STM32 与飞行控制

作者

郑扬珞 | 上海交通大学　肖智文 | 清华大学

wunschunreif@sjtu.edu.cn
xiaozw17@mails.tsinghua.edu.cn

2019–03–01

免责声明

本文件及其所含信息均按“原样”提供，作者不作任何明示或暗示的保证，包括但不限于使用本文件所含信息不会侵犯任何权利或任何适销性或特定用途适用性的暗示保证。

© Copyright 2019 郑扬珞 | 上海交通大学　肖智文 | 清华大学

本文件及其译文可以复制并提供给他人，评论或以其他方式解释本文件或协助本文件实施的衍生作品可以全部或部分制备、复制、出版和分发，不受任何形式的限制，但上述版权公告和本款应包括在所有此类副本和衍生作品。但是，不得以任何方式修改本文档本身，例如删除版权声明或引用作者或其他组织。
本文件以 GPL v3.0 协议发布。

目录

图片索引

表格索引

代码索引

缩略语

SoC System on a Chip

RM Reference Manual

HAL Hardware Abstraction Layer

GPIO General-Purpose Input/Output

NVIC Nested Vectored Interrupt Controller

EXTI EXTernal Interrupt/event controller

1. 绪论

1.1. 为什么要学习 STM32

自动化控制技术在飞行器制造的过程中日趋重要，姿态稳定、舵面控制等都要用到电子控制，而单片机以其重量轻、耗电少、控制逻辑灵活、易于学习应用等诸多优势，成为航模电子系统的首选控制器类型。在各类单片机中，STM32 具有很高的性价比：丰富的片上外设、完备的库函数、灵活的控制方式，以及远高出同类产品的时钟频率，使其成为在各个领域都有很强通用性的单片机产品，而它同样适合作为飞行控制器的核心，因此，我们有必要学习这款单片机，了解它的编程方式、具体功能，以便开发更为强大的控制系统。
（以上都是胡编的，我也不知道为什么要学习 STM32。。。）

1.2. 如何学习 STM32

作为一种应用技术，STM32 的最佳学习方式自然是实践，因此希望读者在阅读本文档的同时，能够多动手进行编程实践。虽然本文档的定位是一份涵盖作者认为编程中所有可能遇到的问题的事无巨细的指南，但不可否认仍有许多的问题只有亲自动手才能发现并解决，从而作为日后的经验积累下来。在这个过程中，我们也欢迎读者对本文档的内容做出斧正与完善。
此外，官方文档始终是学习 STM32 的权威帮助，作者在此推荐 STM32 的 Datasheet 和 Reference Manual，以及其标准外设库的文档，其中包含大量的示例代码以供学习参考。当然，遇到难以解决的问题时，求助网络可能是最快的解决方式，毕竟 STM32 在国内拥有相当大的群众基础，大多数问题都有网友进行总结了。

1.3. 预备知识

本文档认为读者已经基本掌握了下列预备知识：

- 计算机基本操作及其大致原理的了解

- C 语言程序设计（本文将采用 C11 标准），包括熟练掌握 2 阶以下的指针、函数指针、结构体、枚举等知识
- 电子技术，只需掌握简单的电路分析、一些数字电路的基本知识，但要求有一定的实践基础，例如能够正确地连接电路

1.4. 局限性

本文档编写的目的是阐述 STM32 单片机在一般的飞行控制中需要用到的外设的编程控制方法，对于其他的外设讨论很少。此外，本文档将着重介绍 STM32F1 系列的单片机，而并不会涉及性能更强的 STM32F4、STM32F7 等系列的单片机。

本章的最后，祝愿各位读者在今年上半年迅速掌握 STM32 这款单片机，软硬两开花！

Part I.

STM32 基础知识

这一部分主要讲解 STM32 的基本程序设计方法。内容包括:

- STM32 及其编程简介
- 开发环境搭建——基于 macOS 与 GCC
- GPIO 操作及其中断
- SysTick 的应用
- 未完待续。。。

完成这一部分的学习，读者就具备了利用 STM32 编写不太复杂的控制程序的能力，可以实现许多有趣的小项目。

2. STM32 及其编程简介

2.1. STM32 是什么

STM32 是指意法半导体（STMicroelectronics）制造的一系列 32 位 ARM 架构单片机，这些单片机依据它们使用的核心不同，分为许多系列，例如本文档主要介绍的 F1 系列采用的是 Cortex-M3 内核[1]。作为单片机，STM32 不仅包含了 CPU 核，还集成了 Flash 程序存储器、sRAM 内存，以及诸如 GPIO 控制器的丰富外设，构成了一个片上系统（System on a Chip (SoC) ）。此外，STM32F1 系列单片机的时钟频率可以高达 72MHz，远超 Arduino UNO 的时钟频率，因此可以实现一些复杂的、繁重的控制任务，以及一些常用的算法。

本文档主要介绍的是 STM32F103C8 单片机，其中，这个命名包含的信息有：

- STM32 - 表示 ST 生产的 32 位单片机
- F103 - 该单片机的系列
- C - 引脚数为 48 个
- 8 - Flash 大小为 64KiB[2]

此外，它的内存容量为 20KiB，是一款“中密度”单片机产品。

关于 STM32F103C8 的更详细的数据可以从它的 Datasheet[3]中查到，这里不再继续对其进行介绍。

2.2. STM32 程序怎么写

虽然我们认为这篇文档的读者以及掌握了基本的 C 语言程序设计，但是，在一个高度抽象的计算机上编程，与这里为单片机编程有着相当大的区别，所以，我们有必要从本质出发认识 STM32 程序的设计方法。

[1] https://en.wikipedia.org/wiki/STM32
[2] 1 KiB = 1024 Bytes
[3] https://www.st.com/resource/en/datasheet/stm32f103c8.pdf

2.2.1. 程序的本质是什么

程序的本质，当然是计算，毕竟程序是为“计算机”设计的。计算，可以认为是程序唯一的本质，CPU 的所有行为不过是从存储介质中取得运算数，按程序对其计算，再将结果写入规定的存储介质中。可是这样，单片机是如何完成各种强大的控制功能的呢？这些功能，实际上是通过逻辑电路来实现的，CPU 将控制这些逻辑电路的参数写入规定好的寄存器中，就可以告诉相应的逻辑电路该产生什么行为，除非遭遇了宇宙射线什么的，否则这些逻辑器件总是会乖乖听话。这就是单片机控制功能的来源。不如举个例子，查阅 STM32 的参考手册[4]（Reference Manual (RM)），可以知道 GPIO 这个外设有一个寄存器叫做“ODR (Port output data register)”，可以控制 GPIO 的输出电平，这个寄存器每个二进制位的意义如下图：

9.2.4 Port output data register (GPIOx_ODR) (x=A..G)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy:** Port output data (y= 0 .. 15)

These bits can be read and written by software and can be accessed in Word mode only.

Note: For atomic bit set/reset, the ODR bits can be individually set and cleared by writing to the GPIOx_BSRR register (x = A .. G).

图 2.1.: ODR 寄存器

假设 GPIOA 已经正确初始化，那么只要我们向 GPIOA 的 ODR 寄存器写入 0x01，就可以控制 PA0 引脚输出高电平。为此，我们的程序可能长得像下面这样：

```
1 GPIOA->ODR = 0x01;
```

代码清单 2.1: 控制 PA0 输出高电平

实际上，完成这个操作的确这么简单，这一行程序的功能也没有超出“计算”的范围，因为它就是向一个指定的位置写入了一个预先算好的数据。

[4]https://www.st.com/resource/en/reference_manual/cd00171190.pdf

所以我们说，程序的本质是计算。

2.2.2. 我对象在哪儿

在 C 语言中，一切对象皆是地址。

上面的代码中，并没有明显地表现出我们要操作的对象，GPIOA 的 ODR 寄存器在哪里，事实上，仔细看看图2.1，可以知道 ODR 寄存器的相对地址偏移（address offset）为 0x0C，也就是说，在 GPIOA 的基地址上加上 0x0C，就是 GPIOA 的 ODR 寄存器的绝对地址。有了地址，我们就可以对为所欲为了。再次查阅RM，知道 GPIOA 的基地址为 0x40010800，所以控制 PA0 输出高电平还可以写成下面的形式：

```
1 *(uint32_t*)(0x40010800 + 0x0C) = (uint32_t)0x01;
```

代码清单 2.2: 控制 PA0 输出高电平

这行代码首先将地址 0x40010800 + 0x0C 转换为一个指向 32 位无符号整形数的指针，然后对其解引用，并写入 0x01 这个值，其功能与上一节中的代码完全一样。并且，只要单片机的初始化工作已经完成，这一行代码无需引入外部头文件即可编译通过。

当然，这样写代码还不如直接写汇编，所以我们要充分利用 C 语言的抽象性，利用结构体对外设进行建模。同样以 GPIO 这个外设为例，查阅RM知道它的寄存器有下面这几个：

