文档类型	开发文档
保密级别	机密

标准教程

名　称：GD32F3 苹果派开发进阶教程-附加篇

编　号：

版本号：V1.0.0

主　编：深圳市乐育科技有限公司

副主编：

主　审：

目　录

版本

序	修订日期	版本	修订内容	修订人

号		号		
1	2021/04/25	1.0.0	初稿	

内容简介

本手册为 GD32F3 苹果派开发进阶教程的附加篇，主要介绍了 RS232 通信、RS485 通信、呼吸灯、电容触摸按键和读写内部 Flash 共 5 个实验。这 5 个实验的例程同样存放于配套资料包中，例程代码规范和学习手册编写规范与 GD32F3 苹果派开发进阶教程完全一致。

1 RS232 通信实验

RS232 串口通信在工业上十分常用，RS232 是由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家和计算机终端生产厂商共同指定的用于串行通信的标准，该标准不仅要求通信双方采用一个标准的连接器，而且规定了连接器的每个引脚的信号内容和信号电平。本章重点介绍 RS232 串口通信的通信协议，最后通过设计一个 RS232 通信实验介绍 RS232 串口的使用方法。

1.1 实验内容

本章的主要内容是学习 RS232 通信原理、GD32F3 苹果派开发板上的 RS232 串口硬件电路以及相关通信协议，然后基于开发板设计一个 RS232 通信实验。使用 RS232 串口转 USB 连接线连接开发板与计算机，在计算机上通过串口助手向开发板发送一字节数据，开发板收到之后，将数据显示在 LCD 屏上，还可以通过 LCD 屏上的 GUI 向计算机发送数据，计算机收到的数据将在串口助手上显示。例如，计算机通过串口助手向开发板发送“15”，开发板接收后，在 LCD 屏上的终端显示“15”，在 LCD 的 UI 上输入并发送“12”，在计算机的串口助手上显示“12”。

1.2 实验原理

1.2.1 RS232 与 UART 的关系

UART 是通用异步串行收发器，是一种用于进行通信的设备。通信双方只要采用相同的帧格式和波特率，就可以仅通过两条信号线（TX 和 RX）完成通信过程，帧格式和波特率由 UART 通信协议规定。

串行数据通过硬件电路在设备间传输时，该硬件电路需要遵循一个电平标准，电平标准规定了信号线上逻辑 1 和逻辑 0 所对应的电压值范围。例如，MCU 与 SOC 使用 UART 通信时，遵循 UART 通信协议，通常采用 TTL 电平标准。在 TTL 电平标准中，逻辑 1 用高电平表示，逻辑 0 用低电平表示，这里的高低电平为范围值，一般规定，引脚作为输出时，电压低于 0.4V 稳定输出低电平，电压高于 2.4V 稳定输出高电平；引脚作为输入时，电压低于 0.8V 稳定输入低电平，电压高于 2V 稳定输入高电平。微控制器通常也采用 TTL 电平标准，但其对引脚输入输出高低电平的电压范围有额外的规定，实际应用时需要参考数据手册。

而 RS232 是一种标准，是对电气特性和物理特性的规定，作用于数据的传输通路上，并不

包含对数据的处理方式。RS232 的电平标准：+3V～+15V 表示逻辑 0，-3V～-15V 表示逻辑 1。由于电平标准的差异，在进行 RS232 通信时，需要使用 RS232 驱动芯片将 MCU 传输过来的 TTL 电平信号转换为 RS232 电平信号，然后再通过 RS232 规定的物理接口（DB9 或 DB25）将信号输出。而 MCU 与 RS232 驱动芯片之间通过 UART 通信，同样遵循 UART 通信协议。因此，RS232 通信与单片机的 UART 通信相比，电平标准和物理接口不一致，但二者均遵循 UART 通信协议。

1.2.2 RS232 串口通信协议

RS232 串口通信协议分为物理层与协议层，物理层采用的是 RS232 标准，RS232 标准的全称是 EIA-RS-232 标准（Electronic Industry AssociaTIon- ecommeded standard-232），被定义为一种在低速率串行通信中增加通信距离的单端标准。RS232 是计算机与通信工业中应用最广泛的一种串行接口，其协议层与 UART 串口通信协议的协议层相同，这里不再赘述，下面具体介绍 RS232 串口通信协议物理层。

1．RS232 电平详解

对于 RS232 标准，一般逻辑 1 为-3~-15V，逻辑 0 为+3~+15V。在图 1-1 中，为了增加串口通讯的远距离传输及抗干扰能力，用-15V 表示逻辑 1，用+15V 表示逻辑 0。对于 TTL 电平标准：理想情况下，用 5V 或 3.3V 表示逻辑 1，用 0V 表示逻辑 0。RS232 的电压变化范围在 30V 左右，而 TTL 的电压变化范围在 5V 或 3.3V 左右，电压差异明显，无法直接用 TTL 电平驱动使用 RS232 电平标准的器件。GD32F3 苹果派开发板使用 GD32F30x 系列微控制器，由于微控制器一般采用 TTL 电平标准，因此需要通过电平转换芯片将 TTL 电平转换为 RS232 电平。

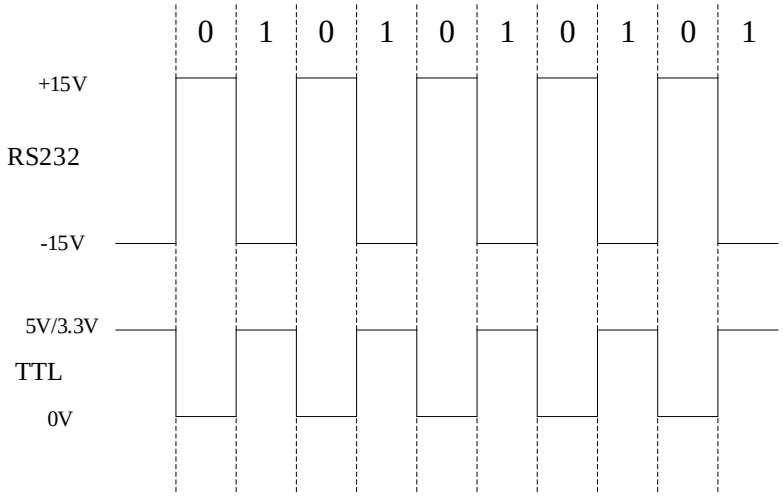

图 1-1　RS232 与 TTL 电平

2．RS232 通信方式详解

两个设备之间使用 RS232 串口进行通信的过程如图 1-2 所示，若设备 A 发送数据到设备 B，设备 A 先使用电平转换芯片（如 SP3232，MAX3232 等）将控制器 A 发出的 TTL 电平信号转换为 RS232 电平信号，然后通过串口线连接 AB 设备上的 DB9 接口，串口线中使用 RS232 标准传输数据信号。当 B 设备接收到信号时，由于 RS232 电平标准的信号不能直接被控制器 B 识别，因此需要先经过电平转换芯片将 RS232 电平信号转换为 TTL 电平信号再传输给控制器 B 完成通信，设备 B 同样也可以通过以上流程发送数据到设备 A。

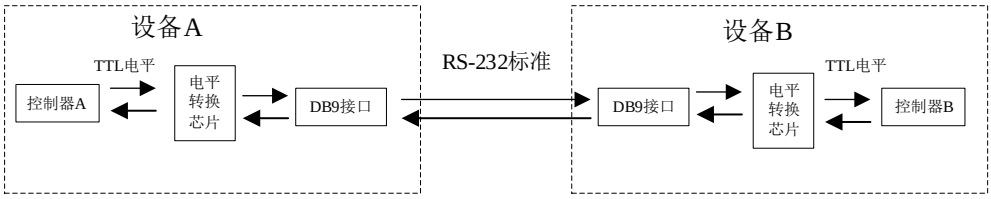

图 1-2　设备 A 与 B 使用 RS232 串口通信

开发板通过 RS232 串口与计算机之间进行通信的过程如图 1-3 所示，若数据从开发板发送到计算机，则首先通过电平转换芯片将 TTL 电平转换为 RS232 电平，然后使用如图 1-4 所示的 RS232 串口转 USB 连接线连接开发板和计算机，将 RS232 电平转换为 USB 电平。由于 USB 电平与 TTL 电平相似，所以在计算机内部不需要再进行一次电平转换，数据从计算机发送到开发板的电平转换过程类似。

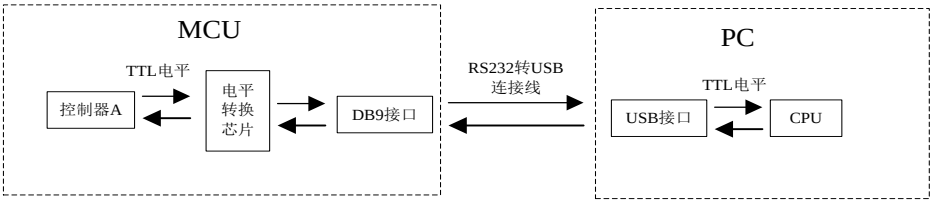

图 1-3　开发板通过 DB9 串口与计算机通信

图 1-4　RS232 串口转 USB 连接线

3．RS232 串行接口标准

RS232 是异步全双工通信，与 UART 一致，都没有时钟线，所以通信双方需要约定好波特

率，即每个码元的长度，以便对信号进行解码。RS232 有两条数据线可以实现双向接收与发送，属于串行传输，因此速度比并行传输慢。RS232 采取不平衡传输方式通信，即单端通信，接收与发送信号都是相对于信号地，容易产生共模干扰。发送数据时，输出正电平在+5~+15V 之间，负电平在-5V~-15V 之间。接收器典型工作电平在+3~+12V 与-3~-12V 之间。由于发送电平与接收电平的差仅为 2V~3V 左右，所以其共模抑制能力较差。由于存在共模干扰等问题，RS232 串口传输距离有限，虽然理论上可以达到 50 英尺，但在实际应用中一般不超过 15 米。

4．RS232 连接器

RS232 串口通信一般可以使用 DB9 接口和 DB25 接口，由于 DB25 接口的引脚过多，占据较大空间的同时引脚利用率低，因此在 GD32F3 苹果派开发板中使用 DB9 接口。DB9 分为公头与母头，如图 1-5 所示。公头与母头最大的区别在于 2 号引脚与 3 号引脚功能相反，DB9 共有 9 个引脚，各个引脚的功能如表 1-1 所示。在本实验中，仅实现了 RS232 串口的接收与发送功能，因此只需要三条信号线：即发送数据线 TXD、接收数据线 RXD 和地线 GND。注意，如果通信双方都使用 DB9 接口，则二者的 TXD 与 RXD 要交叉连接，所以串口线分为直连线和交叉线两种，可以使用万用表测量，若接口两端 2 号引脚和 2 号引脚连通，3 号引脚和 3 号引脚连通则为直连串口线；若一端 2 号引脚与另一端 3 号引脚连通，则为交叉串口线。图 1-6 为串口线实物图。

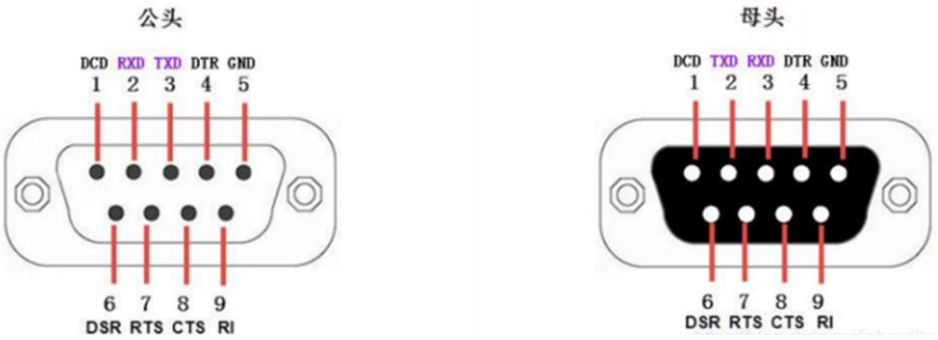

图 1-5　公头与母头

图 1-6　串口线实物图

表 1-1　DB9 接口引脚功能说明

引脚号	信号方向来自	缩写	描述
1	调制解调器	CD	载波检测
2	调制解调器	RXD	接收数据
3	PC	TXD	发送数据
4	PC	DTR	数据终端准备好
5		GND	信号地
6	调制解调器	DSR	通讯设备准备好
7	PC	RTS	请求发送
8	调制解调器	CTS	允许发送
9	调制解调器	RI	响铃指示器

5．RS232 硬件流控制

RS232 的设备可以分为数据终端设备（DTE）和数据通信设备（DCE）两类，这种分类定义了不同的线路用来发送和接受信号。其中 DTR 和 RTS 信号由 DTE 产生，DSR、CTS、DCD 和 RI 信号由 DCE 产生，GND 信号为公共端。首先介绍 DTR 和 DSR，DTE 发送 DTR 信号通知 DCE 准备通信，DCE 开启 DSR 来应答，然后通过 RTS 和 CTS 控制数据传输，当 DCE 准备接收数据时，使 CTS 为 ON（1），当 DCE 不能接收更多数据时，使 CTS 为 OFF（0），类似的，当 DTE 可以接收数据时，使 RTS 为 ON（1），不能接收数据时则 RTS 为 OFF（0）。除了硬件流控制，还可以通过 XON 和 XOFF 进行软件流控制。在本实验中，仅使用 TXD 和 RXD 进行数据通信，并未采用硬件流控制以及软件流控制。

1.2.3 RS232 电路原理图

RS232 电路参考 SP3232 芯片典型工作电路设计而成，主要由 SP3232 电压转换芯片和 DB9 接口组成。SP3232 芯片的工作电压为 3.3V，其内部含有一个高效电荷泵，可以将输入的 3.3~5V 电压转化为符合 RS232 电平标准的电压信号，此外，还有两个 RS232 驱动器以及两个 RS232 接收器，其中驱动器具有反相发送的特点，能将 TTL 电平或 CMOS 电平转换为与输入逻辑电平相反的 RS232 电平。SP3232 芯片的各引脚说明如表 1-2 所示。

表 1-2　SP3232 芯片引脚说明

引脚号	引脚名	用途	引脚号	引脚名	用途
1	C1+	倍压电荷泵电容的正极	9	R2OUT	TTL/CMOS 接收器输出
2	V+	电荷泵产生的+5.5V 电压	10	T2IN	TTL/CMOS 驱动器输入
3	C1-	倍压电荷泵电容的负极	11	T1IN	TTL/CMOS 驱动器输入
4	C2+	反相电荷泵电容的正极	12	R1OUT	TTL/CMOS 接收器输出
5	C2-	反相电荷泵电容的负极	13	R1IN	RS232 接收器输入
6	V-	电荷泵产生的-5.5V 电压	14	T1OUT	RS232 驱动器输出

7	T2OUT	RS232 驱动器输出	15	GND	地
8	R2IN	RS232 接收器输入	16	VCC	+3.3V~+5.5V 电源电压

在图 1-7 中，13 号引脚 R1IN 为 RS232 接收器输入，连接到 DB9 接口的数据接收引脚 DB9_RX，14 号引脚 T1OUT 为 RS232 驱动器输出，连接到 DB9 接口的数据发送引脚 DB9_TX。如果通过跳线帽将 RS232_RX 与 USART1_TX 相连，将 RS232_TX 与 USART1_RX 相连，那么 11 号引脚 T1IN 为逻辑输入，连接到 RS232_RX（USART1_TX）可以读取单片机发送的数据，12 号引脚 T2IN 为逻辑输出，连接到 RS232_TX（USART1_RX）可以向单片机发送数据，从而实现 RS232 通信。此外，图中放置了两个 DB9 接口，一个为公头，一个为母头，方便不同连接线的使用。

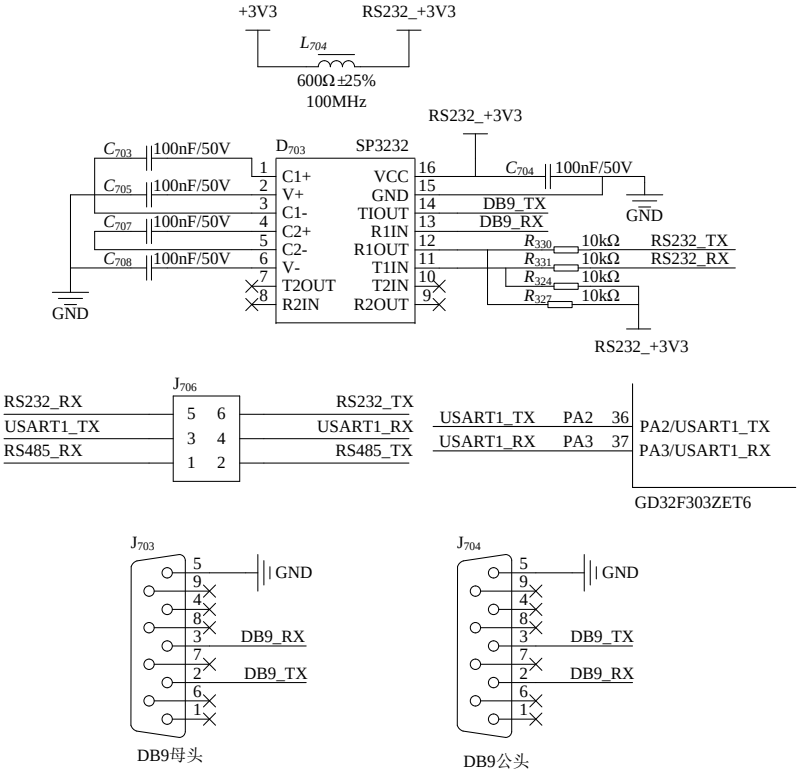

图 1-7　RS232 模块电路图

1.2.4　串口数据接收和数据发送流程

RS232 使用的串行通信协议与 USART 相同。本实验中的串口模块包含串口发送缓冲区和串口接收缓冲区，这两个缓冲区均为结构体，串口的数据接收和发送过程如图 1-8 所示。数据发送过程（写串口）分为三步：（1）调用 WriteUART1 函数将待发送的数据通过 EnQueue 函数写入发送缓冲区，同时开启中断使能；（2）当数据发送寄存器为空时，会产生中断，在串口模块的 USART1_IRQHandler 中断服务函数中，通过 ReadSendBuf 函数调用 DeQueue 函数取出发送缓冲区中的数据，再通过 usart_data_transmit 函数将待发送的数据写入 USART 数据发送寄存器（USART_TDATA）；（3）微控制器的硬件会将 USART_TDATA 中的数据写入发送移

位寄存器，然后按位将发送移位寄存器中的数据通过 TX 端口发送出去。数据接收过程（读串口）正好与写串口过程相反，但也分为三步：（1）微控制器的接收移位寄存器在接收到一帧数据时，会由硬件将接收移位寄存器的数据发送到 USART 接收数据寄存器（USART_RDATA），同时产生中断；（2）在串口模块的 USART1_IRQHandler 中断服务函数中，通过 usart_data_receive 函数读取 USART_RDATA，并通过 WriteReceiveBuf 函数调用 EnQueue 函数将接收到的数据写入接收缓冲区；（3）调用 ReadUART1 函数读取接收到的数据。

