\r\n",SYSTEM_Alarm_NUM);
    for(i =0 ;i <SYSTEM_Alarm_NUM;i++)
    printf("\r\nAlarm[%d]:%d,%d,%d,%d,%d,\r\n",i,Alarm[i].hour,Alarm[i].min,Alarm[i].state,Alarm[i].
weekmask,Alarm[i].saveflag);//输出设置的闹钟信息;
}
```

3、函数配置流程



图表 28：闹钟删除流程图

4.2.7 初始化系统中的闹钟信息----alarm_system()

1、函数功能介绍

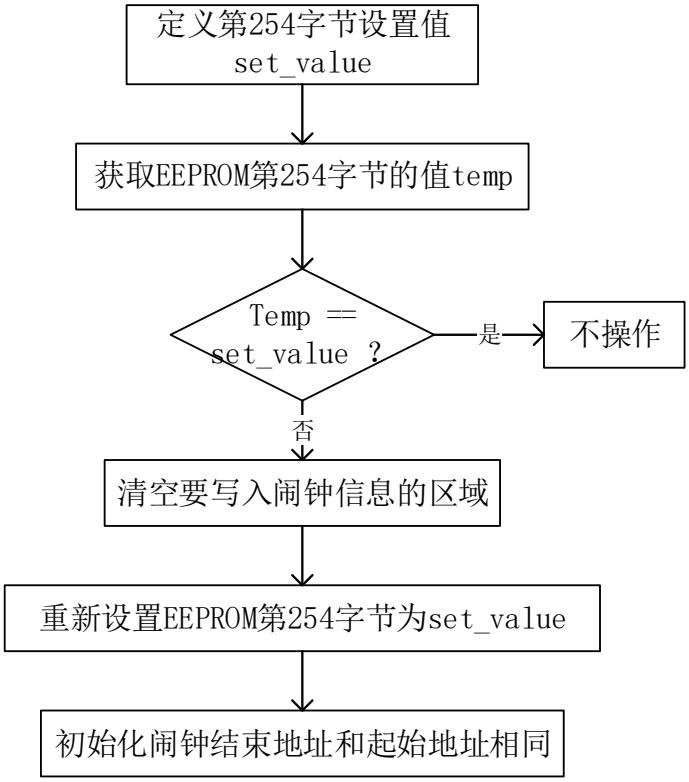
该函数用于清空系统 EEPROM 上的闹钟信息，对其初始化。

当我们将程序下载到一个新的板子上时，我们不知道这个板子上是否之前有过别的程序使用过 EEPROM，所以我们需要对其初始化我们所需要的闹钟存储区域。接下来我们才可以正常使用闹钟模块。初始化一次之后之后每次使用就不会影响了。

2、函数主体

```
//*****************************************************************
//初始化系统中的闹钟信息(设置 AT24CXX 的第 254 个字节为初始化标志)
//返回类型：void
//使用参数：无
//*****************************************************************
void alarm_system(void)
{
    u8 temp;              //临时变量
    u8 i;                 //循环变量
    u8 set_value=0X55;          //第 254 字节设置值
    temp=AT24CXX_ReadOneByte(254);//避免每次开机都写闹钟的地址
    if(temp==set_value) ;
    else//排除第一次初始化的情况
    {
        for(i=99;i<250;i++)
            AT24CXX_WriteOneByte(i,0X00);          //清空
        AT24CXX_WriteOneByte(254,set_value);  //初始化标志
        AT24CXX_WriteOneByte(99,0X64);  //初始填入的结束地址与起始地址相同 100
      temp=AT24CXX_ReadOneByte(254);
    }
}
```

3、函数配置流程



图表 29：系统存储闹钟信息初始化流程图

4.2.8 闹钟比较得到是否已有相同的闹钟存在----alarm_equal()

1、函数功能介绍

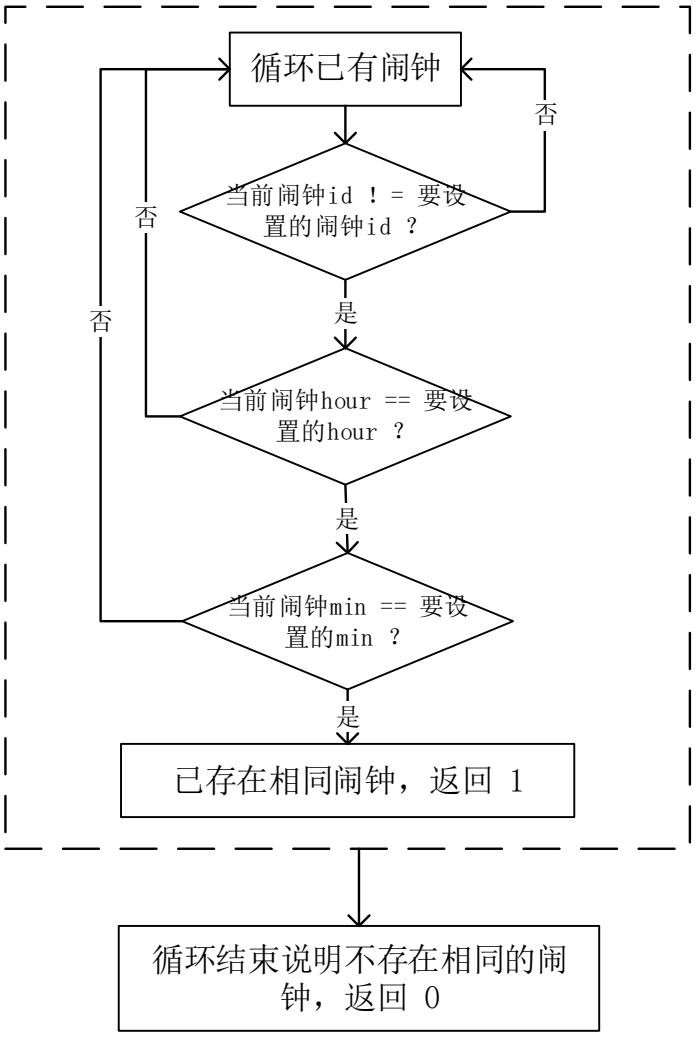
该函数用于比较当前界面设置的闹钟是否是系统中已经存在相同时间的闹钟。因为从用户使用的角度分析，系统中闹钟的时间是不一样的，若有相同时间的闹钟存在，则无法同时设置进入到 RTC 的可编程闹钟中去，所以需要这样的一个函数用来判断系统中是否已经存在相同时间的闹钟，如果有，则不能继续设置该闹钟。

2、函数主体