表 2.1.: GPIO 寄存器

寄存器名称	地址偏移	简要描述
CRL	0x00	控制寄存器，控制 0 ～ 7 引脚
CRH	0x04	控制寄存器，控制 8 ～ 15 引脚
IDR	0x08	引脚输入寄存器
ODR	0x0C	引脚输出寄存器
BSRR	0x10	引脚置位（置 1）寄存器
BRR	0x14	引脚复位（清 0）寄存器
LCKR	0x18	引脚锁存寄存器

由于 STM32 是 32 位的单片机，因此其中的寄存器大多是 32 位的，即每一个寄存器占据的字节数是 0x04，这与上表中的地址偏移之间的差值是一致的，从而可以发现这些寄存器的地址是连续分布的。这样，我们可以写出下面的结构体对 GPIO 的这几个寄存器进行抽象：

```
typedef struct GPIO_TypeDef {
    volatile uint32_t CRL;
    volatile uint32_t CRH;
    volatile uint32_t IDR;
    volatile uint32_t ODR;
    volatile uint32_t BSRR;
    volatile uint32_t BRR;
    volatile uint32_t LCKR;
} GPIO_TypeDef;
```

代码清单 2.3: GPIO 结构体

需要注意的是，这个结构体中各个成员数据的相对位置不能改变，类型也必须是 32 位大小的整形（最好是无符号的，以免运算时发生莫名其妙的符号位扩展），也不能增减成员，只有这样，才能正确地抽象 GPIO 的这些寄存器，与相应的地址空间对应。成员声明前面的 volatile 表明编译器不得对这个成员做出优化，意思是编译出来的程序必须老老实实的按照地址读写数据，而不能偷懒将它临时拷贝到 CPU 的寄存器中，这是因为这些 GPIO 寄存器并不是只有 CPU 才能读写，而电路本身就会更改它们的值。

利用这个结构体，我们就可以把从 0x40010800 开始的一段地址空间解释为 GPIOA 这个具体的外设，并对它进行操作：

```
GPIO_TypeDef * GPIOA = (GPIO_TypeDef *)(0x40010800);
GPIOA->ODR = 0x01;
```

代码清单 2.4: GPIO 结构体应用

这实际上解释了上一节的程序到底将 0x01 这个数据写到哪里了这个问题。

再强调一遍，C 语言里，一切对象皆是地址，这个概念在 STM32 编程中尤为重要。所以下面，我们来欣赏一下 STM32 的地址空间映射[5]。

[5]截自 STM32F103C8 Datasheet

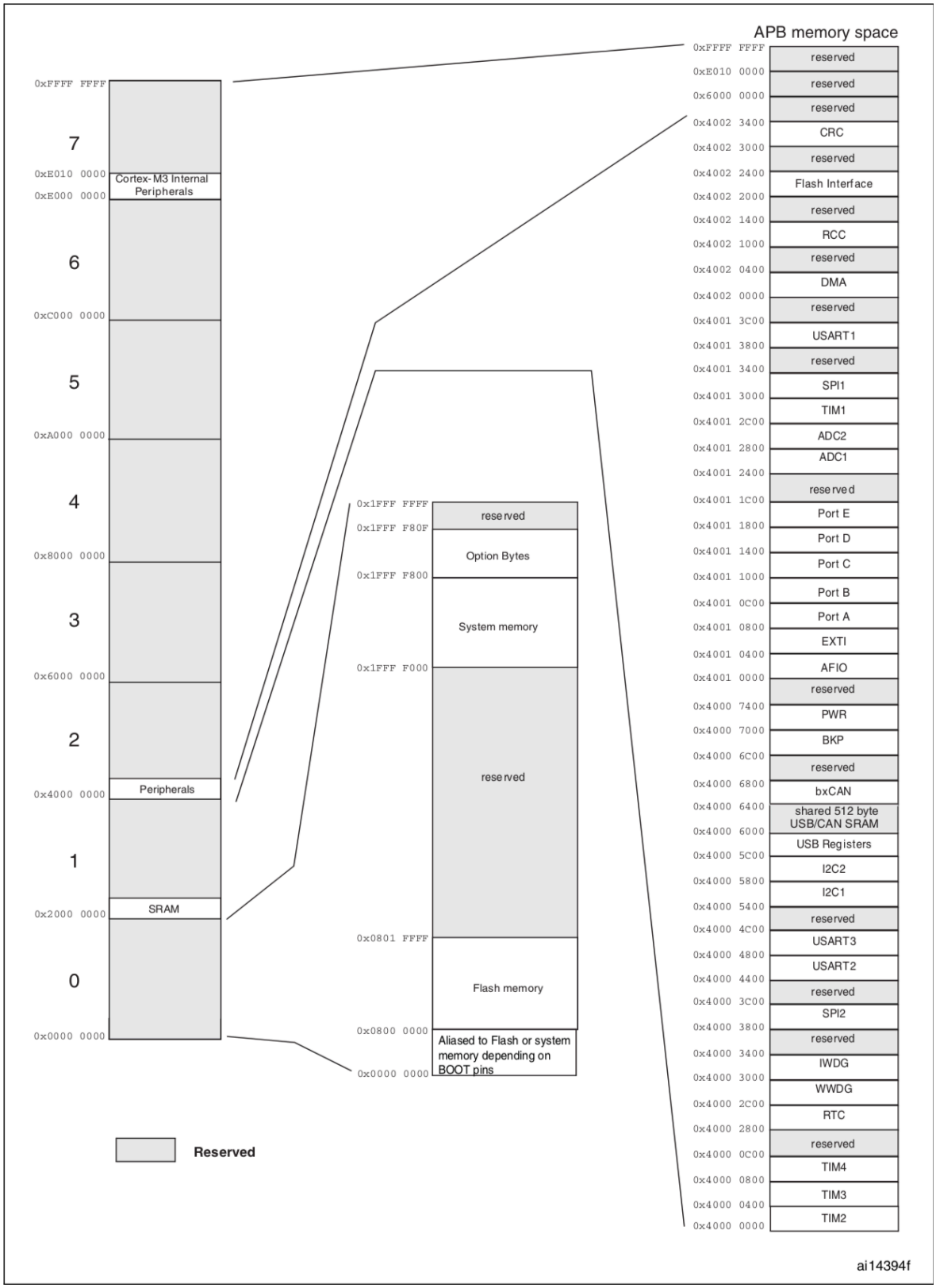

图 2.2.: STM32 地址空间映射图

2.2.3. STM32 标准外设库

我们不造轮子，但要知道轮子怎么造。

按照上一节的方法，对每一个外设逐一建模，再为每一个具体外设定义结构体变量，是一个相当庞大的工程，实现起来可能我们今年上半年就不用干别的了，而且就算实现了，用寄存器的方式操作外设难免会遇到许多玄学问题。因此，ST 官方推出了标准外设库（Standard Peripherals Firmware Library），本文档采用的版本是 3.5.0。目前，ST 似乎把更多的精力转向了推广其HAL（硬件抽象层）库，不过两者大同小异，而且功能都是完备的，所以不必太纠结使用哪个库，学会了其中一个就很容易触类旁通。

标准外设库用结构体、枚举类型及其宏定义的方式抽象出了 STM32 所有的外设资源，对于各种外设的基本操作，也提供了丰富的函数接口对其进行封装。使用标准外设库可以简化 STM32 程序的编写过程，而且其代码都是经过了专业的优化，具有很强的编译期查错能力，对于内存的占用也非常小，所以我们完全可以放心大胆地使用。

使用标准外设库，控制 PA0 输出高电平的语句就变得很直观：

```
1 GPIO_SetBits(GPIOA, GPIO_Pin_0);
```

代码清单 2.5: 控制 PA0 输出高电平

标准外设库的命名方式一般为“外设 _ 功能”的格式，所以对于代码补全非常友好，使用一个具有较强代码补全能力的编辑器（例如 VS Code），多数情况下只需要打几个字母就能完成整行的输入。

既然官方已经为我们造好了这么方便的轮子，我们编写 STM32 的程序就可以处处应用标准外设库。本文档这一部分的介绍也将主要围绕着标准外设库的使用而展开。

2.3. 思考与练习

1. STM32F103C8 的最高时钟频率是多少，如何理解这一数据?
2. STM32F103C8 的内存大小是多少？结合它的时钟频率，说明它适合执行怎么样的算法？举例说明不适合它执行的算法。当然，要考虑数据规模。
3. 仿照代码清单2.3，参考RM，为外设 SPI 编写结构体进行抽象，并定义一个变

量对应 SPI1。完成后，对比标准外设库中的实现，想想为什么不一样（如果一样，那您就是巨佬）。

```
1  //其中的__IO是volatile的一个宏
2  typedef struct
3  {
4      __IO uint16_t CR1;
5      uint16_t  RESERVED0;
6      __IO uint16_t CR2;
7      uint16_t  RESERVED1;
8      __IO uint16_t SR;
9      uint16_t  RESERVED2;
10     __IO uint16_t DR;
11     uint16_t  RESERVED3;
12     __IO uint16_t CRCPR;
13     uint16_t  RESERVED4;
14     __IO uint16_t RXCRCR;
15     uint16_t  RESERVED5;
16     __IO uint16_t TXCRCR;
17     uint16_t  RESERVED6;
18     __IO uint16_t I2SCFGR;
19     uint16_t  RESERVED7;
20     __IO uint16_t I2SPR;
21     uint16_t  RESERVED8;
22 } SPI_TypeDef;
```