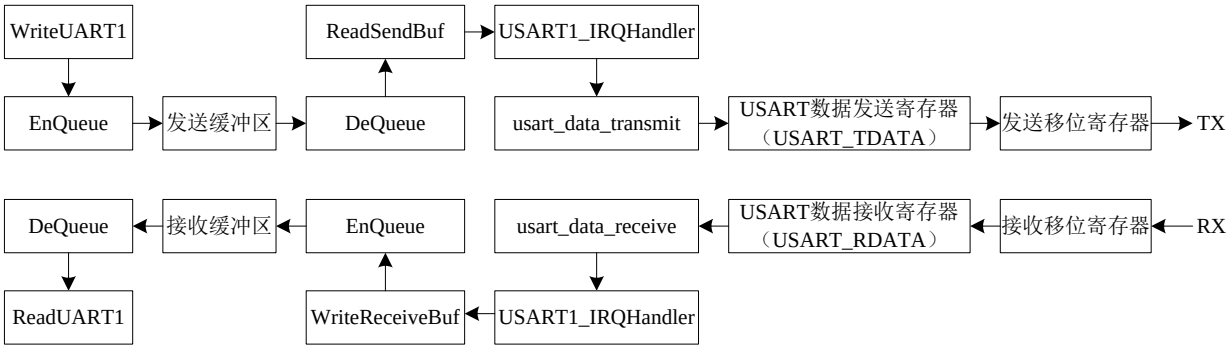

图 1-8　UART1 数据接收和数据发送路径

1.2.5　部分寄存器和固件库函数

本实验涉及的 USART 寄存器包括 USART 控制寄存器 0（USART_CTL0）、USART 控制寄存器 1（USART_CTL1）、USART 波特率寄存器（USART_BAUD）、USART 状态寄存器（USART_STAT）、USART 中断标志清除寄存器（USART_INTC）、USART 数据接收寄存器（USART_RDATA）和 USART 数据发送寄存器（USART_TDATA）。上述寄存器的定义和功能请参考 GD32F30x 系列微控制器的中文版用户手册《GD32F30x_yonghushouce_Rev2.5》，其中介绍串口寄存器的部分位于页码 P432~P458，也可参考英文版用户手册《GD32F30x_User_Manual_Rev2.5》，对应页码为 P454~P482（上述两个手册存放于本书配套资料包“07.数据手册”文件夹下）。

本实验涉及的串口部分固件库函数包括 usart_deinit、usart_baudrate_set、usart_stop_bit_set、usart_word_length_set、usart_parity_config、usart_receive_config、usart_transmit_config、usart_enable、usart_interrupt_enable、usart_interrupt_disable、usart_data_transmit、usart_data_receive、usart_flag_get、usart_flag_clear、usart_interrupt_flag_get 和 usart_interrupt_flag_clear。这些函数在 gd32f30x_usart.h 文件中声明，在 gd32f30x_usart.c 文件中实现。本书配套实验例程所涉及的固件库版本均为“2020-09-30, V2.1.0”。 关于以上固件库函数以及更多其他串口固件库函数的的函数原型、输入输出参数及用法等信息请参考

GD32F30x 系列微控制器的固件库使用指南《GD32F30x_gujiankuyonghuzhinan_Rev1.0》，其中，介绍串口固件库函数的部分位于页码 P631~P675。（上述文件存放于在资料包“07.数据手册”文件夹下）。

1.3 实验代码解析

1.3.1 UART1 文件对

由于在串口通信一章中已经详细介绍了这两个文件，这里不再赘述，具体的代码解释可参考《GD32F3 开发基础教程——基于 GD32F303ZET6》的串口通信实验。

1.3.2 RS232Top 文件对

1．RS232Top.h 文件

在 RS232Top.h 文件的“API 函数声明”区，声明了 2 个 API 函数，如程序清单 1-1 所示。InitRS232Top 函数的主要功能是设置 GUI 界面的显示内容，RS232TopTask 函数主要用于实现接收与发送数据功能。

程序清单 1-1

```
void InitRS232Top(void);   //初始化 RS232 实验应用层模块
void RS232TopTask(void);   //RS232 实验应用层模块任务
```

2．RS232Top.c 文件

在 RS232Top.c 文件的“内部变量”区，定义了 s_structGUIDev 结构体，如程序清单 1-2 所示，s_structGUIDev 为 GUI 设备结构体。

程序清单 1-2

```
static StructGUIDev s_structGUIDev;                //GUI 设备结构体
```

在 RS232Top.c 文件的“内部函数声明”区，声明了 SendProc 函数和 ReadProc 函数，如程序清单 1-3 所示，SendProc 函数在 GUI 发送回调函数中被调用，实现单片机向计算机发送数据，ReadProc 函数在 RS232TopTask 函数中被调用，实现单片机从计算机接收数据。

程序清单 1-3

```
static void SendProc(char* string); //发送处理函数
static void ReadProc(void);                //读串口处理
```

在 RS232Top.c 文件的“内部函数实现”区，首先实现了 SendProc 函数，如程序清单 1-4 所示。

（1）第 7 至 10 行代码：当输入地址为空为空则无数据需要发送，退出函数；否则继续发

送。

（2）第 13 行代码：在每一次发送前清零计数值。

（3）第 14 至 24 行代码：判断（string+i）地址中存放的数据是否为零，不为零则将该数据赋值给 sendData，再通过串口 UART1 的 WriteUART1 写串口函数发送出去，最后计数 i 加一，循环此操作直到 string + i 地址中存放的数据为 0。

（4）第 27 至 30 行代码：在一次发送数据完成后发送回车换行结束，将'\r'回车符与'\n'换行符接连赋值给 sendData，再通过串口 UART1 的写串口函数 WriteUART1 分别发送出去。

程序清单 1-4

```
1.  static void SendProc(char* string)
2.  {
3.      u8 i;             //发送计数
4.      u8 sendData; //发送数据
5.
6.      //校验输入地址是否为空指针
7.      if(NULL == string)
8.      {
9.          return;
10.     }
11.
12.     //通过 UART1 将数据发送出去
13.     i = 0;
14.     while(0 != *(string + i))
15.     {
16.         //获取需要发送的数据
17.         sendData = *(string + i);
18.
19.         //通过串口发送出去
20.         WriteUART1(&sendData, 1);
21.
22.         //发送计数加一
23.         i++;
24.     }
25.
26.     //发送回车换行
27.     sendData = '\r';
28.     WriteUART1(&sendData, 1);
29.     sendData = '\n';
30.     WriteUART1(&sendData, 1);
31. }
```

在 SendProc 函数之后为 ReadProc 函数的实现代码，如程序清单 1-5 所示。

（1）第 3 至 5 行代码：首先定义 s_arrReadBuf 数组作为接收缓冲区，s_iReadCnt 变量用于计数，以及 s_iBeginTime 用于接收到第一个字符时的系统时间，通过 time-s_iBeginTime 记录接收时间。

（2）第 11 行代码：通过 GetSysTime 函数获取系统运行时间。

（3）第 14 至 26 行代码：通过 ReadUART1 函数读取串口数据并储存在 readData 中，当接收到第一个字符时将 time 的值赋给 s_iBeginTime，记录此时系统时间，然后将数据储存到 s_arrReadBuf 缓存区，最后对计数 s_iReadCnt 进行加一操作。

（4）第 29 至 42 行代码：如果接收数据大于 1 位且接收到回车换行符，则向缓冲区加入字符串结尾，对接收计数清零以及通过 s_structDev.showLine 结构体成员输出数据到终端显示。

（5）第 46 至 56 行代码：最后为防止接收错误以及接收数据过长，当超过 250 毫秒还没接收到回车换行则强制更新到终端显示，在输出到终端显示前，需要加上字符串结尾以及清零接收计数值。

程序清单 1-5

```
1.  static void ReadProc(void)
2.  {
3.      static u8    s_arrReadBuf[48] = {0}; //接收缓冲区
4.      static u8    s_iReadCnt       = 0;     //接收计数
5.      static u64 s_iBeginTime       = 0;     //接收到第一个字符时的系统时间
6.
7.      u8    readData;
8.      u64 time;
9.
10.     //获取系统运行时间
11.     time = GetSysTime();
12.
13.     //接收数据处理
14.     while(ReadUART1(&readData, 1))
15.     {
16.         //记录接收到第一个字符时的系统时间
17.         if(0 == s_iReadCnt)
18.         {
19.             s_iBeginTime = time;
20.         }
21.
22.         //将数据储存到缓冲区
23.         s_arrReadBuf[s_iReadCnt] = readData;
24.
25.         //接收计数加一
26.         s_iReadCnt = (s_iReadCnt + 1) % (sizeof(s_arrReadBuf) / sizeof(u8));
27.
28.         //接收到回车换行
29.         if(s_iReadCnt >= 2)
30.         {
31.             if(('\r' == s_arrReadBuf[s_iReadCnt - 2]) && ('\n' == s_arrReadBuf[s_iReadCnt - 1]))
32.             {
33.                 //加上字符串结尾
34.                 s_arrReadBuf[s_iReadCnt] = 0;
35.
```

```
36.                 //接收计数清零
37.                 s_iReadCnt = 0;
38.
39.                 //输出到终端显示
40.                 s_structGUIDev.showLine((char*)s_arrReadBuf);
41.               }
42.           }
43.       }
44.
45.       //超过 250 毫秒还没接收到回车换行则强制更新到终端显示
46.       if((s_iReadCnt > 0) && ((time - s_iBeginTime) > 250))
47.       {
48.           //加上字符串结尾
49.           s_arrReadBuf[s_iReadCnt] = 0;
50.
51.           //接收计数清零
52.           s_iReadCnt = 0;
53.
54.           //输出到终端显示
55.           s_structGUIDev.showLine((char*)s_arrReadBuf);
56.       }
57.   }
```

在 RS232Top.c 文件的“API 函数实现”区，实现了 InitRS232Top 和 RS232TopTask 两个 API 函数，如程序清单 1-6 所示。

（1）第 1 至 14 行代码：InitRS232Top 函数用于对 s_structGUIDev 结构体成员进行赋值操作，实现串口号显示、波特率显示以及通过发送回调函数 SendProc 发送数据到串口，最后通过调用 InitGUI 函数进行初始化 GUI 界面设计从而实现在 LCD 上的显示。

（2）第 16 至 20 行代码：RS232TopTask 函数调用 ReadProc 函数接收串口数据并输出到终端，调用 GUITask 函数执行 GUI 任务。

程序清单 1-6

```
1.    void InitRS232Top(void)
2.    {
3.        //串口号显示
4.        s_structGUIDev.serialPort = "UART1";
5.
6.        //波特率显示
7.        s_structGUIDev.baudRate = "115200";
8.
9.        //发送回调函数
10.       s_structGUIDev.sendCallBack = SendProc;
11.
12.       //初始化 UI 界面设计
13.       InitGUI(&s_structGUIDev);
14.   }
15.
16.   void RS232TopTask(void)
```

```
17.  {
18.      ReadProc(); //接收串口数据并输出到终端
19.      GUITask();   //GUI 任务
20.  }
```

1.3.3 Main.c 文件

在 Main.c 文件的 Proc2msTask 函数中，每 40ms 调用一次 RS232TopTask 函数，实现 RS232 的接收与发送数据功能，如程序清单 1-7 所示。

程序清单 1-7

```
1.   static   void   Proc2msTask(void)
2.   {
3.       static u8 s_iCnt = 0;
4.       if(Get2msFlag())   //判断 2ms 标志状态
5.       {
6.           LEDFlicker(250);//调用闪烁函数
7.
8.           s_iCnt++;
9.           if(s_iCnt >= 20)
10.          {
11.              s_iCnt = 0;
12.              RS232TopTask();
13.          }
14.
15.          Clr2msFlag();     //清除 2ms 标志
16.      }
17.  }
```

1.3.4 实验结果

代码编写完成并编译通过后，下载程序并进行复位。下载完成后，通过跳线帽将开发板上 J_{706} 的 232_RX（RS232_RX）与 PA2（USART1_TX）相连，将 232_TX（RS232_TX）与 PA3（USART1_RX）相连，然后使用 RS232 串口转 USB 连接线连接开发板和计算机，连接好后通过设备管理器查看对应的串口号，并在串口助手中打开该串口，最后进行数据收发：①计算机通过串口助手向开发板发送“15”，开发板接收到数据后，在 LCD 屏上的终端显示“15”；②在 LCD 的 GUI 界面上输入并发送“12”，计算机的串口助手上显示收到数据“12”，如图 1-9 和图 1-10 所示。

图 1-9　计算机端接收和发送数据

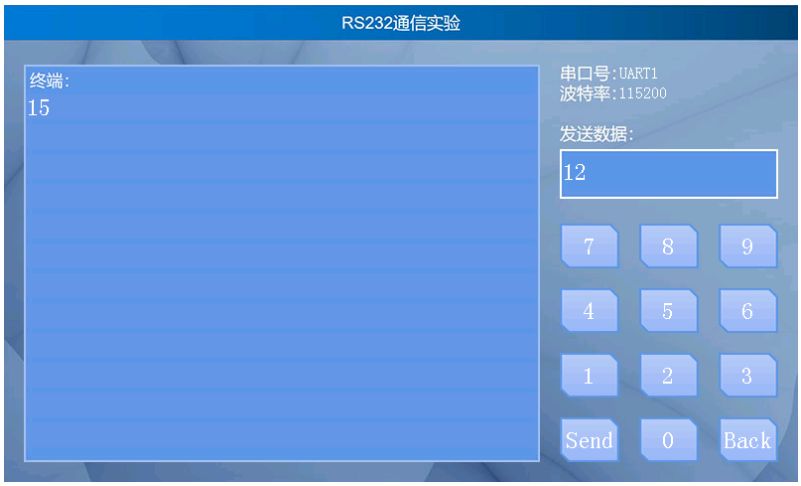

图 1-10　开发板端接收和发送数据

本章任务

在本章实验中，通过 RS232 串口转 USB 连接线实现了 GD32F3 苹果派开发板与计算机之间的 RS232 通信，尝试通过双端 DB9 接口的串口线连接两块开发板，通过 LCD 上的 UI 收发数据，实现两块开发板之间的 RS232 通信。具体要求为：开发板 A 发送数据“1”，开发板 B 的 LCD 屏上终端显示“1”；开发板 B 发送数据“2”，开发板 A 的 LCD 屏上终端显示“2”。

本章习题

1．在 TTL 电平标准和 RS232 电平标准下，逻辑 0 和 1 对应的电压范围分别是多少？

2．简要概括 RS232 通信与 UART 通信的异同。

3．硬件流控制在串口通信过程的作用是什么？

4．本实验使用 GD32F3 苹果派开发板与计算机通信，尝试简述通信流程。

2 RS485 通信实验

RS485（又名 TIA/EIA-485）是隶属于 OSI 模型物理层的，电气特性规定为 2 线、半双工、多点通信的标准，该标准由电信行业协会和电子工业联盟定义。使用该标准的数字通信网络能在远距离条件下以及电子噪声大的环境下有效传输信号。RS485 使得连接本地网络以及多支路通信链路的配置成为可能。本章将介绍 RS485 相关知识，包括 RS485 串口通信的物理层和 RS485 与 UART 的关系等，最后通过设计一个 RS485 通信实验介绍 RS485 串口的使用方法。

2.1 实验内容

本章的主要内容是学习 RS485 通信原理，GD32F3 苹果派开发板上的 RS485 串口硬件电路以及相关通信协议，然后基于开发板设计一个 RS485 通信实验。使用 RS485 串口转 USB 模块连接开发板与计算机，在计算机上通过串口助手向开发板发送一字节数据，开发板收到之后，将数据显示在 LCD 屏上，还可以通过 LCD 屏上的 GUI 向计算机发送数据，计算机收到的数据将在串口助手上显示。例如，计算机通过串口助手向开发板发送“1”，开发板接收后，在 LCD 屏上的终端显示“1”，在 LCD 的 UI 上输入并发送“2”，在计算机的串口助手上显示“2”。

2.2 实验原理

2.2.1 RS485 简介

RS485 是一个仅规定了接收端和发送端的电气特性的通信标准，没有规定或推荐任何数据协议，与 RS232 类似。RS485 用缆线两端的电压差值来传递信号，其特点如下：

（1）接口电平低，不易损坏芯片。RS485 的电平标准：逻辑“1”以两线间的电压差为+2~+6V 表示；逻辑“0”以两线间的电压差为-2~-6V 表示。接口信号电平比 RS232 低，不易损坏接口电路的芯片。

（2）传输速率高。10 米时，RS485 的数据最高传输速率可达 35Mbps，在 1200 米时，传输速度仍可达 100Kbps。

（3）抗干扰能力强。RS485 接口采用平衡驱动器和差分接收器的组合，抗共模干扰能力强，即抗噪声干扰性好。

（4）传输距离远，支持节点多。RS485 总线最长可以支持 1200m 以上传输（速率≤100Kbps），一般最多支持 32 个节点，如果使用特制的 485 芯片，可以支持多达 400 个节点。

由于 RS485 属于低压差分电平，很容易被干扰，而星型、环型网络会引入反射等干扰，所以 RS485 推荐使用在点对点网络中，如线型、总线型连接方式，而不能是星型、环型网络。理想情况下，RS485 线路上需要 2 个匹配电阻，其阻值要求与传输电缆的特性阻抗（一般为 120Ω）相等。若没有特性阻抗，当所有设备都静止或没有能量的时候就会产生噪声。若没有连接终端匹配电阻，会使得发送端产生多个数据信号的边缘信号，导致数据传输出错。RS485 推荐的连接方式是主从连接方式，即一个主机带多个从机，如图 2-1 所示：

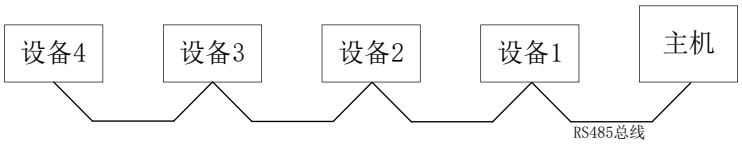

图 2-1　RS485 连接方式

在上述主从连接方式中，一般将匹配电阻添加在总线的起止端，即在主机和设备 4 上各加一个 120 Ω 匹配电阻。

在本章实验中，使用如图 2-2 所示的 RS485 串口转 USB 模块连接 GD32F3 苹果派开发板和计算机实现通信，连接方式如图 2-3 所示：

图 2-2　RS485 串口转 USB 模块

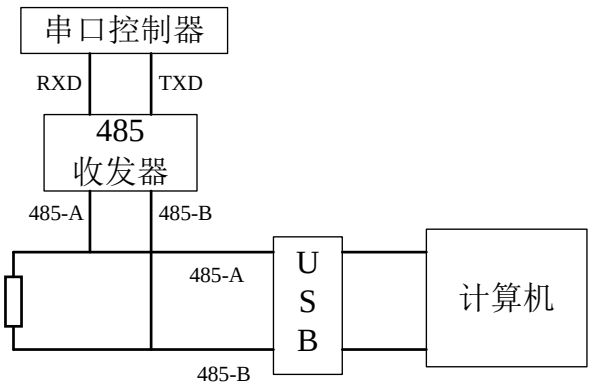

图 2-3　模块和计算机连接图

MCU 与 485 收发器（RS485 驱动芯片）之间通过 UART 通信时，同样遵循 UART 通信协议。因此，RS485 通信与单片机的 UART 通信相比，虽然电平标准和物理接口不一致，但二者均遵循 UART 通信协议。

2.2.2 RS485 电路原理图

GD32F3 苹果派开发板采用 SP3485 芯片作为收发器，该芯片支持 3.3V 供电，最大传输速度可达 10Mbps，支持多达 32 个节点，并且具有输出短路保护功能。芯片内部结构框图如图 2-4 所示，A、B 引脚为总线接口，用于连接 485 总线。RO 为接收输出端，DI 为发送数据收入端，RE 为接收使能信号（低电平有效），DE 为发送使能信号（高电平有效）。

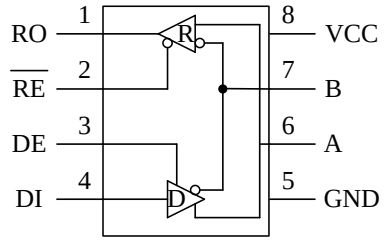

图 2-4 SP3485 芯片框图

如图 2-5 所示，在本章实验中，GD32F303ZET6 微控制器的 USART1_TX（PA2 引脚）和 USART_RX（PA3 引脚）在 J_{706} 处可以通过跳线帽分别与 RS485_RX 和 RS485_TX 相连，而 RS485_RX 和 RS485_TX 分别连接到 SP3485 芯片 DI 及 RO 引脚，RS485_RE（PG11 引脚）连接到 SP3485 芯片的 RE 和 DE 引脚，当 PG11 引脚为低电平时为接收模式，高电平则为发送模式。注意，R_{705} 和 R_{707} 为偏置电阻，用于保证总线空闲时，A、B 之间的电压差大于 200mV（逻辑 1），从而避免由于 A、B 压差不定而导致逻辑错乱。