```
//*****************************************************************
//闹钟比较：闹钟比较得到已存在的闹钟中是否已有相同的闹钟存在
//返回类型：u8:返回值为 1 说明已存在相同的闹钟则不重复设置该闹钟
//使用参数：hour:小时、min:分钟、id：当前设置的闹钟时间
//*****************************************************************
u8     alarm_equal(u8 id,u8 hour,u8 min)
{
     u8 i;              //临时变量
     for(i=0;i<SYSTEM_Alarm_NUM;i++)
     {
       if(i != id)                          //除掉自身 id 号外
       {
```

```
            if(Alarm[i].hour == hour)          //同小时
            {
                if(Alarm[i].min == min)        //同分钟
                    return 1;                                    //存在相同闹钟
            }
        }
    }
    //循环已有的闹钟不存在相同的闹钟
    return 0;
}
```

3、函数配置流程



图表 30：闹钟比较是否有相同闹钟存在流程图

4.2.9 获取闹钟信息----alarm_Get()

1、函数功能介绍

该函数用于返回指定 id 号的闹钟信息结构体。

2、函数主体

```
//*********************************************************************
//获取闹钟信息:用于传输到闹钟设置界面显示
//返回类型：_alarm_obj *     (结构体指针)
//使用参数：id：闹钟数组的标号
//*********************************************************************
_alarm_obj alarm_Get(u8 id)
{
    return Alarm[id];
}
```

4.2.10 读取闹钟从 EEPROM 中----alarm_read()

1、函数功能介绍

该函数用于从 EEPROM 中读取闹钟信息到闹钟结构体中，因为 EEPROM 是按照字节存取的，所以每次读出一个字节的信息到闹钟数组中，直至读取完成一个闹钟的读取。

2、函数主体

```
//*********************************************************************
//读取 AT24CXX 里的闹钟信息到数组中
//返回类型：void
//使用参数：id：闹钟数组的标号
//                addr：写入该闹钟的系统起始地址
//*********************************************************************
void alarm_read(u16 addr,u8 id)
{
    AT24CXX_Read(addr,&Alarm[id].hour,1);//读取闹钟信息到数组中
    AT24CXX_Read(addr+1,&Alarm[id].min,1);
    AT24CXX_Read(addr+2,&Alarm[id].state,1);
    AT24CXX_Read(addr+3,&Alarm[id].weekmask,1);
    AT24CXX_Read(addr+4,&Alarm[id].saveflag,1);
}
```

4.2.11 保存闹钟到 EEPROM 中----alarm_save()

1、函数功能介绍

该函数用于将闹钟结构体中闹钟信息保存到 EEPROM 中，因为 EEPROM 是按照字节存取的，所以每次存储一个字节的信息到闹钟数组中，直至完成一个闹钟的存储。

2、函数主体

```
//*********************************************************************
//写入闹钟信息到 AT24CXX 中
//返回类型：void
//使用参数：id：闹钟数组的标号
```

```
//                addr：写入该闹钟的系统起始地址
//*****************************************************************
void alarm_save(u16 addr,u8 id)
{
    Alarm[id].saveflag&=0X0A;                                    //设置闹钟为已保存状态
    AT24CXX_Write(addr,&Alarm[id].hour,1); //写入闹钟信息到 AT24CXX
    AT24CXX_Write(addr+1,&Alarm[id].min,1);
    AT24CXX_Write(addr+2,&Alarm[id].state,1);
    AT24CXX_Write(addr+3,&Alarm[id].weekmask,1);
    AT24CXX_Write(addr+4,&Alarm[id].saveflag,1);
}
```

4.2.12 闹钟响闹铃----alarm_ring()

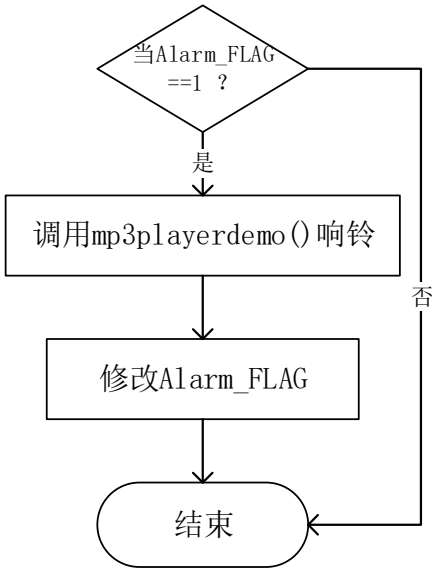
1、函数功能介绍

该函数用于判断当前是否有闹钟响铃。当RTC的Alarm_A达到闹钟设置的时间时，就进入闹钟的中断服务函数，修改 Alarm_FLAG 为 1，然后再判断当其为 1 时则调用 mp3 播放函数，播放闹铃。

2、函数主体