代码清单 2.6: 标准外设库中的 SPI 结构体

4. 标准外设库有什么好处？为什么不直接用寄存器编程？

5. 简述操作 STM32 外设的本质。

3. 开发环境搭建与工程模板

3.1. 开发环境概述

编写 STM32 程序的过程大致可以分为 3 个阶段：编写代码、编译链接、下载程序。其中，编写代码是使用 C 语言，结合标准外设库，编制源代码的阶段，编译链接是利用编译器，将源代码编译成 STM32 平台的可执行文件，而下载程序是利用 ST 提供的接口驱动，将电脑上的可执行文件传输至 STM32 的 Flash 程序存储器中的过程。下面针对这 3 个阶段，分别介绍本文档推荐的工具。

编写代码 本文档推荐的代码编辑工具是微软公司推出的开源跨平台编辑器 VS Code。这款编辑器拥有丰富的插件，配合这些插件可以实现对 C 语言很好的支持，代码补全的能力相当高，而且可以一边编写代码一边查错。此外，利用它的集成终端功能，可以在编辑器内执行命令行命令，完成编译和下载。

编译链接 本文档推荐的编译器是 arm-none-eabi-gcc，是由 GCC 提供的专为 ARM 平台编译程序的交叉编译工具链，历史悠久，性能可靠，最重要的一点是这个编译器完全免费而且开源，使用它不必担心遭到起诉，是彻头彻尾的正版软件。相比之下，Keil、IAR 等软件虽然也是 STM32 程序编写和编译的流行软件，但它们正版软件的价格高昂，而且打击盗版的力度较大，因此不推荐使用。

下载程序 本文档推荐的下载程序的接口和软件为 ST-Link，这是 ST 公司为 STM32 设计的下载与调试接口，只需要 4 根线就可以连接。此外，使用 J-Link 也可以完成程序下载的功能，但 J-Link 接口需要 20 线，且下载器价格较高，因此并不推荐使用。目前，ST-Link 也是 ST 公司官方推荐的连接方式。

3.2. 软件的安装

VS Code 的安装方法较为简单，只需在官网上下载安装，之后安装几个与 C/C++ 有关的插件即可。

arm-none-eabi-gcc 可以在 https://github.com/gnu-mcu-eclipse/arm-none-eabi-gcc 中找到下载链接，可以下载对应操作系统编译好的版本，也可以下载源码后自行编译。如果下载的是编译好的版本，那么在解压后得到的目录下的 bin 文件夹中就会看到 arm-none-eabi-gcc 这个可执行文件。方便起见，可以将这个目录添加到系统的环境变量 PATH 中。对于 macOS 用户（部分 Linux 系统用户也适用），可以在终端执行

```
nano ~/.bash_profile
```

然后在文件末尾添加：

```
export PATH="/path/to/gcc/bin:$PATH"
```

保存退出后，再执行命令：

```
source ~/.bash_profile
```

即可完成环境变量的改变。

st-link 软件可以直接利用系统的包管理器安装，对于使用 homebrew 的 macOS 用户，只需在终端中执行

```
brew install st-link
```

稍等片刻就完成了 st-link 软件的安装。

Windows 系统下相关软件安装概述 对于 Windows 系统，代码编辑器 VS code 和交叉编译器 arm-none-eabi-gcc 都有相应的版本，只需下载安装即可，而对于下载软件 st-link，也可用 Windows 系统下功能相同的软件替代。当然，我们推荐使用正版软件。

3.3. 建立工程模版

首先,假定我们已经下载了 STM32F1 系列的标准外设库,并且位于 STDPeriphLib/目录下。接下来，我们需要新建一个目录，作为工程模板的根目录，不妨设这个目录为 C8Template/。在 C8Template/下，我们需要新建下列文件夹和文件：

表 3.1.: C8Template/下的内容

项目名称	类型	简要描述
CMSIS/	目录	与 Cortex-M3 内核有关的库文件所在目录
Startup/	目录	单片机启动代码所在目录
Lib/	目录	外设库所在目录
User/	目录	开发者编写的代码所在目录
makefile	文本文件	构建工程的脚本
stm32_flash.ld	文本文件	gcc 需要的链接器脚本

下面逐一介绍各个目录和文件的内容。

3.3.1. CMSIS/

这个目录下放置的是与单片机核心有关的库文件，需要从标准外设库的 STDPeriphLib/Libraries/CMSIS/CM3/中拷贝以下 5 个文件：

- CoreSupport/core_cm3.c
- CoreSupport/core_cm3.h
- DeviceSupport/ST/STM32F10x/system_stm32f10x.c
- DeviceSupport/ST/STM32F10x/system_stm32f10x.h
- DeviceSupport/ST/STM32F10x/stm32f10x.h

3.3.2. Startup/

这个目录下放置的是单片机的启动代码，需要从标准外设库的 STDPeriphLib/Libraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x/startup/gcc_ride7/ 目录下拷贝所有的文件。

3.3.3. Lib/

这个目录下放置的是所有外设的库文件，需要拷贝标准外设库的 STDPeriphLib/Libraries/STM32F10x_StdPeriph_Driver/下的 inc/和 src/两个目录，此外，还需要创建一个名为 makefile 的文本文件，其内容如下:

```
CC = arm-none-eabi-gcc
AR = arm-none-eabi-ar
vpath %.c src ../CMSIS
vpath %.h inc
CFLAGS  = -g -O2 -Wall
CFLAGS += -mlittle-endian -mthumb -mcpu=cortex-m3 -mthumb-interwork
CFLAGS += -ffreestanding -nostdlib
CFLAGS += -Iinc -I../CMSIS -I../User
CFLAGS += -DUSE_STDPERIPH_DRIVER
CFLAGS += -DSTM32F10X_MD

SRCS = misc.c stm32f10x_adc.c stm32f10x_bkp.c stm32f10x_can.c \
            stm32f10x_cec.c stm32f10x_crc.c stm32f10x_dac.c \
            stm32f10x_dbgmcu.c stm32f10x_dma.c stm32f10x_exti.c \
            stm32f10x_flash.c stm32f10x_fsmc.c stm32f10x_gpio.c \
            stm32f10x_i2c.c stm32f10x_iwdg.c stm32f10x_pwr.c \
            stm32f10x_rcc.c stm32f10x_rtc.c stm32f10x_sdio.c \
            stm32f10x_spi.c stm32f10x_tim.c stm32f10x_usart.c \
            stm32f10x_wwdg.c
SRCS += core_cm3.c system_stm32f10x.c

OBJS = $(SRCS:.c=.o)

all: libstm32periLib.a

%.o : %.c
$(CC) $(CFLAGS) -c -o $@ $^

libstm32periLib.a : $(OBJS)
$(AR) -r $@ $(OBJS)

.PHONY: clean
clean:
rm -f $(OBJS)
```

代码清单 3.1: Lib/makefile

虽然你可能不想一个字一个字的敲这一段脚本，但是这也是作者纯手写的，所以。。。

3.3.4. User/

这个目录下放置的是开发者可以改动、添加的代码文件，一般来说，开发 STM32 程序需要建立和编辑的代码都在这个目录下放置。
初始状态下，我们需要从标准外设库中拷贝以下 4 个文件：

- STDPeriphLib/Project/STM32F10x_StdPeriph_Template/main.c
- STDPeriphLib/Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
- STDPeriphLib/Project/STM32F10x_StdPeriph_Template/stm32f10x_it.c
- STDPeriphLib/Project/STM32F10x_StdPeriph_Template/stm32f10x_it.h

3.3.5. stm32_flash.ld

这个文件是链接器脚本，其模板如下：

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = 0x20005000;    /* end of RAM */

_Min_Heap_Size = 0;      /* required amount of heap  */
_Min_Stack_Size = 0x400; /* required amount of stack */

/* Memories definition */
MEMORY
{
  RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 20K
  ROM (rx)       : ORIGIN = 0x8000000, LENGTH = 64K
}

/* Sections */
SECTIONS
{
  /* The startup code into ROM memory */
  .isr_vector :
```

```
22  {
23    . = ALIGN(4);
24    KEEP(*(.isr_vector)) /* Startup code */
25    . = ALIGN(4);
26  } >ROM
27
28  /* The program code and other data into ROM memory */
29  .text :
30  {
31    . = ALIGN(4);
32    *(.text)           /* .text sections (code) */
33    *(.text*)          /* .text* sections (code) */
34    *(.glue_7)         /* glue arm to thumb code */
35    *(.glue_7t)        /* glue thumb to arm code */
36    *(.eh_frame)
37
38    KEEP (*(.init))
39    KEEP (*(.fini))
40
41    . = ALIGN(4);
42    _etext = .;        /* define a global symbols at end of code */
43    _exit = .;
44  } >ROM
45
46  /* Constant data into ROM memory*/
47  .rodata :
48  {
49    . = ALIGN(4);
50    *(.rodata)         /* .rodata sections (constants, strings, etc.) */
51    *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
52    . = ALIGN(4);
53  } >ROM
54
55  .ARM.extab   : {
56    . = ALIGN(4);
57    *(.ARM.extab* .gnu.linkonce.armextab.*)
58    . = ALIGN(4);
59  } >ROM
60
61  .ARM : {
62    . = ALIGN(4);
63    __exidx_start = .;
64    *(.ARM.exidx*)
65    __exidx_end = .;
```