在本章实验中，通过 ZK-U485 模块（工业级 USB 转 RS485 通讯模块串口线转换器）实现开发板与计算机之间的 485 通信。具体通信过程如下：通过 ZK-U485 模块连接开发板与计算机，通过开发板上的 LCD 屏上的 UI 和计算机上的串口助手互相收发数据实现数据通信。

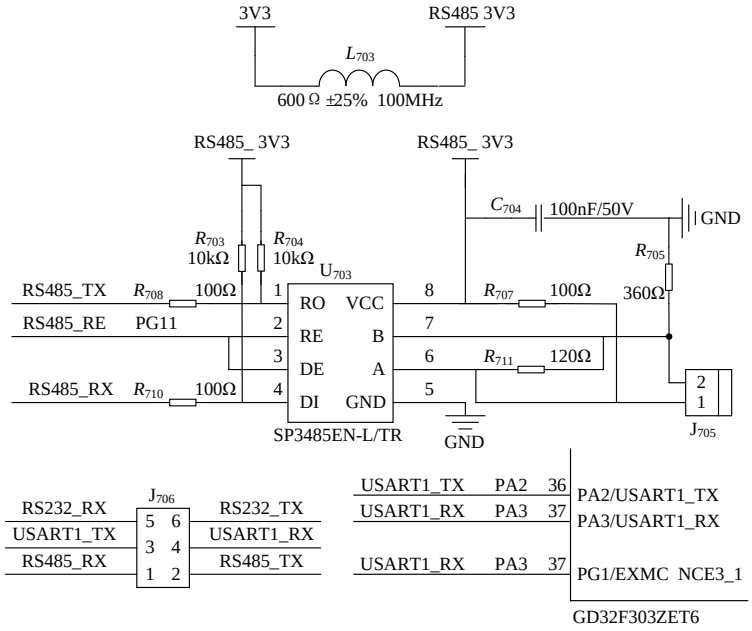

图 2-5　RS485 模块原理图

2.2.3　RS232 和 RS485 对比

上一章介绍的 RS232 是美国电子工业协会 EIA（Electronic Industry Association）制定的一种串行物理接口标准，由于其接口标准出现较早，难免有不足之处。针对 RS232 接口的不足，不断涌现了一些新的接口标准，RS485 便是其中之一。RS232 和 RS485 通信标准的比较如表 2-1 所示，下面简要介绍 RS232 和 RS485 通信标准的不同之处。

表 2-1　RS232 和 RS485 通信标准的比较

	RS232	RS485
工作方式	全双工	半双工
传输方式	单端通信	差分传输
电平特性	+12v~-12v	+5v~-5v
传输最大距离	50英尺	4000英尺
物理接口	DB9或DB25	无具体的物理接口形式
最大传输速率	20kB/s	35Mbps

（1）工作方式不同。RS232 采用全双工工作方式，而 RS485 采用半双工工作方式，任何时候只能有一点处于发送状态，因此，发送电路需由使能信号加以控制。RS232 只允许一对一通信，而 RS485 接口在总线上允许连接多达 128 个收发器，因此，RS485 用于多点互连时优势更为显著，可以减少使用很多信号线。

（2）传输方式不同。 RS232 采取不平衡传输方式，即单端通信，收、发端的数据信号是相对于信号地的。RS485 采用平衡传输方式，即差分传输方式，抑制共模干扰能力较强，能有

效提高抗干扰能力。

（3）电平特性不同。RS232 传输电平信号接口的信号电平值较高，易损坏接口电路的芯片。而 RS485 的接口信号电平较低，不易损坏接口电路的芯片，且该电平与 TTL 电平兼容，可方便与 TTL 电路连接。

综上所述，RS485 的综合特性强于 RS232，因此，其应用也更为广泛，RS485 是工业上运用最广泛的通信标准之一。RS232 更多与 USB 联合使用，随着 USB 端口的普及，将会出现更多将 USB 转换成 RS232 或其它接口的转换装置，通过 USB 接口可连接更多的 RS232 设备，不仅可获得更高的传输速度，实现真正的即插即用，同时解决了 USB 接口不能远距离传输的缺点（USB 通信距离在 5 米内）。

2.2.4 部分寄存器和固件库函数

RS485 使用的串行通信协议同样与 USART 相同，在本章实验中，串口数据接收和数据发送流程请参考 1.2.4 节。此外，本实验涉及的 USART 寄存器和固件库函数与 RS232 实验完全一致，具体请参考 1.2.5 节。

2.3 实验代码解析

2.3.1 UART1 文件对

由于在串口通信一章中已经详细介绍了这两个文件，这里不再赘述，具体的代码解释可参考《GD32F3 开发基础教程——基于 GD32F303ZET6》的串口通信实验。

2.3.2 RS485Top 文件对

1．RS485Top.h 文件

在 RS485Top.h 文件的“API 函数声明”区，声明了 2 个 API 函数，如程序清单 2-1 所示。InitRS485Top 函数的主要功能是设置 GUI 界面的显示内容，RS485TopTask 函数主要用于实现接收与发送数据功能。

程序清单 2-1

```
void InitRS485Top(void);   //初始化 RS485 实验应用层模块
void RS485TopTask(void);   //RS485 实验应用层模块任务
```

2．RS485Top.c 文件

在 RS485Top.c 文件的“内部变量”区，声明了 s_structGUIDev 结构体，如程序清单 2-2 所示，s_structGUIDev 为 GUI 设备结构体。

程序清单 2-2

```
static StructGUIDev s_structGUIDev;                //GUI 设备结构体
```

在 RS485Top. c 文件的“内部函数声明”区，声明了 SendProc 函数和 ReadProc 函数，如程序清单 2-3 所示，SendProc 函数在 GUI 发送回调函数中被调用，实现单片机向计算机发送数据，ReadProc 函数在 RS485TopTask 函数中被调用，实现单片机从计算机接收数据。

程序清单 2-3

```
static void SendProc(char* string); //发送处理函数
static void ReadProc(void);              //读串口处理
```

在 RS485Top.c 文件的“内部函数实现”区，首先实现了 SendProc 函数，如程序清单 2-4 所示。

（1）第 6 至 10 行代码：判断输入地址是否为空，为空则无数据需要发送，退出函数；否则继续发送。

（2）第 13 行代码：在每一次发送前清零计数值。

（3）第 14 至 24 行代码：判断（string+i）地址中存放的数据是否为零，不为零则将该数据赋值给 sendData，再通过串口 UART1 的 WriteUART1 写串口函数发送出去，最后计数 i 加一，循环此操作直到 string + i 地址中存放的数据为 0。

（4）第 27 至 30 行代码：在一次发送数据完成后发送回车换行结束，将'\r'回车符与'\n'换行符接连赋值给 sendData，再通过串口 UART1 的写串口函数 WriteUART1 分别发送出去。

程序清单 2-4

```
1.  static void SendProc(char* string)
2.  {
3.      u8 i;              //发送计数
4.      u8 sendData; //发送数据
5.
6.      //校验输入地址是否为空指针
7.      if(NULL == string)
8.      {
9.          return;
10.     }
11.
12.     //通过 UART1 将数据发送出去
13.     i = 0;
14.     while(0 != *(string + i))
15.     {
16.         //获取需要发送的数据
17.         sendData = *(string + i);
18.
19.         //通过串口发送出去
20.         WriteUART1(&sendData, 1);
```

```
21.
22.         //发送计数加一
23.         i++;
24.     }
25.
26.     //发送回车换行
27.     sendData = '\r';
28.     WriteUART1(&sendData, 1);
29.     sendData = '\n';
30.     WriteUART1(&sendData, 1);
31. }
```

在 SendProc 函数之后为 ReadProc 函数的实现代码，如程序清单 2-5 所示。

（1）第 3 行至 5 行代码：首先定义 s_arrReadBuf 数组作为接收缓冲区，s_iReadCnt 变量用于计数，以及 s_iBeginTime 用于接收到第一个字符时的系统时间，通过 time-s_iBeginTime 记录接收时间。

（2）第 11 行代码：通过 GetSysTime 函数获取系统运行时间。

（3）第 14 至 26 行代码：通过 ReadUART1 函数读取串口数据并储存在 readData 中，当接收到第一个字符时将 time 的值赋给 s_iBeginTime，记录此时系统时间，然后将数据储存到 s_arrReadBuf 缓存区，最后对计数 s_iReadCnt 进行加一操作。

（4）第 29 至 42 行代码：如果接收数据大于 1 位且接收到回车换行符，则向缓冲区加入字符串结尾，对接收计数清零以及通过 s_structDev.showLine 结构体成员输出数据到终端显示。

（5）第 46 至 56 行代码：为防止接收错误以及接收数据过长，如果超过 250 毫秒还没接收到回车换行则强制更新到终端显示，在输出到终端显示前，需要加上字符串结尾以及清零接收计数值。

程序清单 2-5

```
1.  static void ReadProc(void)
2.  {
3.      static u8    s_arrReadBuf[48] = {0}; //接收缓冲区
4.      static u8    s_iReadCnt             = 0;     //接收计数
5.      static u64 s_iBeginTime         = 0;     //接收到第一个字符时的系统时间
6.
7.      u8    readData;
8.      u64 time;
9.
10.     //获取系统运行时间
11.     time = GetSysTime();
12.
13.     //接收数据处理
14.     while(ReadUART1(&readData, 1))
15.     {
16.         //记录接收到第一个字符时的系统时间
17.         if(0 == s_iReadCnt)
```

```
18.         {
19.             s_iBeginTime = time;
20.         }
21.
22.         //将数据储存到缓冲区
23.         s_arrReadBuf[s_iReadCnt] = readData;
24.
25.         //接收计数加一
26.         s_iReadCnt = (s_iReadCnt + 1) % (sizeof(s_arrReadBuf) / sizeof(u8));
27.
28.         //接收到回车换行
29.         if(s_iReadCnt >= 2)
30.         {
31.             if(('\r' == s_arrReadBuf[s_iReadCnt - 2]) && ('\n' == s_arrReadBuf[s_iReadCnt - 1]))
32.             {
33.                 //加上字符串结尾
34.                 s_arrReadBuf[s_iReadCnt] = 0;
35.
36.                 //接收计数清零
37.                 s_iReadCnt = 0;
38.
39.                 //输出到终端显示
40.                 s_structGUIDev.showLine((char*)s_arrReadBuf);
41.             }
42.         }
43.     }
44.
45.     //超过 250 毫秒还没接收到回车换行则强制更新到终端显示
46.     if((s_iReadCnt > 0) && ((time - s_iBeginTime) > 250))
47.     {
48.         //加上字符串结尾
49.         s_arrReadBuf[s_iReadCnt] = 0;
50.
51.         //接收计数清零
52.         s_iReadCnt = 0;
53.
54.         //输出到终端显示
55.         s_structGUIDev.showLine((char*)s_arrReadBuf);
56.     }
57. }
```

在 RS485Top.c 文件的“API 函数实现”区，实现了 InitRS485Top 和 RS485TopTask 两个 API 函数，如程序清单 2-6 所示。

（1）第 1 至 14 行代码：InitRS485Top 函数用于对 s_structGUIDev 结构体成员进行赋值操作，实现串口号显示、波特率显示以及通过发送回调函数 SendProc 发送数据到串口，最后通过调用 InitGUI 函数进行初始化 GUI 界面设计从而实现在 LCD 上的显示。

（2）第 16 至 20 行代码：RS485TopTask 函数调用 ReadProc 函数接收串口数据并输出到终端，调用 GUITask 函数执行 GUI 任务。

程序清单 2-6

```
1.  void InitRS485Top(void)
2.  {
3.      //串口号显示
4.      s_structGUIDev.serialPort = "UART1";
5.
6.      //波特率显示
7.      s_structGUIDev.baudRate = "115200";
8.
9.      //发送回调函数
10.     s_structGUIDev.sendCallBack = SendProc;
11.
12.     //初始化 UI 界面设计
13.     InitGUI(&s_structGUIDev);
14. }
15.
16. void RS485TopTask(void)
17. {
18.     ReadProc(); //接收串口数据并输出到终端
19.     GUITask();   //GUI 任务
20. }
```

2.3.3 Main.c 文件

在 Main.c 文件的 Proc2msTask 函数中，每 40ms 调用一次 RS485TopTask 函数，实现 RS485 的接收与发送数据功能，如程序清单 2-7 所示。

程序清单 2-7

```
1.  static   void   Proc2msTask(void)
2.  {
3.      static u8 s_iCnt = 0;
4.      if(Get2msFlag())   //判断 2ms 标志状态
5.      {
6.          LEDFlicker(250);//调用闪烁函数
7.
8.          s_iCnt++;
9.          if(s_iCnt >= 20)
10.         {
11.             s_iCnt = 0;
12.             RS485TopTask();
13.         }
14.
15.         Clr2msFlag();     //清除 2ms 标志
16.     }
17. }
```

2.3.4 实验结果

代码编写完成并编译通过后，下载程序并进行复位。下载完成后，通过跳线帽将开发板上 J_{706} 的 485_RX（RS485_RX）与 PA2（USART1_TX）相连，将 485_TX（RS485_TX）与 PA3

（USART1_RX）相连，然后通过杜邦线将开放板上 J_{705} 连接座的 485A 接口连接到 RS485 串口转 USB 模块的 A 接口，将 485B 接口连接到 B 接口，再将 RS485 串口转 USB 模块的 USB 端连接到计算机。连接好后通过设备管理器查看对应的串口号，并在串口助手中打开该串口，最后进行数据收发：①计算机通过串口助手向开发板发送“1”，开发板接收到数据后，在 LCD 屏上的终端显示“1”；②在 LCD 的 GUI 界面上输入并发送“2”，计算机的串口助手上显示收到数据“2”，如图 2-6 和图 2-7 所示。

图 2-6　计算机端接收和发送数据

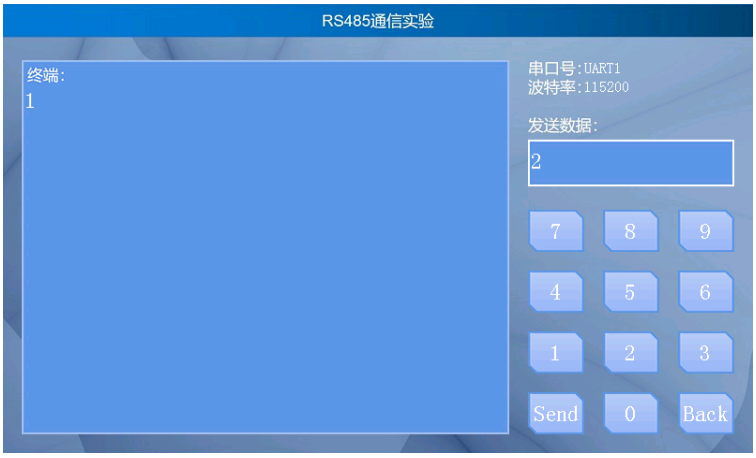

图 2-7　开发板端接收和发送数据

本章任务

在本章实验中，通过 RS485 串口转 USB 模块实现了 GD32F3 苹果派开发板与计算机之间的 RS485 通信，尝试通过杜邦线直接连接两块开发板的 J_{705} 接口，通过 LCD 上的 UI 收发数据，实现两块开发板之间的 RS485 通信。具体要求为：开发板 A 发送数据“1”，开发板 B 的 LCD 屏上终端显示“1”；开发板 B 发送数据“2”，开发板 A 的 LCD 屏上终端显示“2”。

本章习题

1．RS485 的信号传输方式是什么？简要概括其特点。

2．RS485 通信线路上常常需要添加电阻，其作用是什么？

3．简述 RS485 和 RS232 的区别，并根据其特性列举不同的工作场景。

3 呼吸灯实验

PWM（Pulse Width Modulation）——脉冲宽度调制技术，利用高分辨率计数器对一系列脉冲的宽度进行调制，以此获得所需要的波形，从而实现了对模拟信号电平进行数字编码。这项技术被广泛地应用在测量、通信、电源设计、功率控制与变换等诸多领域中，在本章实验中，将通过 PWM 来实现呼吸灯功能。

3.1 实验内容

本章的主要内容是掌握 PWM 相关参数的概念以及 PWM 的原理与模式，了解呼吸灯实现的原理，最后基于 GD32F3 苹果派开发板设计一个呼吸灯实验，通过逐渐改变 PWM 的占空比来实现 LED 灯呼吸。本实验使用 y=e^(-x^2)函数来配置 PWM 的占空比，使呼吸灯的呼吸效果更加自然，同时还可以通过滑动 LCD 触摸屏上的滑条来改变 PWM 的频率从而改变呼吸灯的呼吸频率。

3.2 实验原理

3.2.1 PWM 相关参数概念介绍

PWM 的频率：1s 内信号从高电平到低电平再回到高电平的次数（一个周期）。

PWM 的周期：PWM 频率的倒数。

PWM 的占空比：在一个脉冲周期内，高电平占的时间与整个周期的比值。例如：若一个脉冲周期的时间为 10ms，其中高电平时间为 8ms，低电平时间为 2ms，则占空比为 8/10 = 80%。

如图 3-1 所示，假设 PWM 的周期为 0.5s，则其频率为 2HZ，高电平的时间占据整个周期的 50%，及高电平持续的时间为 0.25s，占空比为 50%。

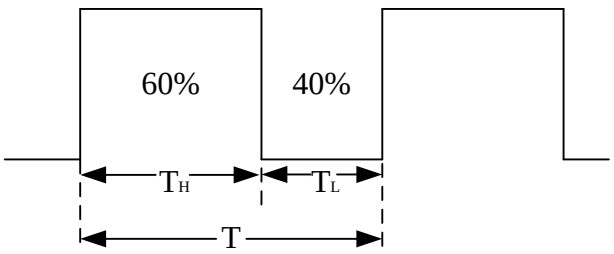

图 3-1 占空比为 60%的 PWM 波

3.2.2 PWM 的原理

单片机的 IO 口输出的是数字信号，通常只能输出高电平和低电平。假设高电平为 3.3V，低电平则为 0V，如果想要单片机的 IO 口输出不同的模拟电压，就需要使用 PWM，通过改变 IO 口输出的方波的占空比，从而获得使用数字信号模拟得到的模拟电压信号。

电压是以一种连接或断开的重复脉冲序列被加到模拟负载上去的（例如 LED 灯、直流电机等），连接即是直流供电输出，断开即是直流供电断开。通过控制连接和断开的时间，理论上可以输出任意不大于最大电压值（即 0~3.3V 之间任意大小）的模拟电压。

PWM 的调节作用来源于对占空比的大小控制，占空比变大，输出的能量就会增加，通过阻容变换电路所得到的平均电压值也会上升，占空比变小，输出的能量就会减少，通过阻容变换电路所得到的平均电压值也会下降。

若占空比为 50%，此时高电平时间与低电平时间相同，均为周期的 1/2，在一定的频率下，就可以得到模拟的 1.65V 输出电压 ，如果是 75%的占空比，就能得到 2.475V 的电压。

3.2.3 PWM 的输出模式

GD32F30x 系列微控制器的定时器分为三类，分别是基本定时器、通用定时器（L0、L1 和 L2）和高级定时器，除了基本定时器，其他的定时器都可以用来产生 PWM 输出，其中高级定时器和通用定时器 L0 均可同时产生多达 4 路的 PWM 输出，而通用定时器 L1 和通用定时器 L2 也分别能同时产生 2 路和 1 路的 PWM 输出，因此，GD32F30x 系列微控制器最多就可以同时产生 32 路 PWM 输出。

高级定时器拥有四个独立的通道用于捕获输入或比较输出是否匹配。每个通道都围绕一个通道捕获比较寄存器建立，包括一个输入级，通道控制器和输出级。输出比较有 8 种模式，分别是时基、匹配时设置为高、匹配时设置为低、匹配时翻转、强制为低、强制为高、PWM 模式 0 和 PWM 模式 1，通过 TIMERx_CHCTL0 的 CHxCOMCTL[2:0]选择输出比较模式。下面仅介绍 PWM 模式 0 和 PWM 模式 1 这两种输出模式。