```
//*********************************************************************
//闹钟响闹铃
//返回类型：void
//使用参数：无
//*********************************************************************
void alarm_ring(void)
{
    if(Alarm_FLAG == 1)
    {
        printf("\r\n【闹 钟 响】----------------------------------------------------\r\n");
        mp3PlayerDemo(APP_MUSIC);
        Alarm_FLAG=0;
    }
}
```

3、函数配置流程



图表 31：闹钟响铃流程图

4.3 file ----- FLASH 读取

4.3.1 获取特殊字体点阵----font_set()

1、函数功能介绍

该函数用于读取系统 SPI FLASH 的特殊字体。

为了使我们的时钟界面更美观，所以需要读取系统的数码管字体，以下是定义的特殊字体的路径。

```
//PC2LCD2002 字体取模方法:逐列式,顺向(高位在前),阴码.C51 格式.
//特殊字体:
//数码管字体:ASCII 集+℃(' '+95)
//普通字体:ASCII 集
u8*const APP_ASCII_S6030="1:/SYSTEM/APP/COMMON/fonts60.fon";      //数码管字体 60*30 大数字字体路径
u8*const APP_ASCII_5427="1:/SYSTEM/APP/COMMON/font54.fon";        //普通字体 54*27 大数字字体路径
u8*const APP_ASCII_3618="1:/SYSTEM/APP/COMMON/font36.fon";        //普通字体 36*18 大数字字体路径
u8*const APP_ASCII_2814="1:/SYSTEM/APP/COMMON/font28.fon";        //普通字体 28*14 大数字字体路径
```

如下为字体点阵变量定义：

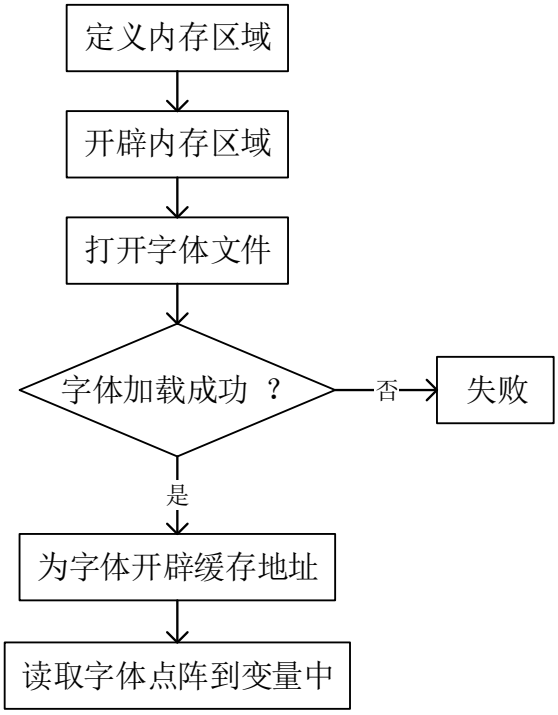
```
u8* asc2_s6030=0; //数码管字体 60*30 大字体点阵集
u8* asc2_5427=0;  //普通字体 54*27 大字体点阵集
u8* asc2_3618=0;  //普通字体 36*18 大字体点阵集
u8* asc2_2814=0;  //普通字体 28*14 大字体点阵集
```

2、函数主体

```
//*********************************************************************
//读取系统中的字体到变量中
//返回类型：u8
//使用参数：无
//*********************************************************************
u8 font_set(void)
{
    u8 res;
    u8 rval=0;                  //设置返回值
    FIL* f_rec=0;       //定义内存区域
    f_rec=(FIL *)gui_memin_malloc(sizeof(FIL));                 //开辟 FIL 字节的内存区域
    //加载特殊字体
    res=f_open(f_rec,(const TCHAR*)APP_ASCII_S6030,FA_READ);//打开文件夹:APP_ASCII_S6030
    if(res==FR_OK)                          //当加载字体成功
    {
        asc2_s6030=(u8*)gui_memex_malloc(f_rec->fsize);   //为大字体开辟缓存地址
        if(asc2_s6030==0)                                 //当字体点阵为空
            rval=1;
        else
            res=f_read(f_rec,asc2_s6030,f_rec->fsize,(UINT*)&br);    //一次读取整个文件
    }
    res=f_open(f_rec,(const TCHAR*)APP_ASCII_2814,FA_READ);//打开文件:APP_ASCII_2814
    if(res==FR_OK)                          //当加载字体成功
    {
        asc2_2814=(u8*)gui_memex_malloc(f_rec->fsize);    //为大字体开辟缓存地址
        if(asc2_2814==0)//当字体点阵为空
            rval=1;
        else
            res=f_read(f_rec,asc2_2814,f_rec->fsize,(UINT*)&br);     //一次读取整个文件
    }
    res=f_open(f_rec,(const TCHAR*)APP_ASCII_3618,FA_READ);//打开文件:APP_ASCII_3618
    if(res==FR_OK)                          //当加载字体成功
    {
        asc2_3618=(u8*)gui_memex_malloc(f_rec->fsize);    //为大字体开辟缓存地址
        if(asc2_3618==0)//当字体点阵为空
            rval=1;
        else
            res=f_read(f_rec,asc2_3618,f_rec->fsize,(UINT*)&br);     //一次读取整个文件
    }
    res=f_open(f_rec,(const TCHAR*)APP_ASCII_5427,FA_READ);//打开文件:APP_ASCII_5427
    if(res==FR_OK)                          //当加载字体成功
    {
        asc2_5427=(u8*)gui_memex_malloc(f_rec->fsize);    //为大字体开辟缓存地址
```