```
66      . = ALIGN(4);
67    } >ROM
68
69    .preinit_array     :
70    {
71      . = ALIGN(4);
72      PROVIDE_HIDDEN (__preinit_array_start = .);
73      KEEP (*(.preinit_array*))
74      PROVIDE_HIDDEN (__preinit_array_end = .);
75      . = ALIGN(4);
76    } >ROM
77
78    .init_array :
79    {
80      . = ALIGN(4);
81      PROVIDE_HIDDEN (__init_array_start = .);
82      KEEP (*(SORT(.init_array.*)))
83      KEEP (*(.init_array*))
84      PROVIDE_HIDDEN (__init_array_end = .);
85      . = ALIGN(4);
86    } >ROM
87
88    .fini_array :
89    {
90      . = ALIGN(4);
91      PROVIDE_HIDDEN (__fini_array_start = .);
92      KEEP (*(SORT(.fini_array.*)))
93      KEEP (*(.fini_array*))
94      PROVIDE_HIDDEN (__fini_array_end = .);
95      . = ALIGN(4);
96    } >ROM
97
98    /* Used by the startup to initialize data */
99    _sidata = LOADADDR(.data);
100
101   /* Initialized data sections into RAM memory */
102   .data :
103   {
104     . = ALIGN(4);
105     _sdata = .;        /* create a global symbol at data start */
106     *(.data)           /* .data sections */
107     *(.data*)          /* .data* sections */
108
109     . = ALIGN(4);
```

```
110     _edata = .;        /* define a global symbol at data end */
111   } >RAM AT> ROM
112
113   /* Uninitialized data section into RAM memory */
114   . = ALIGN(4);
115   .bss :
116   {
117     /* This is used by the startup in order to initialize the .bss secion
          ↪  */
118     _sbss = .;         /* define a global symbol at bss start */
119     __bss_start__ = _sbss;
120     *(.bss)
121     *(.bss*)
122     *(COMMON)
123
124     . = ALIGN(4);
125     _ebss = .;         /* define a global symbol at bss end */
126     __bss_end__ = _ebss;
127   } >RAM
128
129   /* User_heap_stack section, check that there is enough RAM left */
130   ._user_heap_stack :
131   {
132     . = ALIGN(8);
133     PROVIDE ( end = . );
134     PROVIDE ( _end = . );
135     . = . + _Min_Heap_Size;
136     . = . + _Min_Stack_Size;
137     . = ALIGN(8);
138   } >RAM
139
140   /* Remove information from the compiler libraries */
141   /DISCARD/ :
142   {
143     libc.a ( * )
144     libm.a ( * )
145     libgcc.a ( * )
146   }
147   .ARM.attributes 0 : { *(.ARM.attributes) }
148 }
```

代码清单 3.2: stm32_flash.ld

这个文件中绝大部分不能更改，但是针对不同型号的单片机，有以下几个部分可以改动：

- 第 13 行的 LENGTH 值，需要与单片机 RAM 大小相一致
- 第 14 行的 LENGTH 值，需要与单片机的 FLASH 大小相一致
- 第 5 行的 _estack 值，需要等于第 13 行 ORIGIN 和 LENGTH 相加的结果

3.3.6. makefile

这个文件是构建工程的脚本，其模板如下：

```
1  SRCS = main.c stm32f10x_it.c startup_stm32f10x_md.s
2  HEADERS = stm32f10x_it.h stm32f10x_conf.x
3
4  ########################### USER FILES BELOW ############################
5  SRCS +=
6  HEADERS +=
7  ########################### USER FILES ABOVE ############################
8
9  PROJ_NAME = main
10
11 vpath %.c User
12 vpath %.a Lib
13 vpath %.s Startup
14 vpath %.h User
15
16 CC=arm-none-eabi-gcc
17 OBJCOPY=arm-none-eabi-objcopy
18
19 CFLAGS  = -g -O2 -Wall -Tstm32_flash.ld
20 CFLAGS += -mlittle-endian -mthumb -mcpu=cortex-m3 -mthumb-interwork
21 CFLAGS += -DUSE_STDPERIPH_DRIVER
22 CFLAGS += -DSTM32F10X_MD
23
24 CFLAGS += -ILib/inc -ICMSIS -IUser -lc
25
26 OBJS = $(SRCS:.c=.o)
27
28 .PHONY: lib proj
29
30 all: lib proj
```

```
31
32 lib:
33     $(MAKE) -C Lib
34
35 proj: $(PROJ_NAME).elf
36
37 $(PROJ_NAME).elf: $(SRCS)
38     $(CC) $(CFLAGS) $^ -o $@ -LLib -lstm32periLib
39     $(OBJCOPY) -O ihex $(PROJ_NAME).elf $(PROJ_NAME).hex
40     $(OBJCOPY) -O binary $(PROJ_NAME).elf $(PROJ_NAME).bin
41
42 SRCS : HEADERS
43 HEADERS:
44
45 .PHONY: clean
46
47 clean:
48     rm -f *.o *.elf *.hex *.bin
```

代码清单 3.3: makefile

在编写代码的同时，这个文件中也有许多地方需要同时进行更改:

- 每添加一个 C 源文件，需要在第 5 行正确地加上这个文件的文件名（不包含路径，但包含后缀名），以空格隔开，必要时可以使用反斜线“\”作为续行符
- 每添加一个 C 头文件，需要在第 6 行正确地加上这个文件的文件名
- 每创建一个将会包含 C 源文件的文件夹，需要在第 11 行正确地加上这个目录的相对路径
- 每创建一个将会包含 C 头文件的文件夹，需要在第 14 行正确地加上这个目录的相对路径，并在第 24 行加上“-I”+ 目录的相对路径
- 当更换单片机的型号时，需要在第 22 行做出相应的调整，例如，对于低密度产品，要将“MD”替换为“LD”，对于高密度产品则需要替换为“HD”

上面的 5 点注意事项尤为重要，需要牢记在心。不过，由于这些改动较为机械，本文档在接下来的叙述中将不再做出强调，而默认读者会做出正确地操作。
完成上面的步骤后，我们的工程模板就创建好了，以后编写一个项目时，可以直接拷贝一份工程模板作为起始。当然，不建议直接在工程模板中编写代码。

3.4. 工程模板的编译与程序下载

我们的工程模板实际上是一个完备的程序，无需做出任何更改就可以编译并下载到单片机上，这一过程可以作为我们平台搭建与工程模板创建正确性的检验环节。

3.4.1. 编译项目

首先我们需要打开终端并将当前路径切换到工程模板的根目录下，如果使用 VS Code 编辑器，可以打开工程模板文件夹后直接调出集成终端，无需再切换路径。在终端中，输入“make”命令，按下回车。如果没有提示任何错误信息，并且工程模板中多出来了“main.bin”等文件，说明编译过程已经成功完成了。这里得到的“main.bin”就是我们需要的二进制程序。首次执行“make”命令可能会花费较长时间，不过执行过一次之后，此后的编译过程就会变得很快。这是因为 make 只会对上一次编译以来发生过改动的文件重新编译。

3.4.2. 下载程序

在 macOS 上，下载程序需要用到 st-link 软件中的 st-flash 程序，同样打开终端并切换到工程模板根目录下，输入下面的命令：