当输出比较配置为 PWM 输出模式（模式 0 或模式 1）时，通道根据计数器自动重载寄存器 TIMERx_CAR 和通道 x 捕获/比较寄存器 TIMERx_CHxCV 的值输出 PWM 波形。此外，根据计数模式，可以分为两种 PWM 波：EAPWM（边沿对齐 PWM）和 CAPWM（中央对齐 PWM）。本次实验中采用 EAPWM 模式下的 PWM 模式 0。

如图 3-2，当输出比较配置为 PWM 模式 0，在向上计数时，如果计数器值小于 TIMERx_CHxCV，输出的参考信号 Cx OUT 为有效电平，否则为无效电平；在向下计数时，如果计数器的值大于 TIMERx_CHxCV，输出的参考信号 Cx OUT 为无效电平，否则为有效电

平。当输出比较配置为 PWM 模式 1，在向上计数时，如果计数器值小于 TIMERx_CHxCV，输出的参考信号 Cx OUT 为无效电平，否则为有效电平；在向下计数时，如果计数器值大于 TIMERx_CHxCV，输出的参考信号 Cx OUT 为有效电平，否则为无效电平。当 TIMERx_CHCTL2 的 CHxP 为 0 时，通道 x 高电平有效；当 CHxP 为 1 时，通道 x 低电平有效。

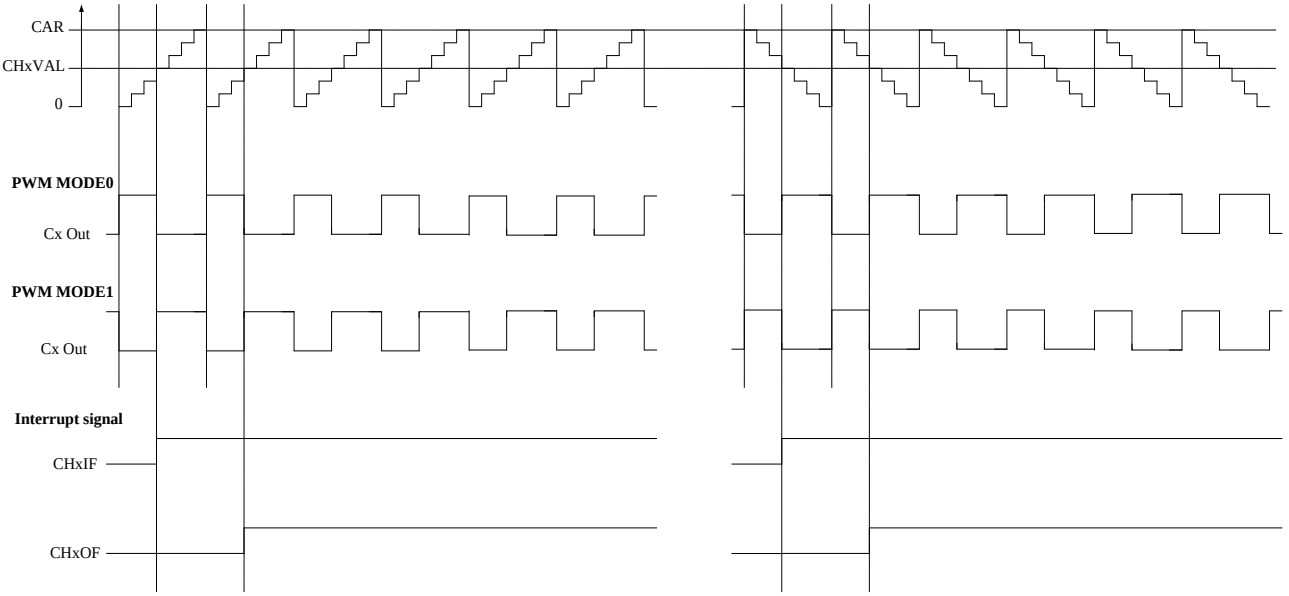

图 3-2　EAPWM 时序图

在 PWM0 模式下，PWM 的占空比的计算方式为：

PWM 的占空比 = TIMERx_CHxCV 寄存器的值 / TIMERx_CAR 寄存器的值 × 100%

假如 TIMERx 时钟的输入频率为 100MHz，则 PWM 的频率为：

f_{pwm} = 100M / ((TIMERx_CAR 寄存器的值 + 1) × (预分频寄存器 TIMERx_PSC 的值 +1))Hz

3.2.4　呼吸灯原理

如图 3-3 所示为本次呼吸灯实验相关硬件原理图，其中绿色发光二极管 LED_1 连接到 GD32F303ZET6 微控制器的 PA8 引脚，而蓝色发光二极管 LED_2 连接到 PE6 引脚，电阻 R_{110} 与 R_{115} 的作用是限流分压。在本章实验中，将 PA8 引脚复用为 TIMER0_CH0，并将其配置为 PWM 输出模式，从而实现 LED_1 的呼吸灯功能。

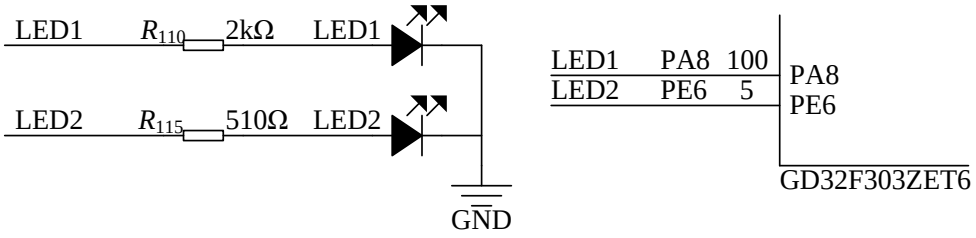

图 3-3　呼吸灯相关硬件原理图

呼吸灯的现象即为周期性提升和减弱 LED 灯的亮度，近似于模拟人体呼吸。人眼的图像滞留时间约为 40ms，理论上如果以 20ms 为一个周期，LED 一直以亮 10ms、灭 10ms 的状态执行，人眼看到的情况应该是 LED 持续点亮。如果每进行一个周期后适当延长 LED 的点亮时间，减短 LED 的熄灭时间，则 LED 的亮度会不断提高；而如果每进行一个周期后适当减短 LED 的点亮时间，延长 LED 的熄灭时间，则 LED 的亮度在逐渐减小。如果不断提高 LED 的亮度，并在亮度达到一定程度时，开始逐渐减小 LED 的亮度，当亮度减弱到一定程度时，再增强 LED 的亮度，如此循环，便可以让 LED 实现呼吸灯的效果。在本章实验中，通过改变 PA8 引脚输出的 PWM 的占空比来控制 LED_1 的点亮时间。

3.2.5　实验流程图

如图 3-4 所示是本章实验的流程图。首先，将 PA8 引脚复用为 TIMER0_CH0，然后将 TIMER0_CH0 配置为 PWM 模式 0，计数模式配置为递增计数。其次，向 TIMER0 对应的计数器自动重载寄存器 TIMER0_CAR 中写入 499，向预分频寄存器 TIMER0_PSC 中写入 199，由于 TIMER0 的 CNT 计数器对经过分频后的 PSC_CLK 时钟进行计数。因此，TIMER0 的 CNT 计数器向上递增计数的范围为 0 到 499。

本实验是将 TIMER0_CH0 配置为 PWM 模式 0，将比较输出设置为高电平有效且设置为向上计数模式，因此，一旦 TIMER0_CNT<TIMER0_CH0CV 时，比较输出引脚（即 PA8）为有效电平，否则为无效电平。

当开发板上的触摸屏的滑条位置发生改变时，将会改变参数 s_structSlider.value 的值，进而改变呼吸灯的周期 period，使得呼吸灯的频率发生变化。注意，period 的最小值为 50（ms），

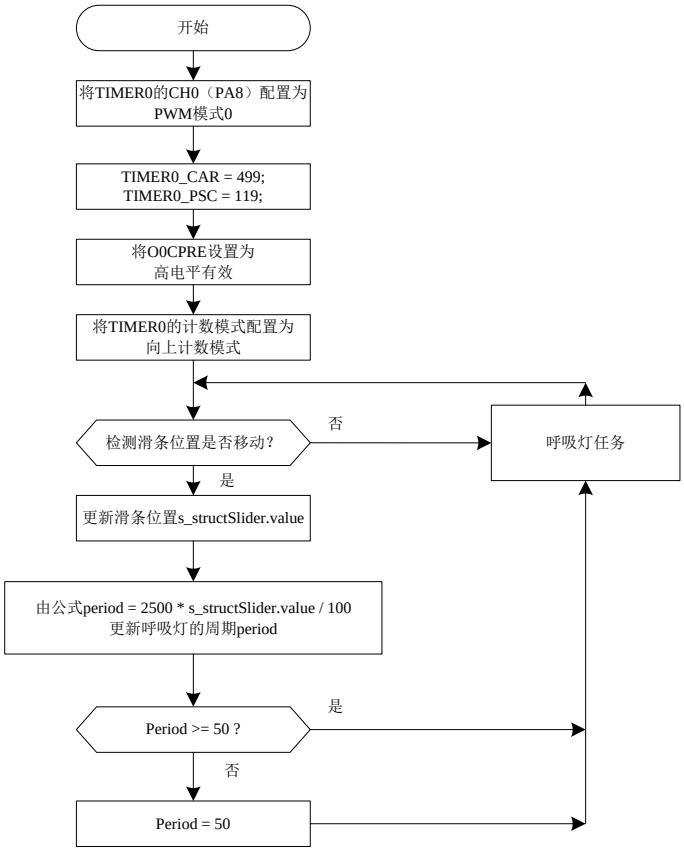

图 3-4　呼吸灯实验流程图

3.3　实验代码解析

3.3.1　BreathLED 文件对

1．BreathLED.h 文件

在 BreathLED.h 文件的“API 函数声明”区，声明了 2 个 API 函数，如程序清单 3-1 所示。InitBreathLED 函数的主要功能是初始化呼吸灯模块，BreathLEDTask 函数实现的是开发板上的呼吸灯任务。

程序清单 3-1

```
void InitBreathLED(void);            //初始化呼吸灯模块
void BreathLEDTask(u32 period); //呼吸灯任务
```

2．BreathLED.c 文件

在 BreathLED.c 文件的“包含头文件”区，包含了 gd32f30x_conf.h 和 BreathLED.h 头文件。因为 BreathLEDTask 函数需要调用 math.h 里的 exp 函数和 Timer.h 里的 GetSysTime 函数来获取系统时间，因此，这里还包含了 math.h 和 Timer.h，如程序清单 3-2 所示。

程序清单 3-2

```
1.  #include "BreathLED.h"
2.  #include "gd32f30x_conf.h"
3.  #include "math.h"
4.  #include "Timer.h"
```

在 BreathLEDTop.c 文件的“内部函数声明”区，声明了两个内部函数，ConfigBreathLEDGPIO 函数用于配置呼吸灯的 GPIO，即 PA8，ConfigBreathLedPWM 函数用于配置呼吸灯的 PWM，如程序清单 3-3 所示。

程序清单 3-3

```
static void ConfigBreathLEDGPIO(void); //配置呼吸灯的 GPIO
static void ConfigBreathLedPWM(void);   //配置呼吸灯的 PWM
```

在 BreathLEDTop.c 文件的“内部函数实现”区，首先实现 ConfigBreathLEDGPIO 函数，如程序清单 3-4 所示。

（1）第 3 至 7 行代码：本次实验使用的 LED_1 与引脚 PA8 相连接，将 PA8 复用为 TIMER0_CH0 并将其配置为 PWM 输出，因此需要通过 rcu_periph_clock_enable 函数使能 GPIOA 时钟、TIMER0 时钟和复用时钟。通过 gpio_init 函数将引脚 PA8 设置为复用功能 TIMER0_CH0，且为推挽输出模式，频率为 50MHz。

程序清单 3-4

```
1.  static void ConfigBreathLEDGPIO(void)
2.  {
3.      //使能 RCU 相关时钟
4.      rcu_periph_clock_enable(RCU_TIMER0); //使能 TIM0 时钟
5.      rcu_periph_clock_enable(RCU_GPIOA);   //使能 GPIOA
6.      rcu_periph_clock_enable(RCU_AF);        //使能复用时钟
7.      gpio_init(GPIOA, GPIO_MODE_AF_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_8);   //PA8 复用推挽
8.  }
```

在 BreathLEDTop.c 文件的“内部函数实现”区，在 ConfigBreathLEDGPIO 函数之后为 ConfigBreathLedPWM 函数的实现代码，如程序清单 3-5 所示。

（1）第 3 至 16 行代码：通过 timer_init 函数配置 TIMER0，该函数涉及 TIMER0_PSC、TIMER0_CAR、TIMER0_CTL0 的 CKDIV[1:0]，以及 TIMER0_SWEVG 的 UPG。CKDIV[1:0] 用于设置时钟分频系数。结构体变量 timer_initpara.prescaler 和 timer_initpara.period 用于设置计数器的预分频器和自动重装载值的值，在本实验中，通过 ConfigBreathLedPWM 函数将两者分别设置为 119 和 499。UPG 用于产生更新事件，本实验中将该值设置为 1，用于重新初始化计数器，并产生一个更新事件。

（2）第 25 行代码：通过 timer_channel_output_config 函数初始化 TIMER0 的 CH0，该函

数涉及 TIMER0_CHCTL0 的 CH0P 和 CH0EN，CH0P 用于设置通道输出极性，CH0EN 用于使能或禁止捕获/比较。在本实验中，将通道 0 设置为高电平有效。

（3）第 27 行代码：通过 timer_channel_output_pulse_value_config 数初始化占空比。

（4）第 29 行代码：通过 timer_channel_output_mode_config 函数设置 TIMER0 通道 0 的输出比较模式为 PWM 模式 0。

（5）第 31 行代码：通过 timer_channel_output_shadow_config 函数设置禁用通道影子寄存器。

（6）第 35 行代码：通过 timer_enable 函数使能 TIMER0。

程序清单 3-5

```
1.  static void ConfigBreathLedPWM(void)
2.  {
3.      timer_oc_parameter_struct timer_ocintpara;
4.      timer_parameter_struct timer_initpara;
5.      //使能 RCU 相关时钟
6.      rcu_periph_clock_enable(RCU_TIMER0); //使能 TIM0 时钟
7.      //复位 TIM0
8.      timer_deinit(TIMER0);
9.      //配置 TIM0
10.     timer_initpara.prescaler         = 119;                    //设置预分频值
11.     timer_initpara.alignedmode       = TIMER_COUNTER_EDGE; //设置边沿对齐模式
12.     timer_initpara.counterdirection  = TIMER_COUNTER_UP;   //设置向上计数模式
13.     timer_initpara.period            = 499;                    //设置自动重装载值
14.     timer_initpara.clockdivision     = TIMER_CKDIV_DIV1;   //设置时钟分割
15.     timer_initpara.repetitioncounter = 0;                      //设置重复计数器
16.     timer_init(TIMER0, &timer_initpara);                       //根据参数初始化定时器
17.
18.     //配置 TIM0 CH0
19.     timer_ocintpara.outputstate  = TIMER_CCX_ENABLE;                    //PWM 输出使能
20.     timer_ocintpara.outputnstate = TIMER_CCXN_DISABLE;                  //禁止比较输出
21.     timer_ocintpara.ocpolarity   = TIMER_OC_POLARITY_HIGH;              //输出极性位高
22.     timer_ocintpara.ocnpolarity  = TIMER_OCN_POLARITY_HIGH;             //比较输出极性为高
23.     timer_ocintpara.ocidlestate  = TIMER_OC_IDLE_STATE_LOW;             //空闲状态通道输出极性
24.     timer_ocintpara.ocnidlestate = TIMER_OCN_IDLE_STATE_LOW;            //空闲状态比较通道输出极性
25.     timer_channel_output_config(TIMER0, TIMER_CH_0, &timer_ocintpara); //根据参数配置定时器通道
26.     //设定 PWM 值
27.     timer_channel_output_pulse_value_config(TIMER0, TIMER_CH_0, 0);
28.     //PWM0 模式
29.     timer_channel_output_mode_config(TIMER0, TIMER_CH_0, TIMER_OC_MODE_PWM0);
30.     //禁用通道影子寄存器
31.     timer_channel_output_shadow_config(TIMER0, TIMER_CH_0, TIMER_OC_SHADOW_DISABLE);
32.     //PWM 输出使能
33.     timer_primary_output_config(TIMER0, ENABLE);
34.     //使能 TIM0
35.     timer_enable(TIMER0);
36. }
```

在 BreathLED.c 文件的“API 函数实现”区，首先实现了 InitBreathLED 函数，通过调用 ConfigBreathLEDGPIO 和 ConfigBreathLedPWM 函数分别配置呼吸灯的 GPIO 和 PWM，如程序清单 3-6 所示。

程序清单 3-6

```
1.  void InitBreathLED(void)
2.  {
3.      ConfigBreathLEDGPIO(); //配置呼吸灯的 GPIO
4.      ConfigBreathLedPWM();  //配置呼吸灯的 PWM
5.  }
```

在 BreathLED.c 文件的“API 函数实现”区，在 InitBreathLED 函数之后为 BreathLEDTask 函数的实现代码，如程序清单 3-7 所示。

（1）第 11 至 16 行代码：通过函数标志位 s_iFirstFlag 来判断是否第一次调用 BreathLEDTask 函数，若第一次调用 BreathLEDTask 函数，则调用 GetSysTime 函数来获得呼吸灯的第一个 PWM 周期的起始时间 s_iLastCycleBeginTime，同时跳出 BreathLEDTask 函数。

（2）第 17 至 18 行代码：再次进入 BreathLEDTask 函数时调用 GetSysTime 来获得此时系统的时间 currenTime，通过此时系统的时间 currenTime 与呼吸灯的第一个 PWM 周期的起始时间 s_iLastCycleBeginTime 相减可获得距离第一个 PWM 周期开始所流经的时间 elapsedTime。

（3）第 19 至 22 行代码：通过调用 TIMER_CAR 函数获得计时器 TIMER0 的自动重装载值，与函数 y=e^(-x^2) 所得到 PWM 值的比例相乘便得到了 PWM 值，再调用 timer_channel_output_pulse_value_config 函数来配置 PWM 的输出。

（4）第 24 至 27 行代码：当一个呼吸灯的周期结束后，呼吸灯的第一个 PWM 周期的起始时间 s_iLastCycleBeginTime 的数值更新为调用此次函数时的系统时间。

程序清单 3-7

```
1.  void BreathLEDTask(u32 period)
2.  {
3.      static u64 s_iLastCycleBeginTime = 0; //上一次循环开始时间
4.      static u64 s_iFirstFlag = 1;                //第一次执行此函数标志位
5.      u64       currenTime;   //现在时间
6.      u64       elapsedTime; //一个循环中流经的时间
7.      double x;                  //弧度
8.      double rate;               //比例
9.      u32       pwm;              //PWM 值
10.     //初始化
11.     if(1 == s_iFirstFlag)
12.     {
13.         s_iFirstFlag = 0;
14.         s_iLastCycleBeginTime = GetSysTime();
15.         return;
```

```
16.      }
17.      currenTime = GetSysTime();//获取系统时间
18.      elapsedTime = currenTime - s_iLastCycleBeginTime; //计算一个周期内流经的时间
19.      x = -2.0 + 4.0 * elapsedTime / period;//计算 x 值，（period 为 50 至 2500）
20.      rate = exp(- x * x); //计算 PWM 值比例
21.      pwm = TIMER_CAR(TIMER0) * rate;//500*rate= PWM 数值
22.      timer_channel_output_pulse_value_config(TIMER0, TIMER_CH_0, pwm); //配置 PWM 输出
23.      //一个周期结束
24.      if(elapsedTime >= period)
25.      {
26.         s_iLastCycleBeginTime = currenTime;
27.      }
28.   }
```