```
        if(asc2_5427==0)//当字体点阵为空
            rval=1;
        else
            res=f_read(f_rec,asc2_5427,f_rec->fsize,(UINT*)&br);      //一次读取整个文件
    }
    rval=0;
    f_close(f_rec);//关闭文件流
    return rval;
}
```

3、函数配置流程



图表 32：特殊字体设置流程图

4.4 mp3play ----- MP3 解码播放

4.4.1 MP3 音频播放----mp3PlayerDemo()

1、函数功能介绍

该函数用于解码 mp3 格式的音乐文件，通过喇叭播放出来。

由于我们没有 SD 卡，所以只能读取到 FLASH 上的一首 mp3 音乐文件，将其解码播放出来。并设置了通过按键来控制其停止播放。

如下为 mp3 音乐文件的路径。

```
u8*const APP_MUSIC="1:/测试用文件/这条街.mp3";                    //音乐路径
```

2、函数主体

```
uint8_t mp3PlayerDemo(u8*const mp3file)
{
    uint8_t res,key,t;
    uint8_t *read_ptr=inputbuf;
    uint32_t frames=0;
    int err=0, i=0, outputSamps=0;
    int   read_offset = 0;                  /*  读偏移指针  */
    int   bytes_left = 0;                   /*  剩余字节数  */
    mp3player.ucFreq=I2S_AudioFreq_Default;      //设置采样频率
    mp3player.ucStatus=STA_IDLE;                                    //状态
    mp3player.ucVolume=40;                                                  //音量
    mp3ctrl=mymalloc(SRAMIN,sizeof(__mp3ctrl)); //为 MP3 控制结构体分配空间
    result=f_open(&file_1,(const TCHAR*)mp3file,FA_READ);          //读取 MP3 文件
    //当读取文件失败时，向串口输出错误信息
    if(result!=FR_OK)
    {
        printf("Open mp3file :%s fail!!!->%d\r\n",mp3file,result);
        result = f_close (&file_1);
        return 0xf0;    /*  停止播放  */
    }
    printf("当前播放文件  -> %s\n",mp3file);
    res=mp3_get_info((u8 *)mp3file,mp3ctrl); //获取播放的 mp3 文件的信息
    if(res!=0)return 0xf4;
    //初始化 MP3 解码器
    Mp3Decoder = MP3InitDecoder();
    if(Mp3Decoder==0)
    {
        printf("初始化 helix 解码库设备\n");
        return 0xf1;    /*  停止播放  */
    }
    printf("初始化中...\n");
    delay_ms(10); /*  延迟一段时间，等待 I2S 中断结束  */
    //不开启是不会响的
    WM8978_ADDA_Cfg(1,0);  //开启 DAC
    WM8978_Input_Cfg(0,0,0);//关闭输入通道
    WM8978_Output_Cfg(1,0);  //开启 DAC 输出
    WM8978_I2S_Cfg(2,0);//飞利浦标准,16 位数据长度
    /*   初始化并配置 I2S    */
    I2S_Play_Stop();
    //     I2S_GPIO_Config();
    I2Sx_Mode_Config(I2S_Standard_Phillips,I2S_DataFormat_16b,mp3player.ucFreq);
    i2s_tx_callback=MP3Player_I2S_DMA_TX_Callback;                          //回调函数
    I2Sx_TX_DMA_Init((uint16_t *)outbuffer[0],(uint16_t *)outbuffer[1],MP3BUFFER_SIZE);
```

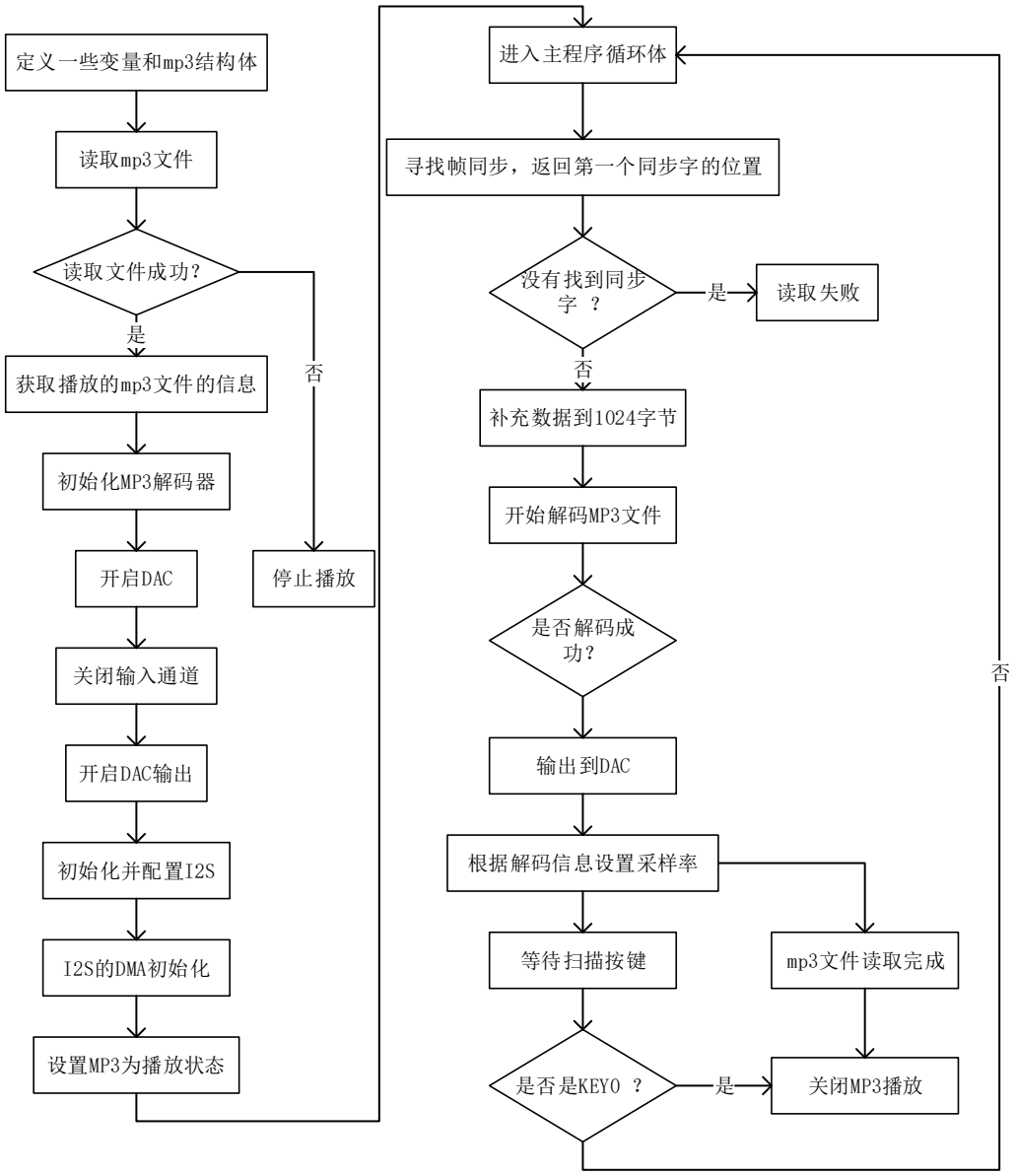
```
    //I2S,DMA 初始化
    bufflag=0;
    Isread=0;
    mp3player.ucStatus = STA_PLAYING;       /* 放音状态 */
  result=f_read(&file_1,inputbuf,INPUTBUF_SIZE,&bw1);
    //读取失败
    if(result!=FR_OK)
    {
        printf("读取%s 失败 -> %d\r\n",mp3file,result);
        MP3FreeDecoder(Mp3Decoder);//释放内存
        return 0xf2;
    }
    read_ptr=inputbuf;
    bytes_left=bw1;
    audiodev.status=3;//开始播放+非暂停
    /* 进入主程序循环体 */
    while(mp3player.ucStatus == STA_PLAYING)
    {
        //寻找帧同步，返回第一个同步字的位置
        read_offset = MP3FindSyncWord(read_ptr, bytes_left);
        //没有找到同步字
        if(read_offset < 0)