```
st-flash write main.bin 0x8000000
```

如果已经将 STM32 通过 ST Link 连接上了计算机，并正确供电，那么此时程序就会写入单片机的 Flash 内部，只要执行上述命令时没有报错，按下单片机的复位键就可以运行程序了。当然，对于工程模板中的程序来说，并不会出现任何现象。

3.4.3. 清理项目

由于执行 make 指令会产生一些没有作用的文件，如果我们想要清理这些文件，可以执行“make clean”指令。不过，这样做以后，下次编译就需要重新编译全部文件了。

3.5. 思考与练习

1. 在你的计算机上搭建开发环境，创建工程模板，并完成首次编译以及程序下载。
2. 熟读并背诵更改 makefile 的注意事项。
3. 考察3.4.2中的命令，其中 0x8000000 是指程序的起始地址，结合图2.2说明为什么是这个值。

4. GPIO 操作及其中断

4.1. GPIO 简介

General-Purpose Input/Output (GPIO)是指通用目的输入输出设备。在 STM32 上，它以几组引脚的形式引出，可以连接其他电路组件，例如 LED 灯，此外，许多引脚有特定的复用功能，例如可以作为串口通信的 Rx 和 Tx 引脚，连接串口设备，例如 GPS 等。GPIO 是 STM32 上最基本的外设，因此在这里我们首先来学习 GPIO 的一些基本应用，以此引出操作 STM32 外设的一般编程流程。

STM32 的 GPIO 是一个比较高级的 IO 外设，不同于，每个引脚都支持几种不同的模式，适用于不同的应用场景，有的引脚还支持高达 5V 的输入电压。GPIO 的基本电路结构如下图[1]。

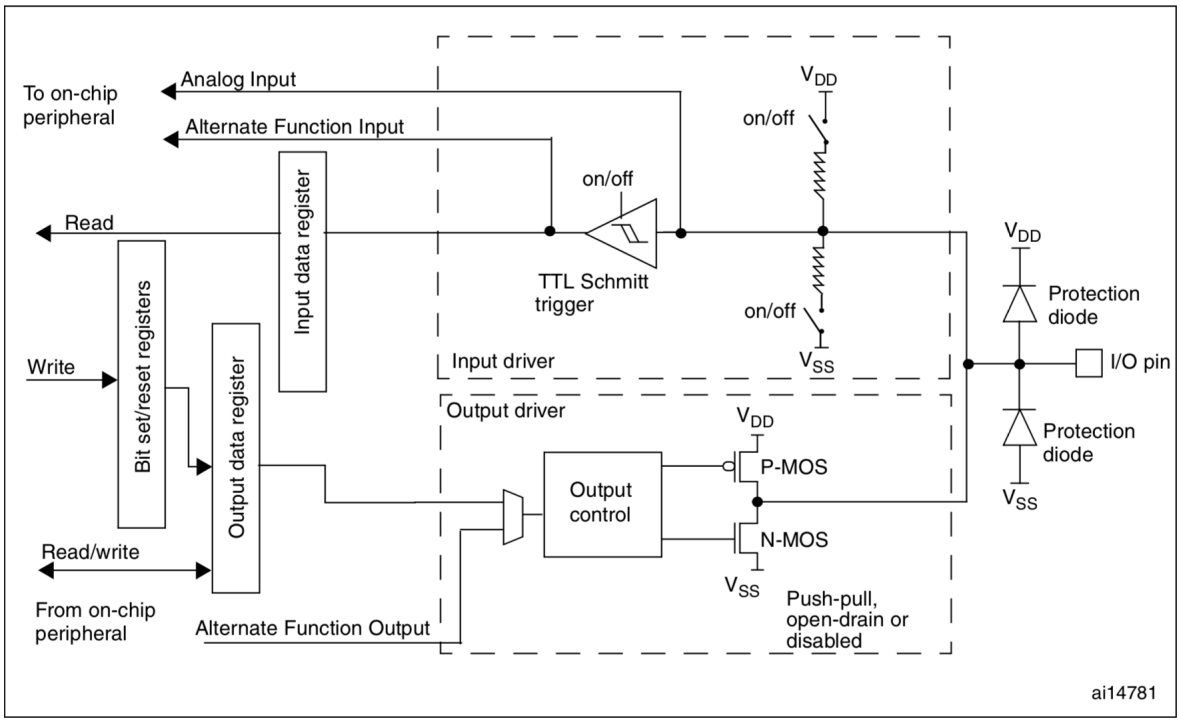

图 4.1.: GPIO 基本结构

[1]截自RM

GPIO 支持的模式见下表:

表 4.1.: GPIO 模式

类型	模式名称	简要描述
输出模式	推挽输出	电流较大的输出模式
输出模式	漏极开路输出	可以双向通信的输出模式，一般用于模拟通信协议（例如 IIC）
输出模式	复用推挽输出	作为复用功能时的推挽输出模式
输出模式	复用漏极开路输出	作为复用功能时的开漏输出模式
输入模式	浮空输入	完全悬空的输入模式
输入模式	上拉输入	内部接入上拉电阻的输入模式
输入模式	下拉输入	内部接入下拉电阻的输入模式
输入模式	模拟输入	输入 ADC 时使用的模式

在使用的时候，应当仔细分析应用场景，找出适合的模式。接下来，我们通过几个简单的例子，了解 GPIO 常用的操作以及对应的库函数用法。

4.2. GPIO 常用操作

在 STM32 中，使用一个外设必须首先完成初始化操作，包括初始化对应的外设模块，打开对应的时钟信号，某些外设还要求产生软件启动指令才能正常工作。我们假设目前的场景下，单片机连接的外部元件如下表所示:

表 4.2.: 单片机引脚定义

引脚	连接的元件	需要使用的模式
PA0	LED 灯（另一侧通过限流电阻接 3.3V）	推挽输出
PA1	LED 灯（另一侧通过限流电阻接 3.3V）	推挽输出
PA2	按钮（另一侧接 3.3V）	下拉输入
PA3	按钮（另一侧接 0V）	上拉输入

在这个例子当中，我们将依次完成 LED 闪烁、流水灯、按钮控制 LED 灯的实验。下面，让我们看一看具体的代码实现。
这个例子中，我们需要在工程文件夹下的 User/文件夹下新建 GPIO.c 和 GPIO.h 两个文件，以模块的形式向外部提供接口。

4.2.1. 预备工作

我们首先来填入 GPIO.h 这个头文件：

```
1  #ifndef __GPIO_H__
2  #define __GPIO_H__
3
4  #include "stm32f10x.h"
5  #include "stm32f10x_conf.h"
6
7  void PinsInit();
8  void LED1Operation(u8 on);
9  void LED2Operation(u8 on);
10 u8 Button1Pressing();
11 u8 Button2Pressing();
12
13 #endif
```

代码清单 4.1: GPIO.h

头文件中首先引入了两个外设库提供的头文件，利用这两个头文件就可以访问外设库中的全部函数。之后，我们声明了 5 个函数，第一个的功能是初始化用到的引脚，此后的两个函数操作对应的 LED 灯，最后两个函数用于检查两个按钮是否按下。这些函数的实现都放在 GPIO.c 文件中。要注意的是，在 C 语言中，并没有 bool 等布尔类型，因此我们需要使用 u8 代替，顾名思义，u8 表示这个类型是无符号（unsigned）8 位整型。在我们编写 GPIO.c 中的代码时，需要包含这个头文件。如果读者学习过 C 语言的模块化设计，应该很熟悉这种操作。

4.2.2. GPIO 的初始化

下面这个函数完成用到的 GPIO 的初始化，位于 GPIO.c 文件中。