3.3.2 BreathLEDTop 文件对

1．BreathLEDTop.h 文件

在 BreathLEDTop.h 文件的“API 函数声明”区，声明了两个 API 函数，如程序清单 3-8 所示。InitBreathLEDTop 函数的主要功能是初始化呼吸灯实验顶层模块，BreathLEDTopTask 函数实现的是开发板上的呼吸灯实验任务。

程序清单 3-8

```
void InitBreathLEDTop(void);   //初始化呼吸灯实验顶层模块
void BreathLEDTopTask(void);   //呼吸灯实验任务
```

2．BreathLEDTop.c 文件

在 BreathLEDTop.c 文件的“包含头文件”区，包含了 gd32f30x_conf.h，而 gd32f30x_conf.h 中包含了 GD32F30x 系列微控制器的各种固件库头文件。

在 BreathLEDTop.c 文件的“内部函数声明”区，声明了内部函数 DisplayBackground，如程序清单 3-9 所示。该函数用于绘制 LCD 背景，但在调用这个函数之前要确保 LCD 显示方向为横屏显示。

程序清单 3-9

```
static void DisplayBackground(void);                                    //绘制背景
```

在 BreathLEDTop.c 文件的“内部函数实现”区，为 DisplayBackground 函数的实现代码，如程序清单 3-10 所示。

（1）第 4 至 7 行代码：定义背景图片控制结构体，并初始化结构体 backgroundImage。

（2）第 9 行代码：通过 DisplayJPEGInFlash 函数解码并显示背景图片。

程序清单 3-10

```
1.    static void DisplayBackground(void)
2.    {
3.       //背景图片控制结构体
```

```
4.      StructJpegImage backgroundImage;
5.      //初始化 backgroundImage
6.      backgroundImage.image = (unsigned char*)s_arrJpegBackgroundImage;
7.      backgroundImage.size  = sizeof(s_arrJpegBackgroundImage) / sizeof(unsigned char);
8.      //解码并显示图片
9.      DisplayJPEGInFlash(&backgroundImage, 0, 0);
10.  }
```

在 BreathLEDTop.c 文件的“API 函数实现”区，首先实现了 InitBreathLEDTop 函数，如程序清单 3-11 所示。

（1）第 4 行代码：调用 LCDDisplayDir 函数来设置 LCD 显示方向为横屏；

（2）第 5 行代码：调用 LCDClear 函数设置清屏的填充色为红色；

（3）第 7 行代码：调用 DisplayBackground 函数来绘制背景；

（4）第 8 至 20 行代码：初始化内部变量 s_arrSliderBackground，并调用 CreateSliderWidget 函数来创建滑条控件。

程序清单 3-11

```
1.   void InitBreathLEDTop(void)
2.   {
3.     //LCD 横屏显示
4.     LCDDisplayDir(1);
5.     LCDClear(GRED);
6.     //绘制背景
7.     DisplayBackground();
8.     //创建滑条控件
9.     s_structSlider.x0          = 100;                                    //起始横坐标
10.    s_structSlider.y0          = 300;                                    //起始纵坐标
11.    s_structSlider.width       = 600;                                    //宽度
12.    s_structSlider.height      = 30;                                     //高度
13.    s_structSlider.value       = 50;                                     //滑条预设置（0-100）
14.    s_structSlider.lineImage   = NULL;                                   //线条不使用刷图片方式
15.    s_structSlider.circleImage = (void*)s_arrBmpSliderCircleImage30x30x32; //滑条圆点图片
16.    s_structSlider.lineSize    = 3;                                      //横线宽度为 6
17.    s_structSlider.lineColor   = 0xFDCA;                                 //线条颜色
18.    s_structSlider.circleColor = 0xFDCA;                                 //圆点颜色
19.    s_structSlider.background  = (void*)s_arrSliderBackground;           //背景缓冲区
20.    CreateSliderWidget(&s_structSlider);                                 //创建滑条控件
21.  }
```

在 BreathLEDTop.c 文件的“API 函数实现”区，在 InitBreathLEDTop 函数之后为 BreathLEDTopTask 函数的实现代码，如程序清单 3-12 所示。

（1）第 3 行代码：定义参数 period 用于控制呼吸灯周期。

（2）第 4 至 11 行代码：调用 ScanSliderWidget 函数，扫描滑动条。改变滑条位置，可使 s_structSlider.value 的值发生变化，进而改变 period 的数值，使呼吸灯的周期发生变化。

程序清单 3-12

```
1.  void BreathLEDTopTask(void)
2.  {
3.      u32 period;
4.      ScanSliderWidget(&s_structSlider); //滑动条扫描
5.      //获取呼吸灯周期
6.      period = 2500 * s_structSlider.value / 100;// s_structSlider.value 的值为 0 至 100
7.      if(period < 50)
8.      {
9.          period = 50;
10.     }//period 为 50 至 2500
11.     BreathLEDTask(period); //呼吸灯任务
12. }
```

3.3.3 Main.c 文件

在 Main.c 文件的 Proc1msTask 函数中，每 20ms 调用一次 BreathLEDTopTask 函数，执行呼吸灯任务，如程序清单 3-13 所示。

程序清单 3-13

```
1.  static  void  Proc1msTask(void)
2.  {
3.      static u8 s_iCnt = 0;
4.      if(Get1msFlag())
5.      {
6.          s_iCnt++;
7.          if(s_iCnt >= 20)
8.          {
9.              ScanTouch();            //触摸屏扫描
10.             BreathLEDTopTask(); //呼吸灯任务
11.             s_iCnt = 0;
12.         }
13.     Clr1msFlag();
14.     }
15. }
```

3.3.4 实验结果

代码编写完成并编译通过后，下载程序并进行复位。下载完成后，可以观察到 LED_1 的呼吸灯效果，同时划动 LCD 屏幕上的滑条位置可以改变呼吸灯的“呼吸”频率，如图 3-5 所示，表示实验成功。

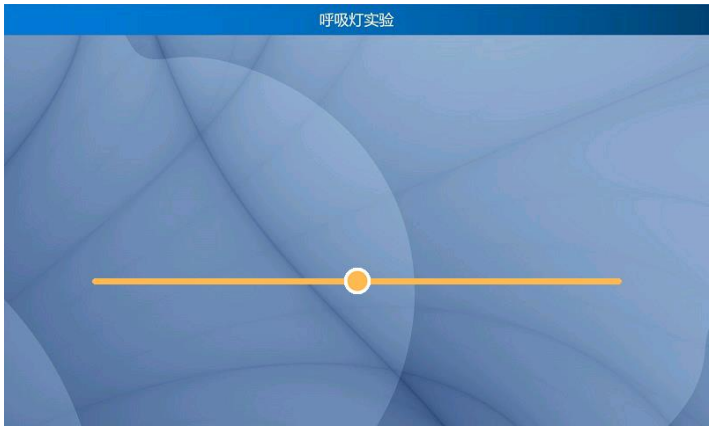

图 3-5　呼吸灯实验 GUI 界面

本章任务

在本章实验中，使用绿色发光二极管 LED_1 作为呼吸灯，且以将 PA8 配置为 PWM 输出的方式来实现。现尝试以蓝色发光二极管 LED_2 作为呼吸灯，并通过普通 GPIO 输出的方式模拟 PWM 来实现以 LED_2 作为呼吸灯，同时可以通过按下 KEY_1 提高呼吸灯的频率，按下 KEY_2 实现复位，按下 KEY_3 降低呼吸灯的频率。

本章习题

1．简要描述呼吸灯的实现思路。

2．根据本实验中的初始配置参数，计算初始状态下的 PWM 周期，并与示波器测量的周期进行对比。

3．能否将 LED_2 对应的引脚同样配置为 PWM 输出，使 LED_2 也实现呼吸灯效果。

4 电容触摸按键实验

人与机器进行交互通常是通过键盘、鼠标、按键等外接设备将信息传入机器中，机器根据输入的信息执行操作，从而达到与机器进行信息交互的目的。以手机为例，以前的手机大多通过实体按键进行交互，而如今，触摸屏已然成为手机不可或缺的部分，传统实体按键的使用场景逐渐变少。相对于传统的机械按键，触摸屏和触摸按键等有寿命长、占用空间少、易于操作等诸多优点。在 GD32F3 苹果派开发板上，集成了一个电容触摸按键，本章将围绕该触摸按键设计实验，捕获按键按下的时间。

4.1 实验内容

本章的主要内容是学习 RC 充放电基本原理和 GD32F3 苹果派开发板上的触摸按键模块电路原理图，掌握检测触摸按键按下的原理，最后基于开发板设计一个电容触摸按键实验。通过 GD32F303ZET6 微控制器的 PB1 引脚持续控制电容充放电，并获取电容充电时间，根据充电时间长短判断是否有手指触摸按键，若有则计算触摸时长并输出至 LCD 屏的终端中，同时输出松开按键的时间。

4.2 实验原理

4.2.1 电容充放电原理

电容一般指电容器。两个相互靠近的极板（导体），中间由一层不导电的绝缘介质隔开，即构成了电容。当为电容的两个极板之间加上电压时，电容将储存电荷。电容的电容量在数值上等于一个导电极板上的电荷量与两个极板之间的电压之比。电容的电容量的基本单位是法拉（F）。

在 *RC* 充电电路中，电容两端的电压与充电时间之间的关系满足以下公式：

$$\mathrm{Vt} = \mathrm{V_0} + (V_1 - \mathrm{V_0}) \times \left[1 - e^{-\frac{t}{RC}}\right]$$

V_0为电容两端的初始电压值，V_1 为电容最终可充到或放到的电压值，Vt为 t 时刻电容两端的电压值，R 为电路中与电容串联的有效电阻，C 为电容值。如果V_0为 0，即从 0V 开始充电，则公式可以简化为：

$$\mathrm{Vt} = V_1 \times \left[1 - e^{-\frac{t}{RC}}\right]$$

若电阻值固定，在同样的条件下，电容值 C 跟时间值 t 成正比，电容越大，充电到达某个临界值的时间越长。电容充放电时间与电容大小之间的关系如图 4-1 所示。

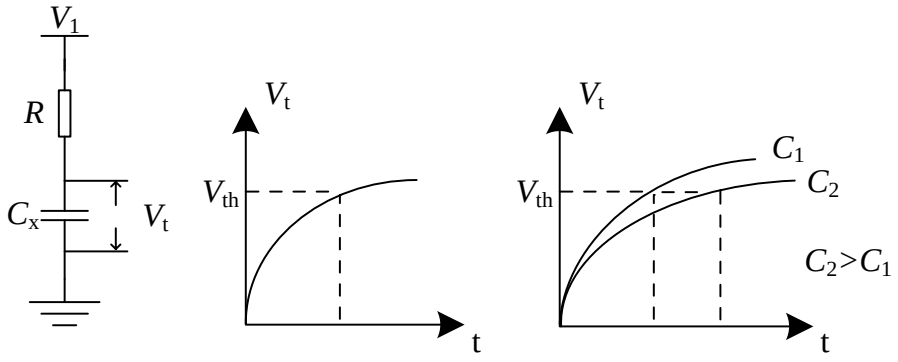

图 4-1　电容充电时间与电容大小之间的关系

4.2.2 电路介绍

GD32F3 苹果派开发板上的电容触摸按键电路的原理图如图 4-2 所示。电容触摸按键实验涉及的电路模块包含 1 个电容触摸按键（在开发板上是一块的覆铜区域）；1 个与电容触摸按键串联的 1MΩ 电阻，其作用是保护电容的同时放大 RC 值。人体有时会带有静电，而静电通常瞬时电压较高，若直接作用在电路中可能会造成元件损坏，因此电路中增加了一个静电放电保护二极管，防止静电损伤。电容触摸按键电路由 GD32F303ZET6 微控制器的 PB1 引脚控制，在本实验中，将 PB1 引脚复用为 TIMER_CH3，通过 TIMER2 的通道 3 捕获电容电压从而获取电容充电时间，根据电容充电时间长短即可判断是否有手指按下电容触摸按键。

图 4-2　电容触摸按键电路原理图

4.2.3 检测按键按下原理

在本章实验中，通过检测电容充放电时间来判断是否有手指按下电容触摸按键，原理如图 4-3 所示，图中 R 为外接的电容充电电阻，C_1 为手指未按下电容触摸按键时，电容触摸按键与地之间的杂散电容，C_2 为手指按下电容触摸按键时，手指与电容触摸按键之间形成的电容。

将 PB1 引脚拉低，使 C_1 完全放电，随后 C_1 开始充电，当手指未按下电容触摸按键时，电

容的充电曲线如图中 S_1 曲线所示；当手指按下电容触摸按键时，手指和电容触摸按键之间引入了新的电容，此时电容的充电曲线如图中的 S_2 曲线所示。在 S_1 和 S_2 两条充电曲线上，V_t 达到 V_1 的时间分别为 T_1 和 T_2。只要能够区分 T_1 和 T_2，即可实现手指按下检测。当充电时间在 T_1 附近，即可认为手指没有按下，当充电时间大于 T_2 时，即认为手指按下（T_2 为检测阀值）。

在本章实验中，通过 PB1（TIMER2_CH3）来实现检测电容触摸按键是否被按下。检测过程如下：①配置 PB1 为推挽输出并将引脚电平拉低，使电容 C_1 放电；②配置 PB1 为浮空输入，利用外部上拉电阻给电容 C_1 充电，同时开启 TIMER2_CH3 的输入捕获，检测引脚电平的上升沿，当检测到上升沿时，表示电容充电完成，完成一次捕获；③获取多次捕获结果后（以 25ms 为一个周期），取其中最大值并保存。若最大值与 T_1 接近，则表明此时间段（上述 25ms）内手指未按下电容触摸按键；若最大值与 T_2 接近，则表明此时间段内手指按下了电容触摸按键。

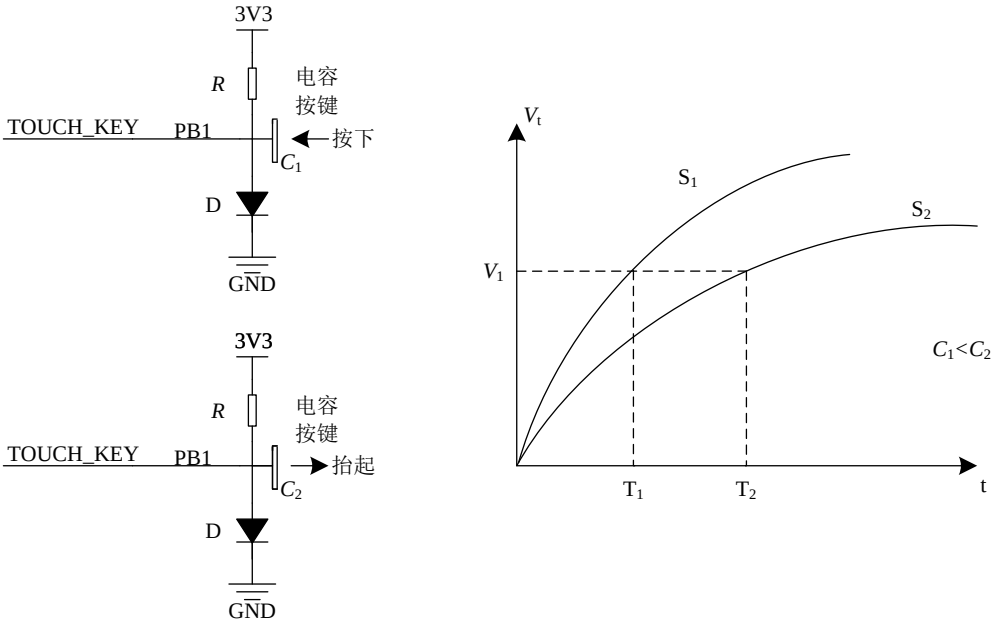

图 4-3　检测按键按下原理

4.2.4　实验流程图

电容触摸按键实验的流程图如图 4-4 所示。首先，初始化电容触摸按键实验顶层模块，初始化 GUI 界面。其次，通过 ConfigTouchKey 函数使能 TIMER2，配置 TIMER2 的 CH3 通道为上升沿捕获。①进入 GUI 界面后，当写有“电容按键”的覆铜区域被按下时，程序将调用函数 TIMER2_IRQHandler。该中断函数每隔 3ms 获取一次电容触摸按键输入捕获结果，以 25ms 为一个周期，在这个周期内取输入捕获值最大值，并保存最大值。②系统将调用 ScanTouchKey 函数，该函数首先获取当前系统时钟，而后判断此时系统的状态。若输入捕获值大于电容按键

空置时的值，则判断此时的输入捕获值为按键按下的值，若输入捕获值小于电容按键空置时的值，则判断此时的输入捕获值为按键抬起的值。③记录系统此刻按下或是抬起的时钟后：1.若电容按键判断为按下，按下时刻系统时钟为 0，则返回 0，否则返回当前系统时刻与按键按下时刻的差值。2.若电容按键判断为抬起，抬起时刻系统时钟为 0，则返回 0，否则返回当前系统时刻与按键抬起时刻的差值。④将按键按下或抬起的时间差值在 GUI 界面中显示。

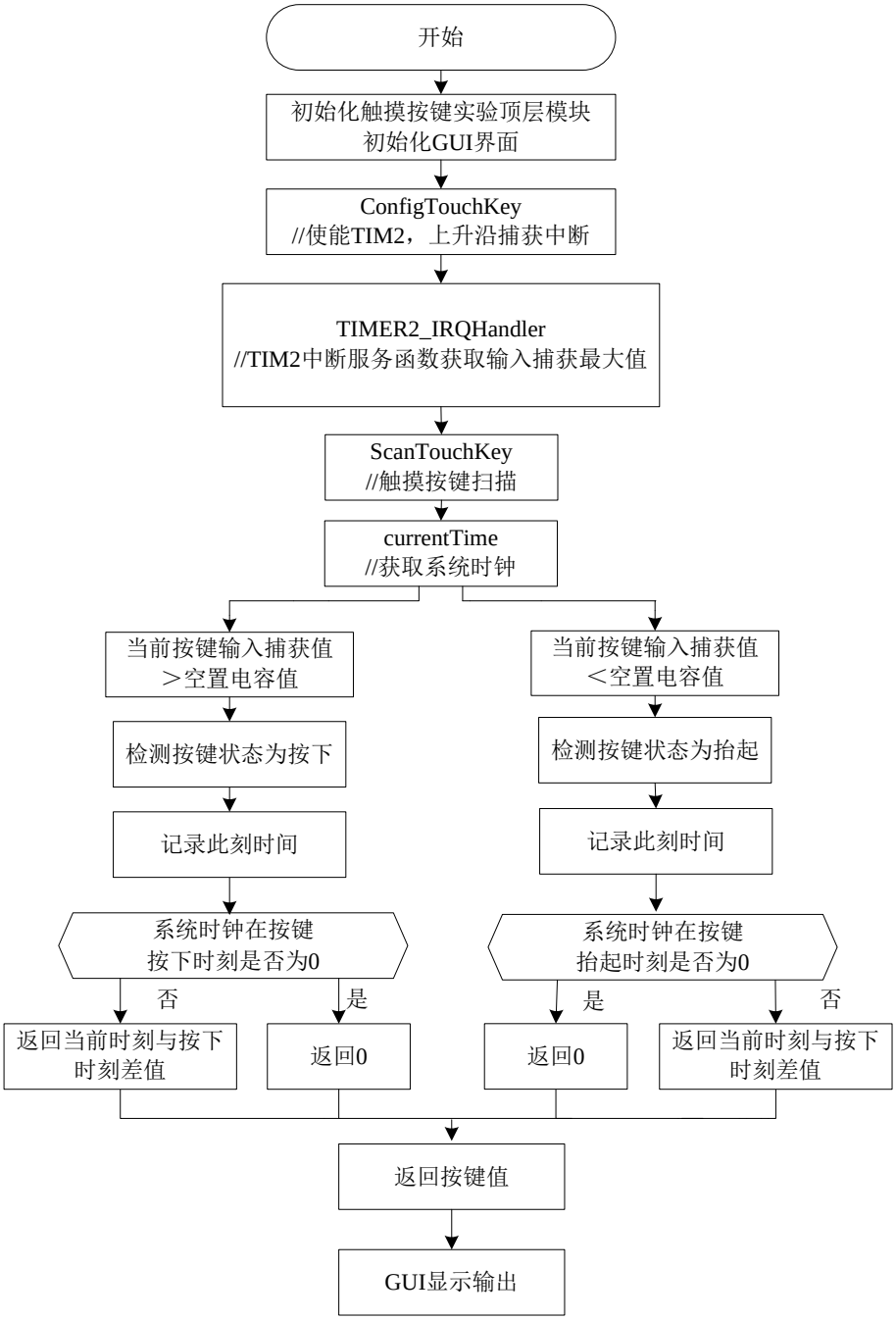

图 4-4　电容触摸按键实验流程图

4.3 实验代码解析

4.3.1 TouchKey 文件对

1．ToucKey.h 文件

在 ToucKey.h 文件的“API 函数声明”区，声明了 2 个 API 函数，如程序清单 4-1 所示。InitTouchKey 函数用于初始化触摸按键驱动模块，ScanTouchKey 函数用于进行触摸按键扫描任务。