        {
            result=f_read(&file_1,inputbuf,INPUTBUF_SIZE,&bw1);//读取
            if(result!=FR_OK)
            {
                printf("读取%s 失败 -> %d\r\n",mp3file,result);
                break;
            }
            read_ptr=inputbuf;
            bytes_left=bw1;//剩余未读字节数
            continue;
        }

        read_ptr += read_offset;                    //偏移至同步字的位置
        bytes_left -= read_offset;                  //同步字之后的数据大小
        if(bytes_left < 1024)                               //补充数据
        {
            /* 注意这个地方因为采用的是 DMA 读取，所以一定要 4 字节对齐 */
            i=(uint32_t)(bytes_left)&3;                                     //判断多余的字节
            if(i) i=4-i;                                                            //需要补充
的字节
            memcpy(inputbuf+i, read_ptr, bytes_left);   //从对齐位置开始复制
```

```
            read_ptr = inputbuf+i;                                  //指向数据对齐位置
            //补充数据
            result = f_read(&file_1, inputbuf+bytes_left+i, INPUTBUF_SIZE-bytes_left-i, &bw1);
            bytes_left += bw1;                                      //有效数据流大小
        }
        //开始解码 参数：mp3 解码结构体、输入流指针、输入流大小、输出流指针、数据格式
        err = MP3Decode(Mp3Decoder, &read_ptr, &bytes_left, outbuffer[bufflag], 0);

        frames++;
        //错误处理
        if (err != ERR_MP3_NONE)
        {
            switch (err)
            {
                case ERR_MP3_INDATA_UNDERFLOW:
                    printf("ERR_MP3_INDATA_UNDERFLOW\r\n");
                    result = f_read(&file_1, inputbuf, INPUTBUF_SIZE, &bw1);
                    read_ptr = inputbuf;
                    bytes_left = bw1;
                    break;
                case ERR_MP3_MAINDATA_UNDERFLOW:
                    /* do nothing - next call to decode will provide more mainData */
                    printf("ERR_MP3_MAINDATA_UNDERFLOW\r\n");
                    break;
                default:
                    printf("UNKNOWN ERROR:%d\r\n", err);
                    // 跳过此帧
                    if (bytes_left > 0)
                    {
                        bytes_left --;
                        read_ptr ++;
                    }
                    break;
            }
            Isread=1;
        }
        else        //解码无错误，准备把数据输出到 PCM
        {
            MP3GetLastFrameInfo(Mp3Decoder, &Mp3FrameInfo);         //获取解码信息

            /* 输出到 DAC */
            outputSamps = Mp3FrameInfo.outputSamps;                 //PCM 数据
个数
            if (outputSamps > 0)
```

```
            {
                if (Mp3FrameInfo.nChans == 1)   //单声道
                {
                    //单声道数据需要复制一份到另一个声道
                    for (i = outputSamps - 1; i >= 0; i--)
                    {
                        outbuffer[bufflag][i * 2] = outbuffer[bufflag][i];
                        outbuffer[bufflag][i * 2 + 1] = outbuffer[bufflag][i];
                    }
                    outputSamps *= 2;
                }//if (Mp3FrameInfo.nChans == 1)      //单声道
            }//if (outputSamps > 0)