```
1  void PinsInit() {
2      GPIO_InitTypeDef gpioInitStruct;
3
4      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
5
6      gpioInitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
7      gpioInitStruct.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
8      gpioInitStruct.GPIO_Speed = GPIO_Speed_50MHz;
9      GPIO_Init(GPIOA, &gpioInitStruct);
10
11     gpioInitStruct.GPIO_Mode = GPIO_Mode_IPD;
12     gpioInitStruct.GPIO_Pin = GPIO_Pin_2;
13     GPIO_Init(GPIOA, &gpioInitStruct);
14
15     gpioInitStruct.GPIO_Mode = GPIO_Mode_IPU;
16     gpioInitStruct.GPIO_Pin = GPIO_Pin_3;
17     GPIO_Init(GPIOA, &gpioInitStruct);
18 }
```

代码清单 4.2: GPIO 初始化

函数的第 2 行定义了一个 GPIO 初始化结构体，以备之后使用。

第 4 行调用库函数打开了 GPIOA 的时钟。读者应当注意到其中 APB2Periph 这个短语，这表示 GPIOA 这个外设是属于 APB2 时钟线上的外设。这个结论来自于 Datasheet，如图4.2所示。

第 6 ～ 9 行设置 PA0 和 PA1 的模式为推挽输出（Push-Pull) 模式，最大输出频率为 50MHz。这一步操作用到了 GPIO_Init(GPIO_TypeDef*, GPIO_InitTypeDef*) 这个函数，它的第一个参数是 GPIO 端口，第二个参数是指向 GPIO 初始化结构体的指针，因此我们不需要在 GPIO 初始化结构体中指明具体的端口，只需配置引脚、模式以及速率。从代码中可以看出，如果要同时配置多个引脚，只需用“按位或”运算符（“|”）连接即可，当然它们必须属于同一个 GPIO 端口。

第 11 ～ 17 行分别配置 PA2 和 PA3 为下拉输入（input pull-down）和上拉输入（input pull-up）模式。由于输入模式不需要指定最大速率，因此 GPIO_Speed 成员不必填写。

总的来说，初始化 GPIO 的步骤是：首先开启所需 GPIO 端口的时钟，接着填写 GPIO 初始化结构体，并用这个结构体初始化具体的引脚。在 STM32 中，初始化大部分外

设的步骤也都与此类似，即首先要使能对应外设的时钟，之后再使情况具体初始化外设，对于部分外设，还需要一个软件使能的过程。

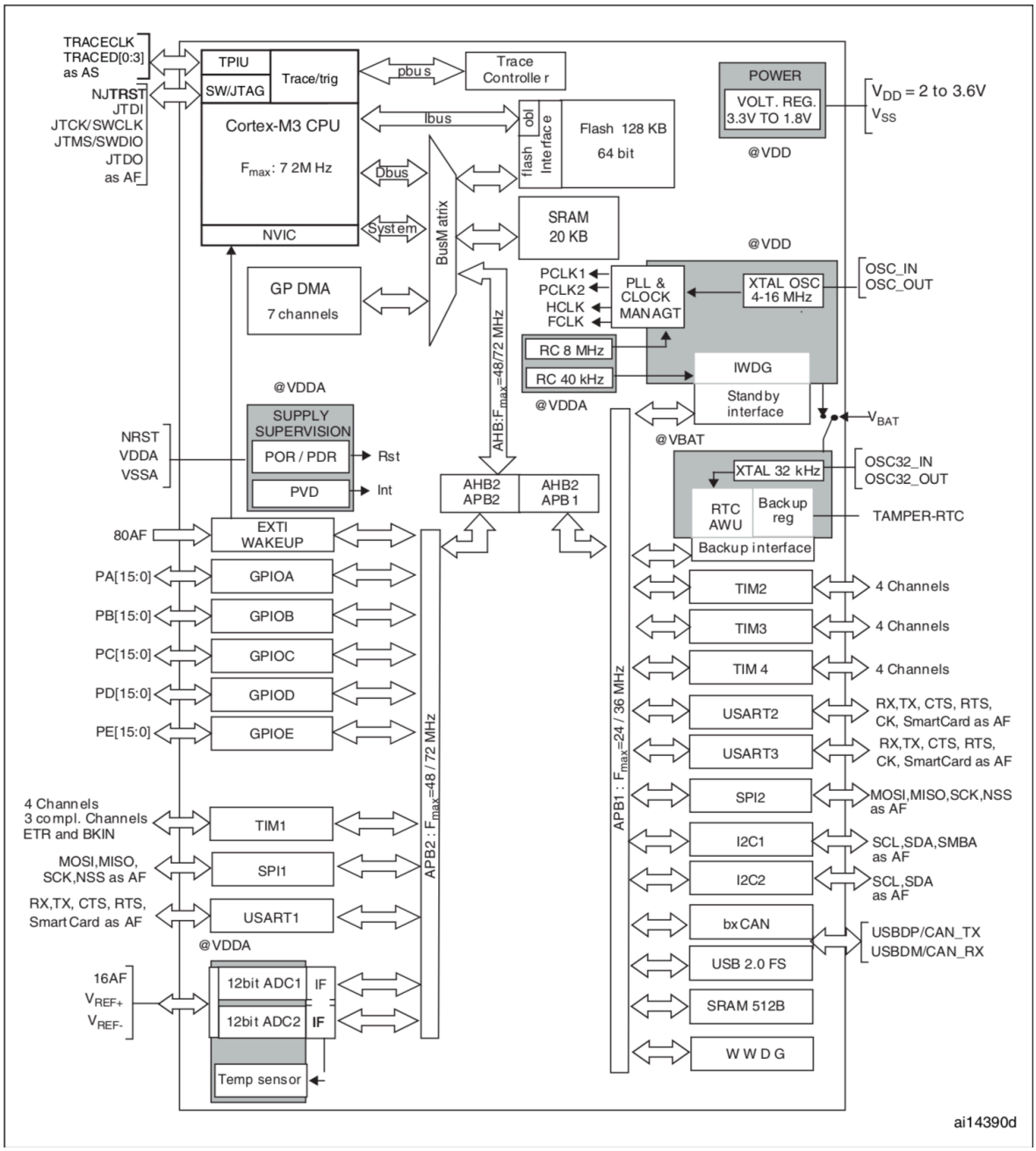

图 4.2.: STM32 时钟线

4.2.3. GPIO 的输出操作

配置为推挽或开漏输出模式的引脚，可以控制其输出高电平或低电平，与之相关的库函数有下面几个：

```
/*控制整个GPIO端口的输出值*/
void GPIO_Write(GPIO_TypeDef * GPIOx, uint16_t PortVal);

/*控制某个引脚的输出值*/
void GPIO_WriteBit(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin,
    BitAction BitVal );

/*拉高（置位，set）某些引脚*/
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);

/*拉低（复位，reset）某些引脚*/
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

其中，第 1 个函数改变整个端口 16 个引脚的输出值，因此它的第二个参数是一个 16 位的无符号整数，从第 15 位到第 0 位分别对应某一端口第 15 号引脚到第 0 号引脚，这种操作一般用于手工进行并行数据传输。第 2 个函数改变某个引脚的输出值，其第三个参数决定是拉高（对应 Bit_SET）还是拉低（对应 Bit_RESET），不过这个函数只能操作某一个引脚。后两个函数分别拉高、拉低某些引脚，第二个参数为用按位或运算符连接的若干引脚，当然，控制单个引脚也是可以的。

下面，我们看看如何用这些函数来控制 GPIO 完成点亮 LED 灯的操作。以下两个函数位于 GPIO.c 文件中。

```
void LED1Operation(u8 on) {
    if(on) {
        GPIO_ResetBits(GPIOA, GPIO_Pin_0);
    } else {
        GPIO_SetBits(GPIOA, GPIO_Pin_0);
    }
}

void LED2Operation(u8 on) {
    GPIO_WriteBit(GPIOA, GPIO_Pin_1, on ? Bit_RESET : Bit_SET);
}
```

代码清单 4.3: LED 控制操作

这里，控制 LED1 的函数用了直接拉高、拉低引脚的库函数，而控制 LED2 的函数则使用了写入某一引脚的库函数。需要注意，也许有的读者知道 Bit_SET 和 Bit_RESET 的值（分别为 1 和 0）之后会想到把上述代码第 10 行的三元表达式替换为 1 - on 甚至是 1 ^ on（位异或运算），但实际上，我们使用了 8 位整型来表达布尔类型，因此任何非零值都应理解为真，如果做出这样的替换，就相当于限制了参只能传入 0 或 1。有时为了较高的性能我们确实会这么做，但一般来说，这是不建议的。

4.2.4. GPIO 的读入操作

配置为浮空输入、上拉输入或者下拉输入模式的引脚，可以读取输入的电平高低，与之相关的库函数有下面几个：

```
1 /*读取某个引脚上的输入电平*/
2 uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
3
4 /*读取整个GPIO端口的输入值*/
5 uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
```

第一个函数读取某个引脚上的输入电平，高电平返回 Bit_SET，低电平返回 Bit_RESET。第二个函数读取整个 GPIO 端口的输入数据，返回一个 16 位无符号数，从第 15 位到第 0 位分别对应某一端口第 15 号引脚到第 0 号引脚上的输入电平。

下面我们看看如何用这些函数检测按键是否被按下了。

```
1 u8 Button1Pressing() {
2     return GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_2) == Bit_SET;
3 }
4
5 u8 Button2Pressing() {
6     return GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_3) == Bit_RESET;
7 }
```

代码清单 4.4: 按键检测

函数的功能简明易懂，但这样，能够完美地实现检测按键的功能吗？

4.2.5. 按键的抖动

由于按键是一种机械结构触发的电子开关，当人手按下按键时，相应信号并不会立即跳变，而是会迅速发生几次跳变，而后稳定下来，当松开按键时，也会发生同样的情

况，这就是按键的抖动，形象的描述见图4.3[2]。这种抖动可能使我们重复检测到按键按下的事件，甚至在松开按键的过程中也会这样，因此，必须想办法消除这种影响。

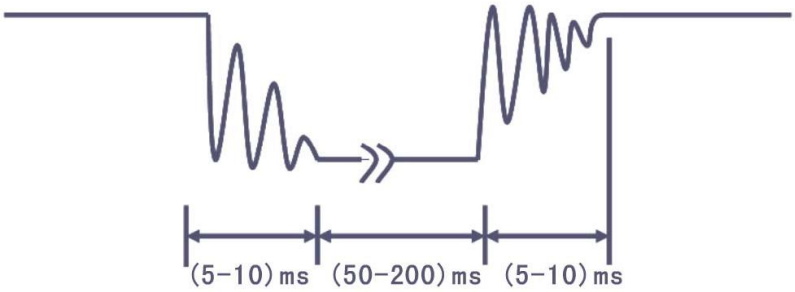

图 4.3.: 按键抖动

我们可以使用一些数字电路来消除这种抖动带来的影响，但这样过于复杂。我们也可以用软件的方式来实现同样的目的，即检测到电平变换后，延时一定时间（例如50ms），再检测一遍，如果仍为按下状态，则认为按键已经按下，否则认为没有按下。但在此之前，让我们想想如何进行延时的操作，要知道，单片机里可没有 <time.h> 之类的操作。

4.2.6. 循环的延时作用

想要延时，最简单的方法是让程序执行相当多次什么也不干的循环。经过几次调参，一个简易的延时函数如下所示:

```
1 void delay(volatile uint32_t ms) {
2     volatile uint32_t i, j;
3     for(i = 0; i < 6000; ++i) {
4         for(j = 0; j < ms; ++j);
5     }
6 }
```