程序清单 4-1

```
void InitTouchKey(void);                                          //初始化触摸按键驱动模块
u64   ScanTouchKey(void(*keyDown)(u64), void(*keyUp)(u64));       //触摸按键扫描
```

2．ToucKey.c 文件

在 TouchKey.c 文件的“宏定义”区，定义了触摸的门限值 TOUCH_GATE_VAL 为 100，即必须大于 s_iTouchDefaultValue（按键未曾按下时的输入捕获值）+ TOUCH_GATE_VAL 才认为是有效按下。

在 TouchKey.c 文件的“内部变量”区，定义了内部变量 s_iTouchDefaultValue 和 s_iKeyValue，如程序清单 4-2 所示。s_iTouchDefaultValue 为按键未按下时的输入捕获值，s_iKeyValue 为当前按键输入捕获值。

程序清单 4-2

```
static u16 s_iTouchDefaultValue = 0;                  //按键未按下时的输入捕获值
static u16 s_iKeyValue          = 0;                  //当前按键输入捕获值
```

在 TouchKey.c 文件的“内部函数声明”区，声明了 1 个内部函数，如程序清单 4-3 所示。ConfigTouchKey 函数用于配置触摸按键。

程序清单 4-3

```
static void ConfigTouchKey(void);                     //配置触摸按键
```

在 TouchKey.c 文件的“内部函数实现”区，首先实现了 ConfigTouchKey 函数，如程序清单 4-4 所示。

（1）第 6 至 9 行代码：本实验通过 TIMER2_CH3（PB1）进行输入捕获，采集电容触摸按键信号，使用 TIMER2 之前，需要先开启 TIMER2 时钟和 GPIOB 时钟。因此需要通过 rcu_periph_clock_enable 函数使能 GPIOB、TIMER2 和复用时钟。

（2）第 11 至 12 行代码：通过 gpio_init 函数将 PB1 引脚配置为复用推挽模式。

（3）第 14 至 25 行代码：通过 timer_deinit 函数复位 TIMER2，再通过 timer_init 函数配置

TIMER2，配置为边沿对齐模式，计数器向上计数等。

（4）第 27 至 33 行代码：配置 TIMER2_CH3 为输入捕获，并开启输入滤波器，捕获 PB1 的上升沿。

（5）第 38 至 42 行代码：通过 timer_interrupt_enable 函数使能 TIMER2 的更新中断，该函数涉及 TIMER2_DMAINTEN 的 UPIE，UPIE 用于禁止或使能更新中断。然后通过 nvic_irq_enable 函数使能 TIMER2 中断，并设置抢占优先级为 1，子优先级为 0。

程序清单 4-4

```
1.  static void ConfigTouchKey(void)
2.  {
3.      timer_parameter_struct       timer_initpara;
4.      timer_ic_parameter_struct timer_icintpara;
5.
6.      //使能 RCU 相关时钟
7.      rcu_periph_clock_enable(RCU_GPIOB);   //使能 GPIOB 的时钟
8.      rcu_periph_clock_enable(RCU_TIMER2); //使能 TIM2 时钟
9.      rcu_periph_clock_enable(RCU_AF);        //使能复用时钟
10.
11.     //配置 PB1 为复用推挽
12.     gpio_init(GPIOB, GPIO_MODE_IN_FLOATING, GPIO_OSPEED_50MHZ, GPIO_PIN_1);
13.
14.     //复位 TIM2
15.     timer_deinit(TIMER2);
16.
17.     //配置 TIM2
18.     timer_struct_para_init(&timer_initpara);                    //设置默认参数
19.     timer_initpara.period                 = 999;                    //设置自动重装载值（1ms）
20.     timer_initpara.prescaler              = 119;                    //设置预分频器值（1MHz）
21.     timer_initpara.alignedmode            = TIMER_COUNTER_EDGE; //设置边沿对齐
22.     timer_initpara.counterdirection   = TIMER_COUNTER_UP;    //设置向上计数模式
23.     timer_initpara.clockdivision        = TIMER_CKDIV_DIV1;    //设置时钟分割
24.     timer_initpara.repetitioncounter = 0;                       //设置重复计数
25.     timer_init(TIMER2, &timer_initpara);                        //根据参数初始化定时器
26.
27.     //配置 TIM2 CH3 输入捕获
28.     timer_channel_input_struct_para_init(&timer_icintpara);             //设置默认参数
29.     timer_icintpara.icfilter       = 3;                                 //配置滤波器
30.     timer_icintpara.icpolarity   = TIMER_IC_POLARITY_RISING;            //上升沿捕获
31.     timer_icintpara.icprescaler = TIMER_IC_PSC_DIV1;                    //设置预分频
32.     timer_icintpara.icselection = TIMER_IC_SELECTION_DIRECTTI;          //ic0 映射到 CI0
33.     timer_input_capture_config(TIMER2, TIMER_CH_3, &timer_icintpara); //根据参数配置 TIM2 CH3
34.
35.     //使能影子寄存器自动重装载
36.     timer_auto_reload_shadow_enable(TIMER2);
37.
38.     //使能定时器的更新中断
39.     timer_interrupt_enable(TIMER2, TIMER_INT_UP);
40.
```

```
41.     //定时器中断 NVIC 使能
42.     nvic_irq_enable(TIMER2_IRQn, 1, 0);
43.
44.     //使能 TIM2
45.     timer_enable(TIMER2);
46.  }
```

在 ConfigTouchKey 函数后为 TIMER2_IRQHandler 函数的实现代码，如程序清单 4-5 所示。

（1）第 12 至 18 行代码：本实验以 25ms 为一个周期判断手指是否按下电容触摸按键，通过比较 25ms 内的最大捕获值与电容充电时间来实现。TIMER2_IRQHandler 每隔 1ms 执行一次，因此，s_iKeyTimeCnt 每隔 1ms 执行加 1 操作后对 25 取余，其取值范围为 0~24。每当 s_iKeyTimeCnt 循环一个周期，即每隔 25ms，将最大捕获值更新到 s_iKeyValue（当前按键输入捕获值）中，并将最大捕获值清零。

（2）第 20 至 50 行代码：捕获计数器 s_iCapTimeCnt 每隔 1ms 执行加 1 操作后对 3 取余，其取值范围为 0~2。通过 switch 语句，以 3ms 为一个周期循环进行电容充放电并获取按键输入捕获值。当捕获计数器 s_iCapTimeCnt 为 0 时，将 PB1 引脚配置为推挽输出并拉低，使电容放电；当 s_iCapTimeCnt 为 1 时，通过 timer_flag_clear 函数，清空定时器输入捕获标志位，再将 PB1 引脚配置为浮空输入，使电容充电并触发 PB1 的输入捕获；当 s_iCapTimeCnt 为 2 时，获取输入捕获值，若该捕获值大于当前 25ms 周期内得到的最大捕获值，则将该捕获值更新为当前最大捕获值。

程序清单 4-5

```
1.  void TIMER2_IRQHandler(void)
2.  {
3.      static u8   s_iCapTimeCnt          = 0;
4.      static u8   s_iKeyTimeCnt          = 0;
5.      static u8   s_iCapture             = 0;
6.      static u16 s_iMaxCaptureValue = 0;
7.
8.      if (timer_interrupt_flag_get(TIMER2, TIMER_INT_FLAG_UP) == SET)   //判断定时器更新中断是否发生
9.      {
10.         timer_interrupt_flag_clear(TIMER2, TIMER_INT_FLAG_UP);          //清除定时器更新中断标志
11.
12.         //更新检测结果到 s_iKeyValue，并将最大值清零
13.         if(0 == s_iKeyTimeCnt)
14.         {
15.             s_iKeyValue = s_iMaxCaptureValue;
16.             s_iMaxCaptureValue = 0;
17.         }
18.         s_iKeyTimeCnt = (s_iKeyTimeCnt + 1) % 25;
19.
20.         //获取按键输入捕获值
21.         switch (s_iCapTimeCnt)
22.         {
```

```
23.
24.         //电容触摸按键放电
25.         case 0:
26.             gpio_init(GPIOB, GPIO_MODE_OUT_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_1);
27.             gpio_bit_reset(GPIOB, GPIO_PIN_1);
28.             break;
29.
30.         //清空定时器输入捕获标志位，配置电容触摸按键引脚为浮空输入
31.         //此时定时器计数值为 0，所以不用清空定时器计数值
32.         case 1:
33.             timer_flag_clear(TIMER2, TIMER_FLAG_CH3);
34.             gpio_init(GPIOB, GPIO_MODE_IN_FLOATING, GPIO_OSPEED_50MHZ, GPIO_PIN_1);
35.             break;
36.
37.         //获取输入捕获检测结果，并保存最大值
38.         case 2:
39.             s_iCapture = timer_channel_capture_value_register_read(TIMER2, TIMER_CH_3);
40.             if(s_iCapture > s_iMaxCaptureValue)
41.             {
42.                 s_iMaxCaptureValue = s_iCapture;
43.             }
44.             break;
45.
46.         default:
47.             break;
48.         }
49.         s_iCapTimeCnt = (s_iCapTimeCnt + 1) % 3;
50.     }
51. }
```

在 TouchKey.c 文件的“API 函数实现”区，首先实现 InitTouchKey 函数，如程序清单 4-6 所示，该函数的主要功能是初始化触摸按键驱动模块。

（1）第 3 行代码：通过 ConfigTouchKey 函数配置触摸按键，包括 PB1 引脚配置、TIMER2 配置、TIMER2_CH3 引脚的输入捕获配置和中断使能等。

（2）第 5 至 21 行代码：通过一个 while 语句，对按键未按下时的输入捕获值进行解析，若捕获值在 0 到 100 间，则打印当前捕获值，若不在 0 到 100 之间，则捕获值获取失败。最后打印按键此时的输入捕获值。

程序清单 4-6

```
1.  void InitTouchKey(void)
2.  {
3.      ConfigTouchKey(); //配置触摸按键
4.
5.      //获取按键未按下时的输入捕获值
6.      while(1)
7.      {
8.          s_iTouchDefaultValue = s_iKeyValue;
9.          if((s_iTouchDefaultValue > 0) && (s_iTouchDefaultValue < 100))
```

```
10.         {
11.             break;
12.         }
13.         else
14.         {
15.             printf("InitTouchKey: Falt to get default value\r\n");
16.             printf("InitTouchKey: value = %d\r\n", s_iTouchDefaultValue);
17.         }
18.         DelayNms(500);
19.     }
20.
21.     printf("Touck key default value: %d\r\n", s_iTouchDefaultValue);
22. }
```

在 TouchKey.c 文件的“API 函数实现”区，在 InitTouchKey 函数实现区后为 ScanTouchKey 函数的实现代码，如程序清单 4-7 所示，ScanTouchKey 函数的功能是进行触摸按键扫描。

（1）第 12 至 20 行代码：通过 if 语句，判断按键当前输入捕获值状态。若当前输入捕获值大于触摸门限值与按键未按下时输入捕获值的和，则判断为按键按下，否则判断为按键抬起。

（2）第 22 至 60 行代码：通过两个 if 语句，分别处理按键按下和抬起的状态。若按键按下，则使用 s_iKeyDownBeginTime 记录按键按下时刻的系统时间，并进行回调处理，若按键抬起，则使用 s_iKeyUpBeginTime 记录按键抬起时刻系统时间，并进行回调处理。

（3）第 62 至 76 行代码：通过 s_enumKeyLastState 储存当前按键状态。再通过 if 语句判断当前按键状态，若按键按下，则返回按下的时长；若按键未按下，则返回 0。

程序清单 4-7

```
1.  u64 ScanTouchKey(void(*keyDown)(u64), void(*keyUp)(u64))
2.  {
3.      static EnumTouchKeyState s_enumKeyLastState    = TOUCH_KEY_UP; //上一次检测结果
4.      static EnumTouchKeyState s_enumKeyNewState     = TOUCH_KEY_UP; //此次测量结果
5.      static u64               s_iKeyDownBeginTime = 0;              //按键按下时刻的系统时钟
6.      static u64               s_iKeyUpBeginTime     = 0;            //按键抬起时刻的系统时钟
7.      u64 currentTime;
8.
9.      //获取系统时钟
10.     currentTime = GetSysTime();
11.
12.     //判断按键状态
13.     if(s_iKeyValue > (s_iTouchDefaultValue + TOUCH_GATE_VAL))
14.     {
15.         s_enumKeyNewState = TOUCH_KEY_DOWN;
16.     }
17.     else
18.     {
19.         s_enumKeyNewState = TOUCH_KEY_UP;
20.     }
21.
22.     //按下处理
```

```
23.     if((TOUCH_KEY_UP == s_enumKeyLastState) && (TOUCH_KEY_DOWN == s_enumKeyNewState))
24.     {
25.         //记录按键按下时刻系统时间
26.         s_iKeyDownBeginTime = currentTime;
27.
28.         //回调处理
29.         if(NULL != keyDown)
30.         {
31.             if(0 == s_iKeyUpBeginTime)
32.             {
33.                 keyDown(0);
34.             }
35.             else
36.             {
37.                 keyDown(currentTime - s_iKeyUpBeginTime);
38.             }
39.         }
40.     }
41.
42.     //抬起处理
43.     if((TOUCH_KEY_DOWN == s_enumKeyLastState) && (TOUCH_KEY_UP == s_enumKeyNewState))
44.     {
45.         //记录按键抬起时刻时钟
46.         s_iKeyUpBeginTime = currentTime;
47.
48.         //回调处理
49.         if(NULL != keyUp)
50.         {
51.             if(0 == s_iKeyDownBeginTime)
52.             {
53.                 keyUp(0);
54.             }
55.             else
56.             {
57.                 keyUp(currentTime - s_iKeyDownBeginTime);
58.             }
59.         }
60.     }
61.
62.     //保存按键状态
63.     s_enumKeyLastState = s_enumKeyNewState;
64.
65.     //按键按下时返回按下时长
66.     if(TOUCH_KEY_DOWN == s_enumKeyNewState)
67.     {
68.         return currentTime - s_iKeyDownBeginTime;
69.     }
70.
71.     //按键未按下
72.     else
73.     {
74.         return 0;
75.     }
```

```
76.  }
```

4.3.2 TouchKeyTop 文件对

1．TouchKeyTop.h 文件

在 TouchKeyTop.h 文件的“API 函数声明”区，声明了 2 个 API 函数，如程序清单 4-8 所示。InitTouchKeyTop 函数的主要功能为初始化电容触摸按键实验顶层模块，TouchKeyToptask 函数的主要功能为执行电容触摸按键实验任务。

程序清单 4-8

```
void InitTouchKeyTop(void);          //初始化电容触摸按键实验顶层模块
void TouchKeyTopTask(void);          //电容触摸按键实验任务
```

2．TouchKeyTop.c 文件

在 TouchKeyTop.c 文件的“内部变量”区，声明了字符串转换缓冲区 s_arrStringBuf，如程序清单 4-9 所示。

程序清单 4-9

```
static char s_arrStringBuf[64]; //字符串转换缓冲区
```

在 TouchKeyTop.c 文件的“内部函数声明”区，声明了 3 个函数，如程序清单 4-10 所示，DisplayBackground 函数用于绘制 GUI 界面背景，KeyDownCallback 和 KeyUpCallback 分别为按键按下和抬起的回调函数。

程序清单 4-10

```
static void DisplayBackground(void);       //绘制背景
static void KeyDownCallback(u64 upTime); //按键按下回调函数
static void KeyUpCallback(u64 downTime); //按键抬起回调函数
```

在 TouchKeyTop.c 文件的“内部函数实现”区，首先实现了 DisplayBackground 函数，绘制当前实验 GUI 界面背景，如程序清单 4-11 所示。

程序清单 4-11

```
1.   static void DisplayBackground(void)
2.   {
3.      //背景图片控制结构体
4.      StructJpegImage backgroundImage;
5.
6.      //初始化 backgroundImage
7.      backgroundImage.image = (unsigned char*)s_arrJpegBackgroundImage;
8.      backgroundImage.size    = sizeof(s_arrJpegBackgroundImage) / sizeof(unsigned char);
9.
10.     //解码并显示图片
11.     DisplayJPEGInFlash(&backgroundImage, 0, 0);
12.  }
```

在 DisplayBackground 函数实现区后为 KeyDownCallback 和 KeyUpCallback 函数的实现代码，如程序清单 4-12 所示。这两个函数分别为按键按下和抬起的回调函数，实现的主要功能是将按键按下时长和按键抬起时长转换成字符串，然后分别输出到 GUI 界面上的终端中。

程序清单 4-12

```
1.  static void KeyDownCallback(u64 upTime)
2.  {
3.      //字符串转换
4.      sprintf(s_arrStringBuf, "Key down: up time = %lld ms\r\n", upTime);
5.
6.      //输出到终端显示
7.      ShowStringLineInGUITerminal(s_arrStringBuf);
8.
9.      //打印到串口显示
10.     printf("%s", s_arrStringBuf);
11. }
12.
13. static void KeyUpCallback(u64 downTime)
14. {
15.     //字符串转换
16.     sprintf(s_arrStringBuf, "Key up: down time = %lld ms\r\n", downTime);
17.
18.     //输出到终端显示
19.     ShowStringLineInGUITerminal(s_arrStringBuf);
20.
21.     //打印到串口显示
22.     printf("%s", s_arrStringBuf);
23. }
```

在 TouchKeyTop.c 文件的“API 函数实现”区，首先实现 InitTouchKeyTop 函数，如程序清单 4-13 所示。InitTouchKeyTop 函数主要的主要功能是初始化电容触摸按键实验顶层模块，包括设置 LCD 显示方向为竖屏、绘制 GUI 界面背景和在界面上创建终端等。

程序清单 4-13

```
1.  void InitTouchKeyTop(void)
2.  {
3.      //LCD 竖屏显示
4.      LCDDisplayDir(0);
5.      LCDClear(GBLUE);
6.
7.      //绘制背景
8.      DisplayBackground();
9.
10.     //创建终端
11.     InitGUITerminal();
12.     ClearGUITerminal();
13.
14.     //终端输出“Touch Key Test”
15.     ShowStringLineInGUITerminal("Touch Key Test");
```

```
16.  }
```

在 InitTouchKeyTop 函数实现区后为 TouchKeyTopTask 函数的实现代码，如程序清单 4-14 所示。在 TouchKeyTopTask 函数中调用 ScanTouchKey 函数进行触摸按键扫描。

程序清单 4-14

```
1.   void TouchKeyTopTask(void)
2.   {
3.       //触摸按键扫描
4.       ScanTouchKey(KeyDownCallback, KeyUpCallback);
5.   }
```

4.3.3 Main.c 文件

在 Main.c 文件的 Proc1msTask 函数中，每 20ms 调用一次 TouchKeyTopTask 函数，执行电容触摸按键实验任务，如程序清单 4-15 所示。

程序清单 4-15

```
1.   static   void   Proc1msTask(void)
2.   {
3.       static u8 s_iCnt = 0;
4.       if(Get1msFlag())
5.       {
6.           s_iCnt++;
7.           if(s_iCnt >= 20)
8.           {
9.               ScanTouch();              //触摸屏扫描
10.              TouchKeyTopTask();   //电容触摸按键实验任务
11.              s_iCnt = 0;
12.          }
13.      Clr1msFlag();
14.      }
15.  }
```