            /* 根据解码信息设置采样率 */
            if (Mp3FrameInfo.samprate != mp3player.ucFreq)      //采样率
            {
                mp3player.ucFreq = Mp3FrameInfo.samprate;

                printf(" \r\n Bitrate          %dKbps", Mp3FrameInfo.bitrate/1000);
                printf(" \r\n Samprate         %dHz", mp3player.ucFreq);
                printf(" \r\n BitsPerSample %db", Mp3FrameInfo.bitsPerSample);
                printf(" \r\n nChans            %d", Mp3FrameInfo.nChans);
                printf(" \r\n Layer             %d", Mp3FrameInfo.layer);
                printf(" \r\n Version           %d", Mp3FrameInfo.version);
                printf(" \r\n OutputSamps     %d", Mp3FrameInfo.outputSamps);
                printf("\r\n");
                //I2S_AudioFreq_Default = 2，正常的帧，每次都要改速率
                if(mp3player.ucFreq >= I2S_AudioFreq_Default)
                {
                    //根据采样率修改 I2S 速率

    I2Sx_Mode_Config(I2S_Standard_Phillips,I2S_DataFormat_16b,mp3player.ucFreq);
                    I2Sx_TX_DMA_Init((uint16_t *)outbuffer[0],(uint16_t
*)outbuffer[1],outputSamps);
                }
        //          audio_start();  //包含了
                    I2S_Play_Start();

                }
            }//else 解码正常

            if(file_1.fptr==file_1.fsize)          //mp3 文件读取完成，退出
            {
                printf("END\r\n");
```

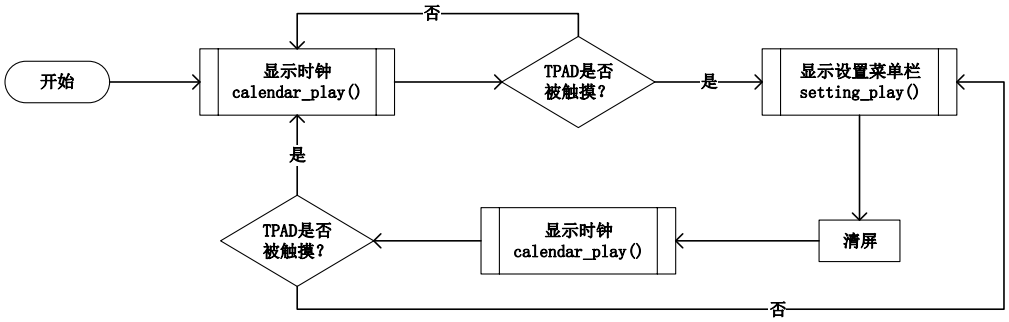
```
            res=KEY0_PRES;
            break;
        }