代码清单 4.5: 简易延时函数

一个优秀的编译器在吸了氧气（指开启-O2 优化）之后可能会把空循环甚至幂等的循环（即循环一次与循环多次等价）直接优化掉，因此我们一定要声明变量为 volatile 的。有了这个延时函数，我们就可以解决按键抖动问题了。

[2]http://mooc.chaoxing.com/nodedetailcontroller/visitnodedetail?knowledgeId=630317

4.2.7. 消抖后的按键检测

```
u8 Button1Pressing() {
    if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_2) == Bit_SET) {
        delay(50);
        return GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_2) == Bit_SET;
    } else return 0;
}

u8 Button2Pressing() {
    if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_3) == Bit_RESET) {
        delay(50);
        return GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_3) == Bit_RESET;
    } else return 0;
}
```

代码清单 4.6: 消抖后的按键检测函数

上面这两个函数的实现思路与之前的分析一致，使用延时的方法来消除抖动的影响。至此，我们已经实现了 LED 灯的控制以及按键检测的功能，下面，我们就可以用这些函数来实现各种各样的效果了。

4.2.8. 功能的实现

想要实现我们希望的功能，需要编辑 main.c 文件，这是程序的入口 main() 函数所在的位置。我们需要将程序的逻辑写在 main() 函数内部。不过，在编辑 main() 函数之前，我们需要在文件开头包含刚刚编写的 GPIO.h 头文件。

首先，让我们看看实现一个 LED 灯闪烁功能的程序:

```
int main(void) {
    PinsInit();
    LED2Operation(0);
    while(1) {
        LED1Operation(1); delay(500);
        LED1Operation(0); delay(500);
    }
}
```

代码清单 4.7: LED 灯闪烁

程序首先调用初始化函数对用到的引脚进行初始化，然后将 LED2 熄灭，此后进入一个无限循环，在这个循环中，以大约 1 秒为周期交替点亮、熄灭 LED1。如果编译并下载这个程序，正确地连接电路，我们就可以看到 LED 闪烁的效果了。这里我们使用的 delay() 函数与之前实现的一样。

此外，流水灯、按键控制灯亮灭的程序也都比较平凡，所以下面直接给出。

```
1  int main(void)
2  {
3      PinsInit();
4
5      u8 on = 1;
6      while(1) {
7          LED1Operation(on); LED2Operation(!on);
8          delay(500);
9          on = !on;
10     }
11 }
```

代码清单 4.8: 流水灯

```
1  int main(void)
2  {
3      PinsInit();
4
5      while(1) {
6          LED1Operation(Button1Pressing());
7          LED2Operation(Button2Pressing());
8      }
9  }
```

代码清单 4.9: 按钮控制亮灭

说一些题外话，在流水灯的实现代码中，我们用 u8 类型作布尔类型，而后又对其进行了逻辑非运算，这一操作的具体行为其实与变量 on 的初值有关，在这里，我们初始化 on 为 1，那么我们采用的编译器会把这个逻辑非操作实现为异或运算，即把 on = !on 实现为 on = 1 ^ on，然而，如果我们初始化 on 为其他值，例如 10，那么编译器会用逻辑右移运算来实现逻辑非，得到的变换序列为 10→0→1→0→1→···。如果取别的值，也许编译器会实现出其他行为，有兴趣的读者可以用 gcc 的命令行参数-S 生成汇编代码来一探究竟。

4.3. 中断

在前面的按钮控制亮灭程序中，读者容易发现，我们的程序一直都在检测按钮是否按下，但事实上我们可以认为使用者并不会花很长时间按按钮，所以这种实现（称为轮询检测，polling）很浪费 CPU 资源。在实际的飞行控制应用中，按钮可能被一些传感器的信号线代替，如果两次轮询之间的间隔选择得不合适，很有可能错过一些信息，造成意外。为此，绝大多数 CPU 都提供一种叫做中断的机制，让 CPU 可以被某些事件打断，进入处理相应事件的状态，而不必再轮询检测事件是否发生，这使得 CPU 的控制逻辑更加丰富。

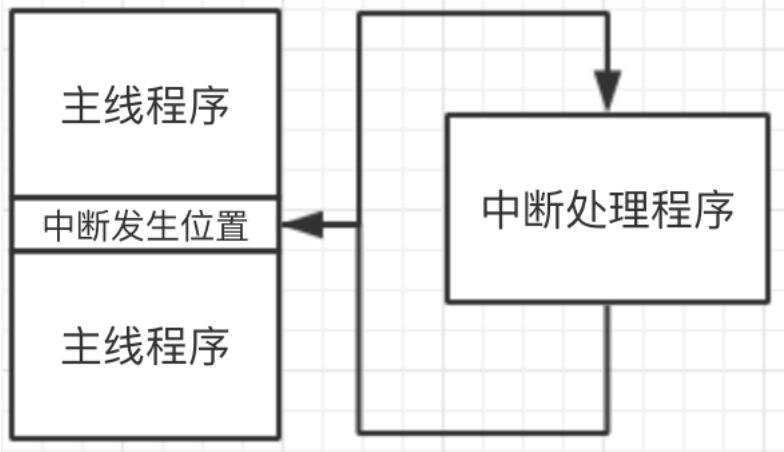

图 4.4.: 中断处理流程

中断处理的过程如上图所示，当主线程序执行时，如果发生引发中断的事件，则 CPU 立即执行相应的中断处理程序，结束以后再回到主线程序，继续运行。事实上，中断程序也会被中断所打断，因此，STM32 的 CPU 引入了优先级机制，只有高优先级的中断才能打断低优先级或同级的中断处理程序，这种中断嵌套的机制是由 CPU 中一个叫做嵌套向量中断控制器（Nested Vectored Interrupt Controller (NVIC)）的核心外设提供的。由于处理中断要保存程序计数器等寄存器的状态，就需要用到栈空间，因此中断的嵌套有一定限度，为避免超过这个限度，一方面要合理设置中断的优先级，另一方面要在程序上保证一个中断不会在处理过程中再次触发，形成无限递归。STM32 中的许多外设都可以作为中断的来源。

4.4. GPIO 的中断操作——EXTI

STM32 为我们提供了来自 GPIO 的中断处理机制，这是由 EXTI 管理的，EXTI 共有 16 条中断信号线，分别对应每个 GPIO 端口的 16 个引脚，换句话说，我们可以把 PA0 的中断信号配置到 EXTI0 上，但不能配置到 EXTI1 上，而且与此同时，PB0 就不能配置中断信号了。与我们之前检测按钮连接引脚的输入电平不同，EXTI 检测的是信号边沿，所以我们可以要求某个引脚在上升沿和/或下降沿时触发中断信号。另一个值得注意的地方是，EXTI0 ～ EXTI15 并不是分别对应着 16 个不同的中断处理函数，实际的情况是 EXTI0 ～ 4 分别占据 5 个中断，EXTI5 ～ 9 共享一个中断，而 EXTI10 ～ 15 共享另一个中断，这种情况下一般要在共享的中断处理函数中检测是哪一个引脚触发了事件。

下面，让我们用中断的方式重新实现之前的按钮控制 LED 灯的实验。

4.4.1. EXTI 中断的配置

要配置 EXTI 中断，我们需要打开 AFIO 的时钟（因为 EXTI 也是 GPIO 的复用功能），配置 GPIO 的 EXTI 功能，初始化 EXTI 和 NVIC。不妨把这些操作抽象成一个函数，放在 GPIO.c 文件中：

```
1  static void InterruptInit() {
2      EXTI_InitTypeDef extiInitStruct;
3      NVIC_InitTypeDef nvicInitStruct;
4
5      RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
6
7      GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource2);
8      GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource3);
9
10     NVIC_SetPriorityGrouping(NVIC_PriorityGroup_1);
11
12     extiInitStruct.EXTI_Line = EXTI_Line2 | EXTI_Line3;
13     extiInitStruct.EXTI_LineCmd = ENABLE;
14     extiInitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
15     extiInitStruct.EXTI_Trigger = EXTI_Trigger_Rising_Falling;
16     EXTI_Init(&extiInitStruct);
17
18     nvicInitStruct.NVIC_IRQChannel = EXTI2_IRQn;
19     nvicInitStruct.NVIC_IRQChannelCmd = ENABLE;
```

```
20      nvicInitStruct.NVIC_IRQChannelPreemptionPriority = 0;
21      nvicInitStruct.NVIC_IRQChannelSubPriority = 0;
22      NVIC_Init(&nvicInitStruct);
23
24      nvicInitStruct.NVIC_IRQChannel = EXTI3_IRQn;
25      nvicInitStruct.NVIC_IRQChannelCmd = ENABLE;
26      nvicInitStruct.NVIC_IRQChannelPreemptionPriority = 1;
27      nvicInitStruct.NVIC_IRQChannelSubPriority = 0;
28      NVIC_Init(&nvicInitStruct);
29  }
```