4.3.4 实验结果

代码编写完成并编译通过后，下载程序并进行复位。下载完成后，可以观察到开发板上的 LCD 显示如图 4-5 所示的 GUI 界面。

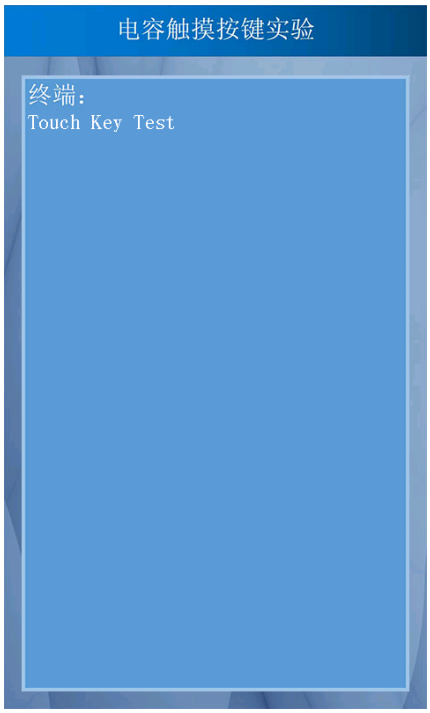

图 4-5　编译后 GUI 界面显示

用手指按下 GD32F3 苹果派开发板上的电容触摸按键，按下一段时间后，抬起手指松开按键，在 GUI 界面的终端中会显示按下的时长，如图 4-6 所示，“Key down：up time”为上一次松开电容触摸按键到此次按下按键的时间差，“Key up：down time”为按下电容触摸按键到松开按键的时间差。

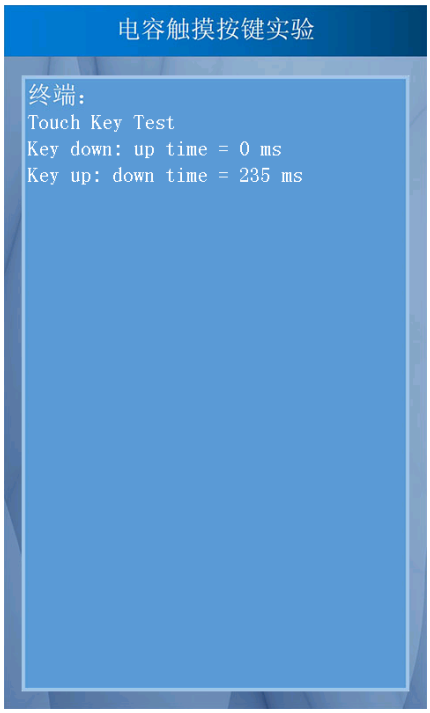

图 4-6　按下抬起后屏幕输出结果

本章任务

本章实验中，通过检测手指是否按下电容触摸按键，实现了对按键按下和松开持续时间的计时。现尝试通过检测手指是否按下电容触摸按键来控制开发板上的 LED 状态，LED_1 初始状态为熄灭，当用手指按下电容触摸按键时，LED_1 持续点亮，直至松开按键时，LED_1 熄灭。

本章习题

1．简述电容充放电原理及过程。

2．简述本章实验中，检测手指是否按下电容触摸按键的原理。

3．在本章实验中，如何提升检测手指按下和松开的灵敏度。

5 读写内部 FLASH 实验

在《GD32F3 苹果派开发进阶教程》中，介绍了内外部 SRAM 和 NAND Flash。本章将对微控制器中另一类存储器 Flash 进行介绍，最后基于 GD32F3 苹果派开发板读写微控制器内部 Flash。

5.1 实验内容

本章的主要内容是了解 Flash 的分类和基本构成，掌握微控制器内部 Flash 的读写方法以及涉及的相关寄存器和固件库函数，最后基于 GD32F3 苹果派开发板设计一个读写内部 Flash 实验，通过 LCD 上的 GUI 界面，实现读写微控制器内部 Flash。

5.2 实验原理

5.2.1 内部 Flash 介绍

GD32F30x 系列微控制器的内部 Flash 主要由主存储闪存块、信息块和可选字节块 3 部分组成，如表 5-1 所示，下面依次介绍三个部分的作用。

表 5-1 Flash 的构成

闪存块		名称	地址范围	大小（字节）
主存储闪存块		第 0 页	0x0800 0000～0x0800 07FF	2KB
		第 1 页	0x0800 0800～0x0800 0FFF	2KB
		第 2 页	0x0800 1000～0x0800 17FF	2KB
		⋮	⋮	⋮
		第 255 页	0x0807 F800～0x0807 FFFF	2KB
		第 256 页	0x0808 0000~0x0808 0FFF	4KB
		第 257 页	0x0808 1000～0x0808 1FFF	4KB
		⋮	⋮	⋮
		第 895 页	0x082F F000～0x082F FFFF	4KB
信息块	GD32F30x_HD	Boot loader 区	0x1FFF F000～0x1FFF F7FF	2KB
	GD32F30x_XD		0x1FFF E000～0x1FFF F7FF	6KB
	GD32F30x_CL		0x1FFF B000～0x1FFF F7FF	18KB

可选字节块	可选字节	0x1FFFF F800~0x1FFF F80F	16B

1．主存储区

GD32F3 苹果派开发板板载的微控制器型号为 GD32F303ZET6，属于 GD32F30x_HD 系列微控制器，因此内部 Flash 容量为 512K，这里的 512K 即为主存储区的容量，在微控制器中主存储区被划分为 0~255 页，每页大小为 2KB。表 5-1 所示的主存储区的阴影部分，为本系列微控制器无效部分。主存储区是用户进行读写 Flash 时使用到的主要区域，而不同的页实质就是不同的扇区，与所有 Flash 一样，在写入数据前，要先按页（扇区）擦除内部的数据。

2．信息块

信息块是用户可访问的区域，在芯片出厂时固化了 GD32 的启动代码，用于实现串口、USB 以及 CAN 等 ISP 烧录功能。不同系列的微控制器信息块大小不同，GD32F30x_HD、GD32F30x_XD 和 GD32F30x_CL 分别为高密度产品、超高密度产品和互联型产品，其信息块大小也从 2KB 到 18KB 不等，本开发板板载的微控制器 GD32F303ZET6 属于 GD32F30x_HD 系列，即高密度产品，信息块大小为 2KB。

3．可选字节块

可选字节块大小为 16 字节，一般用于配置 Flash 写保护、读保护和看门狗等功能，可以通过修改 Flash 的选项字节寄存器进行修改。

5.2.2 Flash 读写过程

1．Flash 读操作

NOR FLASH 可以随机寻址，即通过具体地址获取对应地址上的数据，其取指令和取数据操作分别使用 CPU 的 IBUS 和 DBUS 总线。例如，从地址 addr 读取 1 字的数据（1 字为 32 位）。代码如下：

```
data=*(u32*)addr;
```

将 addr 强制转换为 u32 类型的指针，然后取该指针所指向的地址的值，即可得到地址 addr 上 1 字的数据。类似的，将上面的 u32 类型改为 u16 类型，即可读取指定地址上半字大小的数据。

2．Flash 写入操作

Flash 的写入操作主要有以下三个步骤：解锁、页擦除和数据写入。

（1）解锁

由于内部 Flash 用于存储程序，为了防止误操作修改这些关键内容，芯片复位后默认通过

控制寄存器 FMC_CTLx 进行上锁，禁止修改 Flash 的内容，达到保护的作用。因此向 Flash 写入数据前需要先解锁。过程如下：

复位后，FMC_CTL0 寄存器进入锁定状态，LK 位被置为 1。此时先后向 FMC_KEY0 寄存器写入 0x45670123 和 0xCDEF89AB 两个数据，两次写操作后，FMC_CTL0 寄存器的 LK 位被置 0，FMC_CTL0 寄存器解锁，此时，微控制器可向 Flash 写入数据。

（2）页擦除

由于 Flash 的写入操作仅能将内部的 1 写为 0，而不能将 0 写为 1，因此在写入数据前还需要进行页擦除。FMC 的页擦除功能可以将 Flash 主存储区相应的页初始化为高电平，而不影响其他页的内容。FMC 擦除页步骤如下：

①确保 FMC_CTLx 寄存器不处于锁定状态；

②检查 FMC_STATx 寄存器的 BUSY 位来判定闪存是否正处于擦写访问状态，若 BUSY 位为 1，则需等待该操作结束，BUSY 位变为 0；

③置位 FMC_CTLx 寄存器的 PER 位；

④将待擦除页的绝对地址（0x08XX XXXX）写到 FMC_ADDRx 寄存器；

⑤将 FMC_CTLx 寄存器的 START 位置 1 以发送页擦除命令到 FMC；

⑥等待擦除指令执行完毕，即 FMC_STATx 寄存器的 BUSY 位清 0；

⑦此时页擦除完成，可通过 DBUS 读并验证该页是否擦除成功。

页擦除流程图如图 5-1 所示。

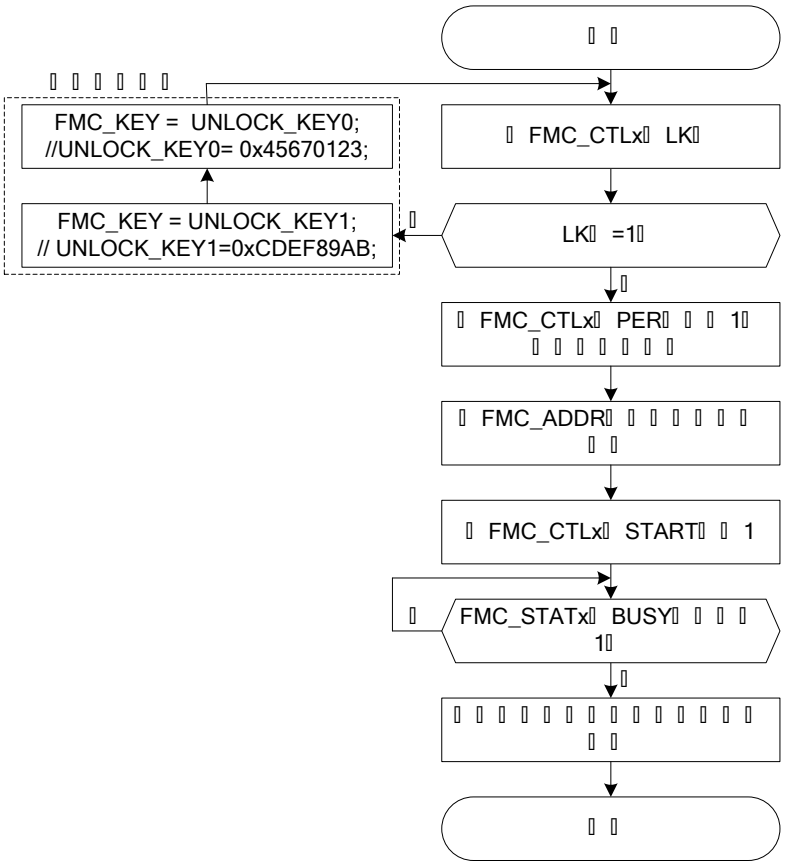

图 5-1　Flash 页擦除过程

（3）数据写入

FMC 提供了一个 32 位整字/16 位半字/位编程功能，用来修改主存储闪存块内容。写入时不仅仅只是通过指针向对应的地址赋值，还需要配置一系列的寄存器，步骤如下：

①确保 FMC_CTLx 寄存器不处于锁定状态；

②等待 FMC_STATx 寄存器的 BUSY 位变为 0；

③置位 FMC_CTLx 寄存器的 PG 位；

④DBUS 写一个 32 位整字/16 位半字到目的绝对地址（0x08XX XXXX）；

⑤等待编程指令执行完毕，即 FMC_STATx 寄存器的 BUSY 位清 0；

此时数据写入完成，可通过 DBUS 读并验证是否编程成功。数据写入流程图如图 5-2 所示：

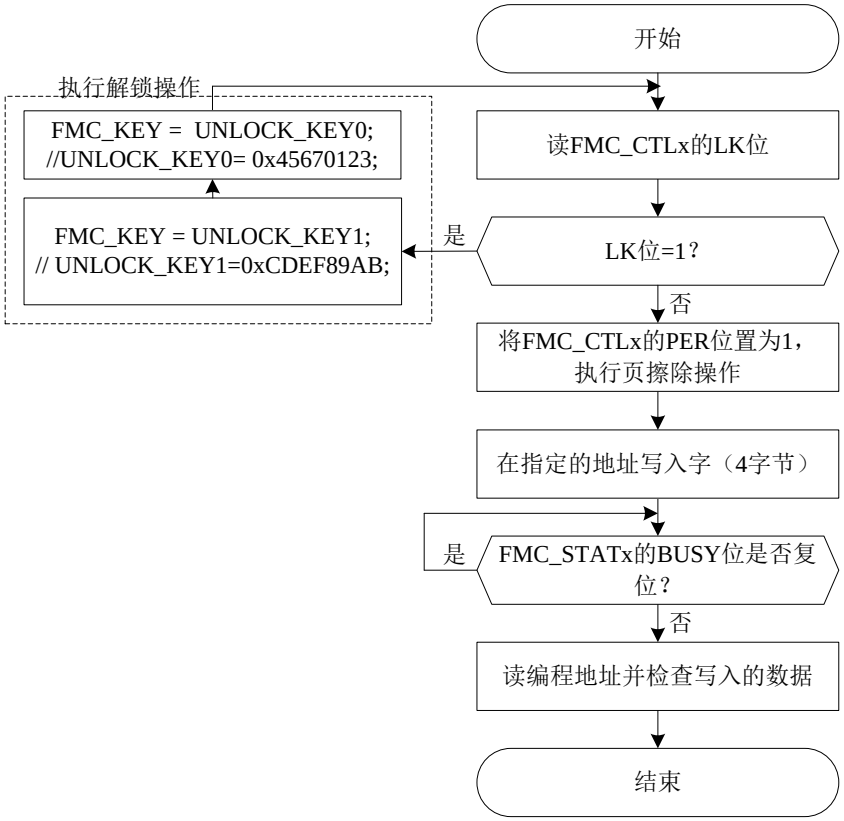

图 5-2　数据写入过程

5.2.3　闪存控制器部分寄存器

GD32F30x 系列微控制器的内部 Flash 的操作主要通过其自带的闪存控制器 FMC 来完成的。FMC 提供了片上闪存需要的所有功能，包括页擦除，整片擦除，以及 32 位整字/16 位半字/位编程等闪存操作。

FMC 包含 14 个 32 位寄存器，分别是：等待状态寄存器(FMC_WS)、解锁寄存器(FMC_KEY0)、选项字节操作解锁寄存器（FMC_OBKEY）、状态寄存器 0(FMC_STAT0)、控制寄存器 0(FMC_CTL0)、地址寄存器 0(FMC_ADDR0)、选项字节状态寄存器(FMC_OBSTAT)、擦除/编程保护寄存器(FMC_WP)、解锁寄存器 1(FMC_KEY1)、状态寄存器 1(FMC_STAT1)、控制寄存器 1(FMC_CTL1)、地址寄存器 1(FMC_ADDR1)、等待状态使能寄存器(FMC_WSEN)、产品 ID 寄存器(FMC_PID)，下面简要介绍上述列举的部分寄存器：

1．解锁寄存器（FMC_KEY0）

FMC_KEY0 的结构、偏移地址和复位值如图 5-3 所示，对部分位的解释说明如表 5-2 所示。

偏移地址：0x04
复位值：0x0000 0000
该寄存器只能按字（32位）访问

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
KEY[31:16]															
w															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
KEY[15:0]															
w															

图 5-3　FMC_KEY0 的结构、偏移地址和复位值

表 5-2　FMC_KEY0 的部分位说明

位/位域	名称	描述
31:0	KEY[31:0]	FMC_CTL0 解锁寄存器。 这些位仅能被软件写。 写解锁值到 KEY[31:0]可以解锁 FMC_CTL0 寄存器

2．控制寄存器 0（FMC_CTL0）

FMC_CTL0 的结构、偏移地址和复位值如图 5-4 所示，对部分位的解释说明如表 5-3 所示。

偏移地址：0x10
复位值：0x0000 0080
该寄存器只能按字（32位）访问

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留			ENDIE	保留	ERRIE	OBWEN	保留	LK	START	OBER	OBPG	保留	MER	PER	PG
			rw		rw	rw		rs	rs	rw	rw		rw	rw	rw

图 5-4　FMC_CTL0 的结构、偏移地址和复位值

表 5-3　FMC_CTL0 的部分位说明

位/位域	名称	描述
12	ENDIE	操作结束中断使能位。 软件置 1 和清 0。 0：无硬件中断产生； 1：使能操作结束中断
10	ERRIE	出错中断使能位。 软件置 1 和清 0。 0：无硬件中断产生；

		1：使能出错中断
9	OBWEN	可选字节擦除/编程使能位。 当正确的序列写入 FMC_OBKEY 寄存器，此位由硬件置 1。此位可以被软件清 0
7	LK	FMC_CTL0 寄存器锁定标志位。 当正确的序列写入 FMC_KEY0 寄存器，此位由硬件清 0。此位可以由软件置 1
6	OBER	可选字节擦除命令位。 软件置 1 和清 0。 0：无作用； 1：可选字节擦除命令
4	OBPG	可选字节编程命令位。 软件置 1 和清 0。 0：无作用； 1：可选字节编程命令
2	MER	主存储块整片擦除命令位。 软件置 1 和清 0。 0：无作用； 1：主存储块整片擦除命令
1	PER	主存储块页擦除命令位。 软件置 1 和清 0。 0：无作用； 1：主存储块页擦除命令
0	PG	主存储块编程命令位。 软件置 1 和清 0。 0：无作用； 1：主存储块编程命令

3．状态寄存器 0（FMC_STAT0）

FMC_STAT0 的结构、偏移地址和复位值如图 5-5 所示，对部分位的解释说明如表 5-4 所示。

偏移地址：0x0C

复位值：0x0000 0000

该寄存器只能按字（32位）访问

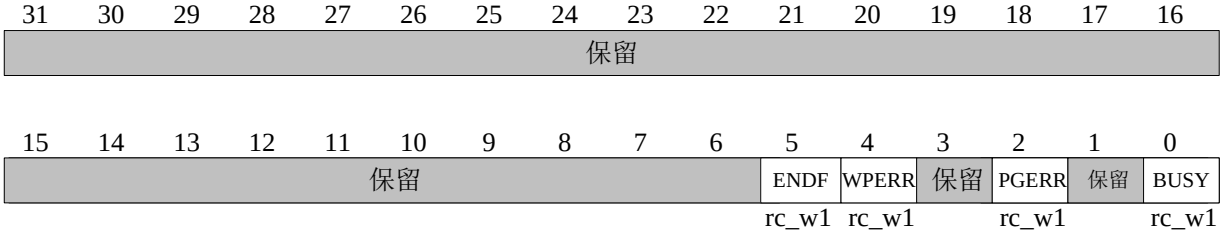

图 5-5　FMC 的结构、偏移地址和复位值

表 5-4　FMC_STAT0 的部分位说明

位/位域	名称	描述

5	ENDF	操作结束标志位。 操作成功执行后，此位被硬件置 1，软件写 1 清 0
4	WPERR	擦除/编程保护错误标志位。 在受保护的页上擦除/编程操作时，此位被硬件置 1。软件写 1 清 0
2	PGERR	编程错误标志位。 当被编程区域状态不为 0xFFFF 时，对闪存编程，此位被硬件置 1。软件写 1 清 0
0	BUSY	闪存忙标志。 当闪存操作正在进行时，此位被置 1。当操作结束或者出错，此位被清 0