        while(Isread==0)   //等待填充完毕
        {
        }
        Isread=0;
        while(1)            //按键检测
        {
            key=KEY_Scan(0);
            if(key==WKUP_PRES)          //暂停
            {
                if(audiodev.status&0X01)audiodev.status&=~(1<<0);
                else audiodev.status|=0X01;
            }
            if(key==KEY2_PRES||key==KEY0_PRES)//关闭
            {
                res=key;
                mp3player.ucStatus = STA_IDLE;
                break;
            }
            mp3_get_curtime(&file_1,mp3ctrl); //获取解码信息
            t++;
            if(t==20)
            {
                t=0;
                LED0=!LED0;     //LED0 闪烁
            }
            if((audiodev.status&0X01)==0)delay_ms(10);
            else break;
        }
    }
    I2S_Play_Stop();   //播放停止
    MP3FreeDecoder(Mp3Decoder);//释放内存
    f_close(&file_1);   //关闭文件
    return res;
}
```

3、函数配置流程



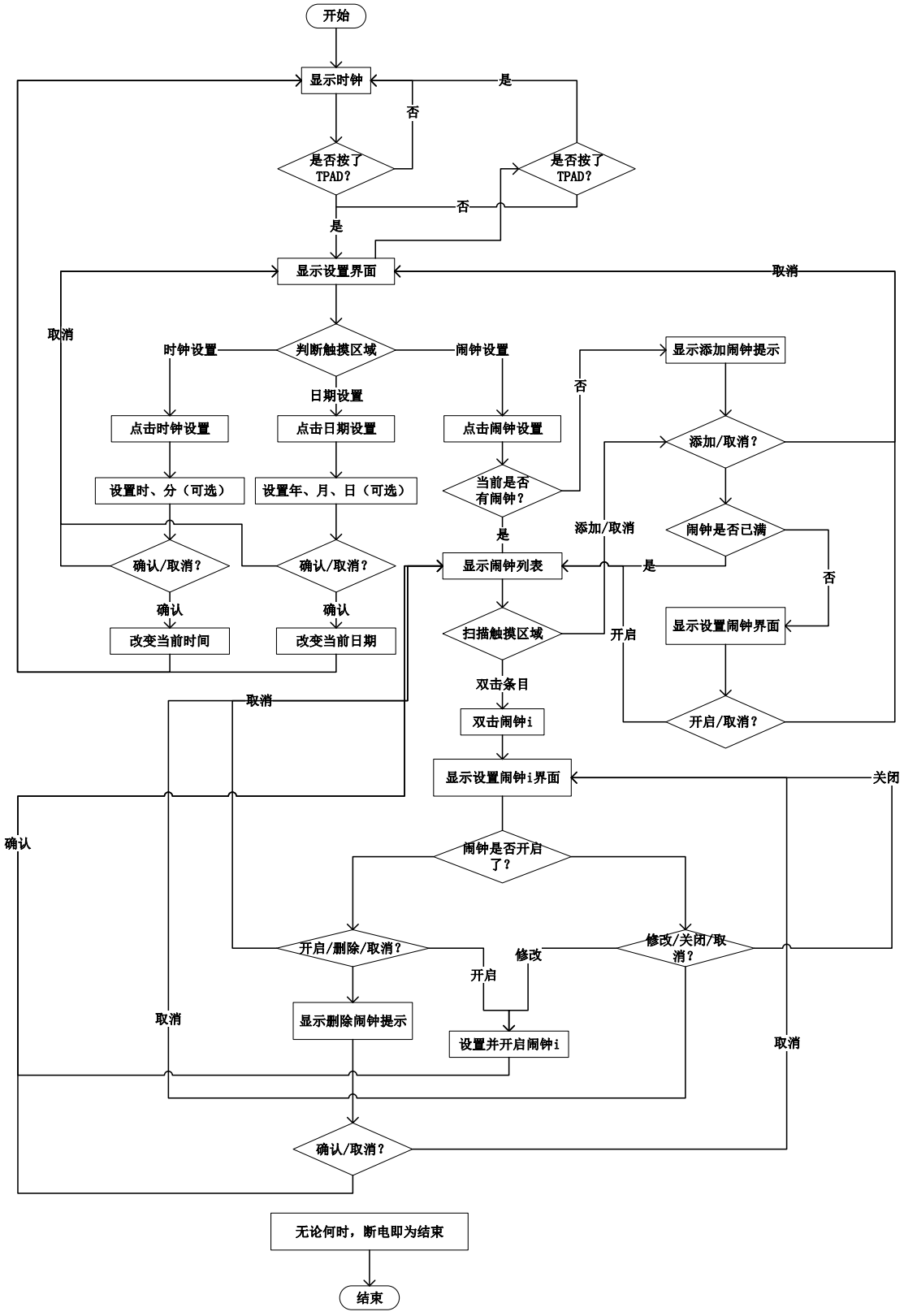
图表 33：MP3 播放流程图

4.5 main ----- 主函数



图表 34：主函数流程图

4.6 组员 nosteglic 负责的 GUI 界面流程图



图表 35：GUI 界面流程图

五、 实现效果

5.1 时钟界面

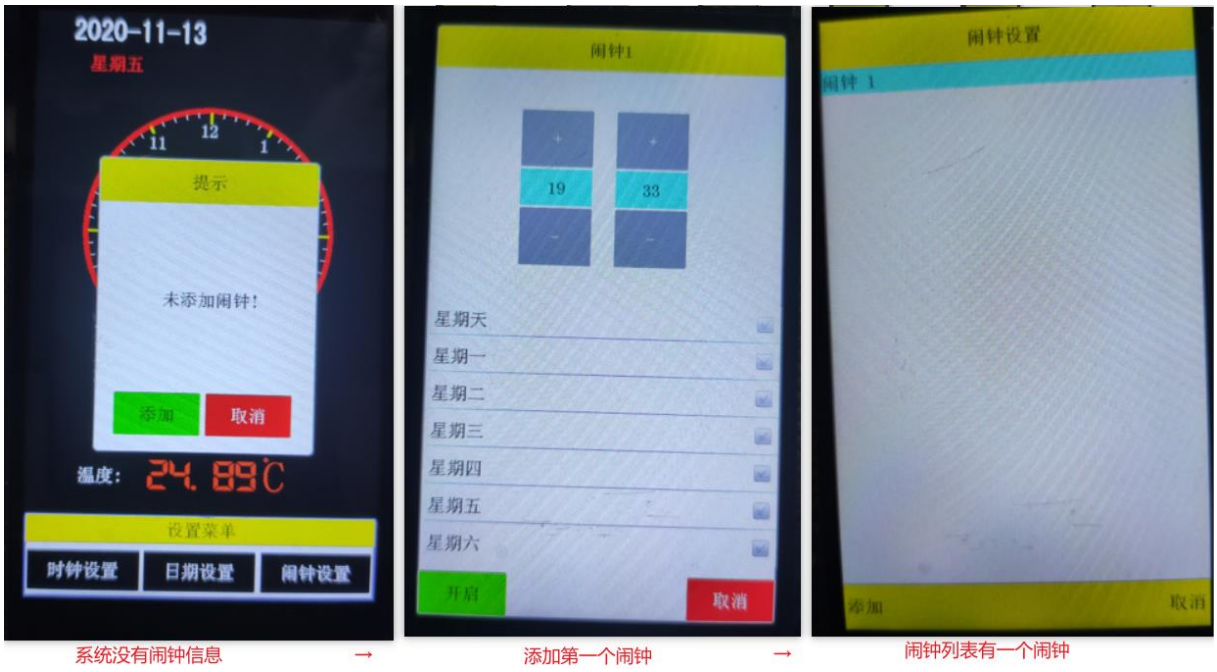
图表 36：时钟/设置界面

5.2 修改日期、时间界面

图表 37：修改时间界面

图表 38：修改日期界面

5.3 闹钟设置界面



图表 39：添加第一个闹钟界面

图表 40：闹钟信息设置界面

图表 41：相同闹钟提醒界面

当闹钟处于关闭状态时 → 可以关闭闹钟 → 跳出删除提示信息

图表 42：删除闹钟提醒界面

系统最多有8个闹钟 → 再添加闹钟时会提示"闹钟已满"！

图表 43：闹钟已满提醒界面

5.4 串口调试界面

图表 44：系统调试界面

六、 总结

6.1 最终实现的功能

1、表盘、日历、时间、内部温度显示在 LCD 屏上；

2、整点报时、半小时报时；

3、触摸设置时钟时间：时、分；

4、触摸设置日历：年、月、日；

5、闹钟设置，包括：

（1）添加和删除闹钟

（2）开启和关闭闹钟

（3）设置闹钟的时间

（4）设置闹钟的模式：按星期

（5）显示已设置的闹钟

（6）未添加闹钟提醒

（7）闹钟已满提醒

（8）闹钟删除提醒

（9）闹钟响铃播放 mp3 歌曲，停止闹钟

6、修改某个变量，可以清空实验板已有闹钟；

6.2 小组分工

本次嵌入式课程设计由我和 nosteglic 同学协作完成。我负责后端的时钟和闹钟函数接口，nosteglic 同学负责前端 GUI 界面开发。

在设计过程中，我们先讨论好各自定义的函数模块所需要传递的参数，以及参数的要求，增量修改我们设计的程序模型。我们首先实现了时钟的显示以及显示日期、时间等基础功能，再进行闹钟部分的设计。此次设计的难点着重在闹钟部分，直到结束依旧有很大的改善空间。

6.3 课程设计中遇到的问题及解决办法

6.3.1 个人遇到的问题

1、特殊字体显示问题

因为导入的 font 字体中，只有小号的字体，这样显示的字体不美观，所以参照了综合测试实验中的字体，将其中的“读取特殊字体”部分摘出来使用，使得我们的时间和温度显示为数码管字体。

2、画表盘数字