4．地址寄存器（FMC_ADDR0）

FMC_ADDR0 的结构、偏移地址和复位值如图 5-6 所示，对部分位的解释说明如表 5-5 所示。

偏移地址：0x14
复位值：0x0000 0000
该寄存器只能按字（32位）访问

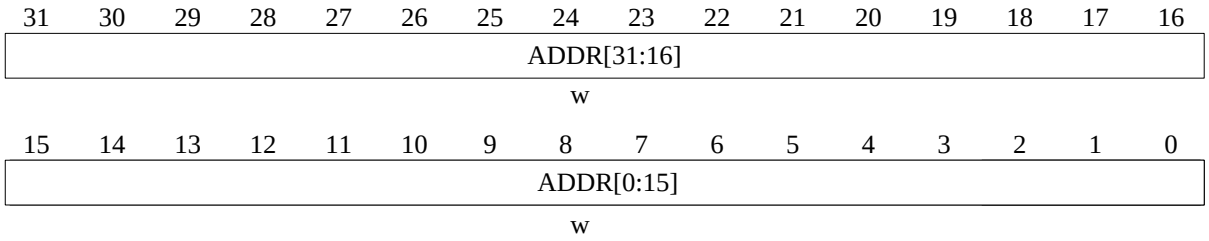

图 5-6　FMC_ADDR0 的结构、偏移地址和复位值

表 5-5　FMC_ADDR0 的部分位说明

位/位域	名称	描述
31:0	ADDR[31:0]	闪存擦除或编程地址。 该位通过软件设置。 ADDR 位是闪存擦除命令的地址

5.2.4　闪存控制器部分固件库函数

本实验涉及的 Flash 固件库函数包括 fmc_unlock、fmc_page_erase、fmc_word_program、fmc_lock。这些函数均在固件库的 gd32f30x_fmc.h 文件中声明，在 gd32f30x_fmc.c 文件中实现，下面简要介绍这些固件库函数：

1．fmc_unlock

函数 fmc_unlock()的功能是解锁 Flash 编写擦除控制器。该函数通过向 FMC_KEY0 寄存器写入特定序列 UNLOCK_KEY0，UNLOCK_KEY1 以解锁 FMC_CTL0 寄存器。该函数的描述如表 5-6 所示。

表 5-6　函数 fmc_loc 的描述

函数名	fmc_unlock()
函数原型	void fmc_unlock(void)
功能描述	解锁 FMC_CTL0 寄存器
输入参数	无
输出参数	无
返回值	void

例如，解锁 Flash，代码如下：

```
fmc_unlock();
```

2．fmc_page_erase

函数 fmc_page_erase 的功能是进行页擦除。该函数通过向 FMC_CTL0 和 FMC_ADDR0 寄存器写入相应的参数完成页擦除。该函数的描述如表 5-7 所示。

表 5-7　函数 fmc_page_erase 的描述

函数名	fmc_page_erase
函数原型	fmc_state_enum fmc_page_erase(uint32_t page_address)
功能描述	擦除 Flash 的页
输入参数	page_address：待擦除的页地址
输出参数	无
返回值	擦除操作状态

例如，擦除 Flash 的地址为 addr 的页面，代码如下：

```
fmc_page_erase(addr);
```

3．fmc_word_program

函数 fmc_word_program 的功能是在指定地址编写 1 字的数据。该函数的描述如表 5-8 所示。

表 5-8　函数 fmc_word_program 的描述

函数名	fmc_word_program
函数原型	fmc_state_enum fmc_word_program(uint32_t address, uint32_t data)
功能描述	在指定地址写 1 字
输入参数 1	address：待写入数据的地址
输入参数 2	data：待写入的数据
输出参数	无
返回值	操作状态

例如，向 addr1 地址写入 data1，代码如下：

```
u32 data1 = 0x1234567;
u32 addr1 = 0x8000000;
```

```
fmc_state= fmc_word_program (addr1, data1);
```

4．fmc_lock

函数 fmc_lock 的功能是锁定 Flash 编写擦除控制器。该函数通过向 FMC_CTLx 寄存器写入特定序列 FMC_CTL0_LK，FMC_CTL1_LK 后锁定 FMC_CTLx 寄存器。该函数的描述如表 5-9 所示。

表 5-9　函数 fmc_lock 的描述

函数名	fmc_lock()
函数原型	void fmc_lock(void)
功能描述	锁定 FMC_CTLx 寄存器
输入参数	无
输出参数	无
返回值	void

例如，锁定 Flash，代码如下：

```
fmc_lock();
```

5.3　实验代码解析

5.3.1　Flash 文件对

1．Flash.h 文件

在 Flash.h 文件的“宏定义”区，定义了 Flash 每页的空间大小为 2KB 以及 Flash 的读写起始地址和结束地址。

在 Flash.h 文件的“API 函数声明”区，声明了 3 个 API 函数，如程序清单 5-1 所示。InitFlash 函数用于初始化 Flash 模块，FlashWriteWord 函数用于向 Flash 相应地址写入相应长度的数据，FlashReadWord 函数用于从 Flash 相应地址读取相应长度的数据。

程序清单 5-1

```
void InitFlash(void);                                                    //初始化内部 Flash 模块
void FlashWriteWord(uint32_t startAddr, uint32_t* pBuf, uint32_t len); //向 Flash 中写入字
void FlashReadWord(uint32_t startAddr, uint32_t* pBuf, uint32_t len);    //从 Flash 读取字
```

2．Flash.c 文件

在 Flash.c 文件的“内部变量”区，声明了内部变量 s_arrFlashBuff[FLASH_PAGE_SIZE / 4]，如程序清单 5-2 所示。数组 s_arrFlashBuff 为内部 Flash 的写入缓冲区，FLASH_PAGE_SIZE 为 Flash 的页空间大小。由于数据按字（32 位）写入，因此将宏定义 FLASH_PAGE_SIZE 除以 4。因此，数组 s_arrFlashBuff 的大小为 2048/4=512。

程序清单 5-2

```
static u32 s_arrFlashBuf[FLASH_PAGE_SIZE / 4];  //Flash 写入缓冲区，大小为 2K 字节
```

在 Flash.c 文件的“API 函数实现”区，首先实现 InitFlash 函数，InitFlash 函数用于初始化 Flash 模块，如程序清单 5-3 所示，本实验不需要对 Flash 模块初始化，因此该函数为空函数。如果需要对 Flash 模块进行初始化，可添加相应函数体后在 main.c 文件中的 InitHardware 函数中调用本函数完成初始化。

程序清单 5-3

```
1.  void InitFlash(void)
2.  {
3.
4.  }
```

在 InitFlash 函数实现区后为 FlashWriteWord 函数的实现代码，如程序清单 5-4 所示。其中，参数 startAddr 是数据写入地址，由于数据按字（32 位）写入，因此该地址必为 4 的倍数，pBuf 是待写入数据存放的数组地址，len 是需要写入的以字为单位的数据个数。

（1）第 10 至 16 行代码：根据输入参数计算相应的页地址和页内偏移量并通过 fmc_unlock 函数解锁 Flash。

（2）第 24 至 76 行代码：将一整页的数据读取至缓冲区并将该页擦除，修改缓冲区相应位置的数据后通过 for 循环将缓冲区中的数据写入原地址。

（3）第 79 行代码：通过 fmc_lock 函数对 Flash 上锁.

程序清单 5-4

```
1.  void FlashWriteWord(uint32_t startAddr, uint32_t* pBuf, uint32_t len)
2.  {
3.      u32 i;                  //循环变量
4.      u32 pageAddr;     //页地址，即起始地址 startAddr 所在的页地址
5.      u32 pageOff;      //页内偏移地址（32 位计算），即起始地址 startAddr 所在的页的偏移地址
6.      u32 rwAddr;        //读写地址
7.      u32 dataCnt;      //已写入数据量
8.
9.      //计算页地址和页内偏移量
10.     pageAddr = startAddr;
11.     pageOff    = 0;
12.     while(0 != (pageAddr % FLASH_PAGE_SIZE))
13.     {
14.         pageAddr = pageAddr - 4;
15.         pageOff++;
16.     }
17.     //解锁 Flash，准备写入
18.     fmc_unlock();
19.
20.     //已写入数据量清零
```

```
    dataCnt = 0;

    //写入 Flash 时需要先读取整页的内容到缓冲区，然后擦除一整页，将缓冲区修改后再写回 Flash
    while(1)
    {
        //读取一整页的数据到缓冲区
        rwAddr = pageAddr;
        for(i = 0; i < FLASH_PAGE_SIZE / 4; i++)
        {
            s_arrFlashBuf[i] = *(u32*)rwAddr;
            rwAddr = rwAddr + 4;
        }


        //擦除一整页
        fmc_page_erase(pageAddr);

        //修改缓冲区内的内容
        while(pageOff < (FLASH_PAGE_SIZE / 4))
        {
            //将数据保存到缓冲区
            s_arrFlashBuf[pageOff] = pBuf[dataCnt];

            //已写入数据加一
            dataCnt++;

            //页内偏移量加一
            pageOff++;

            //写入完成
            if(dataCnt >= len)
            {
                break;
            }
        }

        //页内偏移量清零
        pageOff = 0;

        //将修改后的缓冲区内容写回 Flash
        rwAddr = pageAddr;
        for(i = 0; i < FLASH_PAGE_SIZE / 4; i++)
        {
            fmc_word_program(rwAddr, s_arrFlashBuf[i]);
            rwAddr = rwAddr + 4;
        }

        //更新到下一页首地址
        pageAddr = pageAddr + FLASH_PAGE_SIZE;

        //写入完成
        if(dataCnt >= len)
        {
```

```
74.             break;
75.         }
76.     }
77.
78.     //Flash 上锁
79.     fmc_lock();
80. }
```

在 FlashWriteWord 函数实现区后为 FlashReadWord 函数的实现代码，如程序清单 5-5 所示。参数 startAddr 是数据读取地址，该地址必须是 4 的倍数，pBuf 是待读取数据读取后的存放地址，len 是需要读取的以字为单位的数据个数。由于内部 Flash 类型为 NORFlash，地址线与数据线分开，因此可以直接通过地址读取数据。

程序清单 5-5

```
1.  void FlashReadWord(uint32_t startAddr, uint32_t* pBuf, uint32_t len)
2.  {
3.      u32 addr;
4.      u32 i;
5.
6.      addr = startAddr;
7.      for(i = 0; i < len; i++)
8.      {
9.          pBuf[i] = *(u32*)addr;
10.         addr = addr + 4;
11.     }
12. }
```

5.3.2 ReadwriteInFlash 文件对

1．ReadwriteInFlash.h 文件

在 ReadwriteInFlash.h 文件的“API 函数声明”区，声明了 2 个 API 函数，如程序清单 5-6 所示。InitReadWriteInFlash 函数用于初始化读写内部 Flash 模块，ReadWriteInFlashTask 函数用于执行读写内部 Flash 模块任务。

程序清单 5-6

```
void InitReadWriteInFlash(void);   //初始化读写内部 Flash 模块
void ReadWriteInFlashTask(void);   //读写内部 Flash 模块任务
```

2．ReadwriteInFlash.c 文件

在 ReadwriteInFlash.c 文件的“包含头文件”区，包含了 Flash.h 和 GUITop.h 等头文件，Flash.h 文件的代码包含着对内部 Flash 的块读写函数的声明，ReadWriteInFlash.c 文件需要通过调用这些函数完成对内部 Flash 的读写，因此需要包含 Flash.h 头文件。由于地址、数据等信息都需要通过 LCD 进行显示，因此，还需要包含 GUITop.h 头文件。

在 ReadwriteInFlash.c 文件的“宏定义”区，定义了关于显示字符最大长度的宏定义

MAX_STRING_LEN 为 64，即 LCD 显示字符串的最大长度为 64 位。

在 ReadwriteInFlash.c 文件的“内部变量”区，声明了内部变量 s_structGUIDev、s_arrBuff[512]、s_arrStringBuff[MAX_STRING_LEN]，如程序清单 5-7 所示。s_structGUIDev 是 GUI 的结构体，包含读写地址、读写函数等数据；s_arrBuff[512]为内部 Flash 的读写缓冲区，512 表示该工程每次读写 Flash 获得的数据最多为 512 位；s_arrStringBuff[MAX_STRING_LEN]为字符串转换缓冲区，该缓冲区最多转换 64 位字符。

程序清单 5-7

```
static StructGUIDev s_structGUIDev;                          //GUI 设备结构体
static u32    s_arrBuff[512];                                //读写缓冲区
static char s_arrStringBuff[MAX_STRING_LEN];                 //字符串转换缓冲区
```

在 ReadwriteInFlash.c 文件的“内部函数声明”区，声明了 2 个内部函数，如程序清单 5-8 所示。ReadProc 函数用于从 Flash 相应地址上读取相应长度的数据，WriteProc 函数向 Flash 相应地址写入相应长度的数据。

程序清单 5-8

```
static void ReadProc(u32 addr, u32 len);      //读取操作处理
static void WriteProc(u32 addr, u32 data);    //写入操作处理
```

在 ReadwriteInFlash.c 文件的“内部函数实现”区，首先实现了 ReadProc 函数，如程序清单 5-9 所示。

（1）第 7 行代码：由于本函数按字（32 位）进行数据读取，因此首先校验输入的地址参数是否为 4 的整数倍并位于内部 Flash 的地址范围内。

（3）第 16 行代码：通过 FlashReadWord 函数读取相应地址上的数据并将读取到的数据存入读取缓冲区。

（4）第 19 至 28 行代码：将读取的数据连同其对应的地址通过 LCD 及串口进行显示。

程序清单 5-9

```
1.  static void ReadProc(u32 addr, u32 len)
2.  {
3.      u32 i;      //循环变量
4.      u32 data; //读取到的数据
5.
6.      //校验地址范围和校验地址是否为 4 的整数倍
7.      if((0 == (addr % 4)) && (addr >= s_structGUIDev.beginAddr) && ((addr + (len - 1) * 4) <= s_structGUIDev.endAddr))
8.      {
9.
10.         //输出读取信息到终端和串口
11.         sprintf(s_arrStringBuff, "Read : 0x%08X - 0x%02X\r\n", addr, len);
12.         s_structGUIDev.showLine(s_arrStringBuff);
13.         printf("%s", s_arrStringBuff);
```

```
14.
15.         //从 Flash 中读取数据
16.         FlashReadWord(addr, s_arrBuff, len);
17.
18.         //打印到终端和串口上
19.         for(i = 0; i < len; i++)
20.         {
21.             //读取
22.             data = s_arrBuff[i];
23.
24.             //输出
25.             sprintf(s_arrStringBuff, "0x%08X: 0x%04X\r\n", addr + i * 4, data);
26.             s_structGUIDev.showLine(s_arrStringBuff);
27.             printf("%s", s_arrStringBuff);
28.         }
29.     }
30.     else
31.     {
32.         //无效地址
33.         s_structGUIDev.showLine("Read: Invalid address\r\n");
34.         printf("Read: Invalid address\r\n");
35.     }
36. }
```

在 ReadProc 函数实现区后为 WriteProc 函数的实现代码，如程序清单 5-10 所示。WriteProc 与 ReadProc 的函数体类似，校验地址后显示信息并通过 FlashWriteWord 函数将数据写入到 Flash 相应地址上。

程序清单 5-10

```
1.  static void WriteProc(u32 addr, u32 data)
2.  {
3.      //校验地址范围和校验地址是否为 4 的整数倍
4.      if((0 == (addr % 4)) && (addr >= s_structGUIDev.beginAddr) && (addr <= s_structGUIDev.endAddr))
5.      {
6.          //输出信息到终端和串口
7.          sprintf(s_arrStringBuff, "Write: 0x%08X - 0x%04X\r\n", addr, data);
8.          s_structGUIDev.showLine(s_arrStringBuff);
9.          printf("%s", s_arrStringBuff);
10.
11.         //写入内部 Flash
12.         FlashWriteWord(addr, &data, 1);
13.     }
14.     else
15.     {
16.         //无效地址
17.         s_structGUIDev.showLine("Write: Invalid address\r\n");
18.         printf("Write: Invalid address\r\n");
19.     }
20. }
```

在 ReadwriteInFlash.c 文件的“API 函数实现”区，首先实现 InitReadWriteInFlash 函数，

InitReadWriteInFlash 函数完成内部 Flash 模块的初始化，并实现了 GUI 界面与底层读写函数的联系。如程序清单 5-11 所示。

（1）第 4 至 13 行代码：赋值 GUI 结构体 s_structGUIDev 中相应的成员变量，如读写首地址、结束地址和读写回调函数。

（3）第 16 行代码：通过 InitGUI 函数初始化 UI 界面以及相应的界面参数。

（4）第 19 至 21 行代码：将内部 Flash 读写地址范围显示在 LCD 及计算机上。

程序清单 5-11

```
1.  void InitReadWriteInFlash(void)
2.  {
3.      //读写首地址
4.      s_structGUIDev.beginAddr = USER_FLASH_START_ADDR;
5.
6.      //读写结束地址
7.      s_structGUIDev.endAddr = USER_FLASH_ENDADDR - 4;
8.
9.      //设置写入回调函数
10.     s_structGUIDev.writeCallback = WriteProc;
11.
12.     //设置读取回调函数
13.     s_structGUIDev.readCallback = ReadProc;
14.
15.     //初始化 UI 界面设计
16.     InitGUI(&s_structGUIDev);
17.
18.     //打印地址范围到终端和串口
19.     sprintf(s_arrStringBuff, "Addr: 0x%08X - 0x%08X\r\n", s_structGUIDev.beginAddr, s_structGUIDev.endAddr);
20.     s_structGUIDev.showLine(s_arrStringBuff);
21.     printf("%s", s_arrStringBuff);
22. }
```

在 InitReadWriteInFlash 函数实现区后为 ReadWriteInFlashTask 函数的实现代码，该函数每隔 40ms 被调用一次，通过调用 GUITask 函数完成 UI 界面的扫描以及内部 Flash 模块读写任务的执行，如程序清单 5-12 所示。

程序清单 5-12

```
1.  void ReadWriteInFlashTask(void)
2.  {
3.      GUITask(); //GUI 任务
4.  }
```

5.3.3 Main.c 文件

在 Main.c 文件的 Proc2msTask 函数中，每 40ms 调用一次 ReadWriteInFlashTask 函数，实现读写内部 Flash 功能，如程序清单 5-13 所示。

程序清单 5-13

```
1.    static    void    Proc2msTask(void)
2.    {
3.        static u8 s_iCnt = 0;
4.        if(Get2msFlag())    //判断 2ms 标志状态
5.        {
6.            LEDFlicker(250);//调用闪烁函数
7.
8.            s_iCnt++;
9.            if(s_iCnt >= 20)
10.           {
11.               s_iCnt = 0;
12.               ReadWriteInFlashTask();
13.           }
14.
15.           Clr2msFlag();      //清除 2ms 标志
16.       }
17.   }
```

5.3.4 实验结果

代码编写完成并编译通过后，下载程序并进行复位。下载完成后，可以观察到开发板上的 LCD 显示如图 5-7 所示的 GUI 界面。

图 5-7　读写内部 Flash 实验 GUI 界面

点击“写入地址”一栏并输入“0807F000”，点击“写入数据”一栏并输入“F303”，点击“WRITE”按钮，此时 LCD 显示如图 5-8 所示，串口助手输出与 LCD 显示相同，表示数据写入成功。

图 5-8　数据写入

点击“读取地址”一栏同样输入“0807F000”，点击“读取长度”一栏并输入“1”，点击“READ”按钮，此时 LCD 显示如图 5-9 所示，串口助手输出与 LCD 显示相同，表示数据读取成功。

图 5-9　读取数据

本章任务

基于 GD32F3 苹果派开发板，编写程序实现密码解锁功能，例如：设置微控制器初始密码为 0x12345678，并将其写入内部 Flash（切勿写入到代码区），通过按下 KEY_1 按键模拟输入密码 0x12345678，通过按下 KEY_2 按键模拟输入密码 0x87654321，通过按下 KEY_3 按键进行密码匹配，如果密码正确，则在 LCD 上显示“Success！”，如果密码不正确，则显示“Failure！”。

本章习题

1．简述内部 Flash 的作用。

2．简述两种 Flash 的区别以及优缺点。

3．简述微控制器完成内部 Flash 写操作的流程。