参照综合测试实验绘制了我们需要的时钟表盘。但从用户的角度出发，时钟上有小时的数字显示更加直观，所以在绘制表盘时也绘制了数字，使得表盘更美观。

3、更新日历时，只有修改日才更新当前日历的问题

在一开始编写 calendar_play()函数时，只考虑到在 0 点时时间切换到第二天时，需要更新屏幕上显示的日历信息。但在增加了修改日期的模块之后，修改日期的任何一项信息都要立即刷新显示的日期，所以将函数中的判断条件进行修改，只要年、月、日有任何一项变化都立即刷新日期。

4、相隔一分钟的闹钟只响前一个的问题

因为最初比较下一个要设置的闹钟时，只比较了闹钟的小时和分钟，所以当设置了两个只差一分钟的闹钟时，第一个闹钟响铃过后，不会立即将下一个闹钟设置进入，会在这一分钟过后，才设置下一个闹钟，但此时下一个闹钟已经过期了，所以在比较脑中的时候还需要比较“秒”，这样在当前闹钟响过之后就能立即将下一个闹钟设置到

Alarm_A 中。

5、最近需要响铃的闹钟重复写入 Alarm_A 问题

因为闹钟调度是放在 calendar_play()之中的，时间不断变化时，闹钟也会不断调度，所以当每次判断的是相同的闹钟时，就会重复将这个闹钟写入到 RTC 的 Alarm_A 中，所以我们引入一个变量 ring_id 用来存储当前 Alarm_A 中已经设置的闹钟，这样比较下一个要设置的闹钟 id 和 ring_id 就可以判断是否是相同的闹钟，如果是相同的闹钟就不重复写入。

6.3.2 小组共同遇到的问题

1、传参问题，主要是指针

由于函数局部变量需要对结构体的某些变量进行修改，设计指针形式达到传递参数地址的目的。

2、调试方案：printf + while(1)

在编写的模块代码比较多时，很难保证每个模块都是正常工作的，所以在测试时必须知道是哪个模块可能出现了错误，所以我们使用串口调试的 printf 函数将某一步骤的信息输出到串口上，这样就可以根据输出信息判断传递的变量等是否是正确的。

3、模块初始化

在实验过程中，多次由于使用了某些硬件模块，但在主函数中却没有对该模块初始化，因为 MDK5 在编译时不会报错，所以经常会漏掉，导致运行的效果和预期不同。

4、使用全局变量的问题

在编写.c 文件中的函数时定义了很多相关变量，而这些变量有时候需要在其他的.c 文件中用到，所以直接 include 头文件是会报错的，必须 extern 到对应的头文件中，作为外部变量被其他文件使用。

5、各函数接口衔接问题

由于是小组合作，在相互磨合的过程中，不免不遇到了两人定义的函数衔接出现问题，具体表现在：（1）缺少需要的参数；（2）A 定义的函数无法作用与 B 定义的函数等。但总体上函数还是封装的较好的，基本上都是逻辑问题，不用大幅度的修改代码。

6.3.3 待改进的问题

1、调试时 printf 的输出问题

因为 MDK5 编码问题，所以在使用 printf 输出中文时，有时会有乱码问题，所以有些地方的调试信息不是很完美，还有待改善。

2、闹钟响铃触摸控制停止问题以及闹钟响铃时时钟不走秒问题

运用 ucosII 来对时钟任务、闹钟任务、闹钟响铃任务进行任务调度。

3、单次闹钟→按星期闹钟切换

因为根据 RTC 定义的闹钟结构体的参数，闹钟可以设置为按星期模式、按天模式，我们选择了按星期模式来设置我们的闹钟，但参考我们手机上的闹钟可以看出，闹钟还应该有“单次模式”，所以单次闹钟和按星期切换的模式切换还有待实现。

6.4 课程设计的收获与心得

本次嵌入式课程设计耗费了三周的时间来设计电子钟，虽然一开始和 nosteglic 商量过想选别的题目，但想到我们之前的课程设计中数字逻辑、微机原理都实现过电子钟，所以也很想尝试一下嵌入式实现的电子钟是什么样的，所以选择了这个题目。

开始的第一周一直在构想我们要设计的电子钟应该是什么样的，参照了综合测试实验中的电子钟、上学期学长做的串口调试的电子钟后，我们决定设计一个“基于触摸屏控制的多功能电子钟”然后我们就开始各自看资料。

之后的两周我们一直在实现电子钟，我负责后端的时钟和闹钟功能实现，nosteglic 负责前端的界面设计。因为上学期疫情在家时购买了开发板，所以我和 nosteglic 就在宿舍写代码、调试，在第二周结束的时候就基本结束了我们的课程设计，但还有很多小问题，就一直在不断调试，最终在第三周结束了课程设计，虽然还有很多需要改进的地方，但整体来讲还是很不错的。

在这次课程设计中对于嵌入式系统也学习到了很多之前上课没有学习到的地方，同时也很感谢老师在之前课上的教导，感谢我的组员 nosteglic 和我一起学习讨论，这个过程是非常开心和有收获的。