代码清单 4.10: EXTI 配置

函数中首先定义了 EXTI 和 NVIC 的初始化结构体，然后打开了 AFIO 的时钟。这些是常规操作。

第 7 行和第 8 行分别配置 EXTI2 和 EXTI3 的信号来源为 GPIOA，或者说，把 PA2 和 PA3 连接到 EXTI2 和 EXTI3 上。需要指出的是，这个配置操作的寄存器是属于 GPIO 的，因此调用的函数前缀是 GPIO，而不是 EXTI。

第 10 行，设置了 NVIC 的优先级组分配策略。STM32 的中断优先级有 4 个二进制位来描述，这 4 个位可以进一步划分为抢占优先级和响应优先级。抢占优先级高的中断可以打断抢占优先级低的中断，但相同抢占优先级的中断不可以相互打断，不过抢占优先级相同时，如果两个中断同时发生，那么响应优先级高的中断先执行，当然，优先级完全相同的中断则是按发生时间先后执行。至于在 4 个位中，抢占优先级占多少位，则是由开发者决定的，例如在这里，我们为抢占优先级分配了 1 位的空间，即 NVIC_PriorityGroup_1。

第 12 ～ 16 行配置了 EXTI2 和 EXTI3 的具体行为，使能了这两条线的中断触发，配置模式为中断模式（而不是事件模式，即可以引发其他外设做出动作），并把触发边沿设置为既检测上升沿，又检测下降沿，这是因为我们既需要知道按钮按下，又需要知道按钮松开。

第 18 ～ 22 行和第 24 ～ 28 行告诉 NVIC 去设置 EXTI2 和 3 的中断优先级，同时使能两个中断。其中的 EXTI2_IRQn 和 EXTI3_IRQn 是两个中断的中断号，由外设库预定义好。可以看出，我们为 EXTI2 设置的抢占优先级为 0，为 EXTI3 设置的抢占优先级为 1，两者的响应优先级相同。要注意，数字越小意味着优先级越高。

这样一来中断就配置完成了，不要忘记在 PinsInit() 函数的最后调用这个函数。

4.4.2. EXTI 中断处理函数

完成了中断的配置，我们还需要实现相应的中断处理函数，否则中断发生时会进入默认的处理函数——死循环中。按照习惯，中断处理函数需要放在工程目录中的 User/stm32f10x_it.c 中，并在 User/stm32f10x_it.h 中给出声明。

首先我们需要在 stm32f10x_it.c 中引入”GPIO.h” 这个头文件，以完成对 LED 的控制。然后，需要在文件中添加下面这两个函数:

```
1  void EXTI2_IRQHandler() {
2      if(EXTI_GetITStatus(EXTI_Line2) == SET) {
3          LED1Operation(Button1Pressing());
4          EXTI_ClearITPendingBit(EXTI_Line2);
5      }
6  }
7
8  void EXTI3_IRQHandler() {
9      if(EXTI_GetITStatus(EXTI_Line3) == SET) {
10         LED2Operation(Button2Pressing());
11         EXTI_ClearITPendingBit(EXTI_Line3);
12     }
13 }
```

代码清单 4.11: 中断处理函数

代码的内容先按下不表，这两个函数最为重要的是它们的名字，为了指明它们是相应的中断处理函数，它们的名字必须按照规定好的来书写，而这是在启动代码 Startup/startup_stm32f10x_md.s 中定义的，具体的语句如下:

```
1  /*摘自第145行*/
2      .word   FLASH_IRQHandler
3      .word   RCC_IRQHandler
4      .word   EXTI0_IRQHandler
5      .word   EXTI1_IRQHandler
6      .word   EXTI2_IRQHandler
7      .word   EXTI3_IRQHandler
8      .word   EXTI4_IRQHandler
```

这里用汇编语言定义了中断向量表，规定了函数的名字，因此我们写的中断处理函数必须与这里规定的名字完全一致，否则就无法被 gcc 正确链接，从而无法被 NVIC 调用。

再来看函数具体的内容，首先我们获取相应 EXTI 线的中断标志位，判断中断是否

确实发生。事实上这一步并不一定是必须的，因为我们的中断函数几乎不可能在其他情况下触发，这种操作更重要的应用场景是多个中断共享一个处理函数时用于判断具体是哪个事件触发了中断。接下来读取按键的状态并控制 LED。对于按钮来说，这里最好再加一个短暂的延时，等待状态稳定，但如果信号来自于可靠的电子设备，则一般不用等待。最后，我们用软件的方式清除对应 EXTI 线的中断等待标志位（pending bit），防止中断再次触发。这个标志位并不会由硬件清除，所以务必要在代码中完成。

4.4.3. 主程序

下面是 main.c 中的主函数内容。

```
1 int main(void)
2 {
3     PinsInit();
4 
5     while(1);
6 }
```

代码清单 4.12: 主程序

事实上，主函数在完成初始化工作以后，就没有什么需要做的了，为了防止程序跑飞（因为 main 返回以后单片机的行为是难以确定的），所以在最后要加上一个死循环。这时，中断来了就会执行对应的程序了。

4.5. 思考与练习

1. 使用一个外设之前大致要完成哪些工作？
2. 查阅资料，分析推挽输出和开漏输出之间的区别。
3. 与 GPIO 的输入输出相关的函数有哪些？它们各有什么用法？注意，文档中列举的函数不一定全面，你需要查阅标准外设库的帮助文档。
4. 按键软件消抖的原理是什么？
5. 中断检测和轮询检测有什么区别？以 GPIO 为对象，比较两者的优劣。

6. STM32 中，中断的优先级是如何决定中断发生的顺序的?
7. 用轮询检测设计一个多人抢答器。
8. 用中断检测重新完成上一题，你能否感受出区别？注意，有人抢答以后，其他人再抢答将不做出响应。
9. 设计一个三人表决器，即两人及以上通过时显示通过。
10. 尝试学习 IIC 协议，用本章的 GPIO 操作和简易延时函数模拟 IIC 协议，与 MPU6050 等器件进行通信，将获取的数据用某种方法显示出来，例如用数码管。注意选取合适的 GPIO 模式。

5. SysTick 定时器

在上一章中，我们使用了循环的方式来进行延时，这种方法虽然易于编程，但是精度并不高，循环次数的确定并没有一种有效而又准确的方法。因此，对于时间精度要求较高的场合，我们应当考虑利用硬件资源来完成功能，毕竟，STM32 外部的晶体振荡器的频率是相当精确的。

5.1. SysTick 概览

Systick 又叫系统滴答定时器，是包含在 Cortex M3 内核当中的一个核心外设。也就是说，这个器件与具体的芯片种类是无关的，只要使用了 Cortex M3 内核，就一定有这个外设。当然，也正因为 SysTick 要满足这种普适性，它并没有太多的功能，只是一个 24 位递减计时器。这意味着 SysTick 以一定的频率进行递减计数，当数到 0 时，它可以产生一个中断，同时自动重新填入计数初值，开始下一轮计数。当然，这个初值最大只能是 $2^{24}-1$。

利用 SysTick 计数的特性，我们可以方便地实现延时、获取芯片运行时间等。在实时操作系统中，它还可以用来产生一个信号，让操作系统可以执行进程切换，实际上，这才是 SysTick 的最主要作用。

不过，在飞行控制中，我们通常不需要编写操作系统，所以下面我们还是利用 SysTick 来完成最基本的计时操作，重新实现上一章的延时函数，并完成一些额外功能。请注意，如无特殊说明，本部分的每一章都将沿用前几章的工作，包括引脚功能的约定、编写的程序文件等，也就是说，在这里我们沿用上一章中 GPIO 的相关函数。

5.2. SysTick 使用

5.2.1. 预备工作

这一部分的程序将写在 User/文件夹下的 SysTick.c 和 SysTick.h 两个文件中。首先看一看头文件 SysTick.h 中的内容：

```
#ifndef __SYSTICK__H__
#define __SYSTICK__H__

#include "stm32f10x.h"
#include "stm32f10x_conf.h"

void SysTickInit();
void DelayUsingSysTick(uint32_t ms);
void OnSysTick();

extern volatile uint32_t millis;

#endif
```

代码清单 5.1: SysTick.h

可以看到，文件中声明了三个函数、一个变量，其中，SysTickInit() 负责初始化 SysTick 定时器，使其每 1ms 产生一次中断；DelayUsingSysTick(uint32_t ms) 使用 SysTick 定时器进行延时，单位为 ms；OnSysTick() 为 SysTick 中断发生时的处理函数，我们将在 SysTick 的中断服务程序中调用这个函数。变量 millis 负责记录系统运行至今经过的毫秒数，它需要在 SysTick.c 文件中进行定义。下面，我们首先了解一下 SysTick 的初始化。

5.2.2. SysTick 初始化

SysTick 的初始化函数实现如下：

```
void SysTickInit() {
    if(SysTick_Config(SystemCoreClock / 1000)) {
        while(1);
    }
}
```

代码清单 5.2: SysTick.h

可以看到，其初始化工作非常简单，只需调用 SysTick_Config() 这个函数即可。其参数为 SysTick 计数器的初值，意义是经过这么多次时钟脉冲之后会触发中断。此外，这个函数的调用会配置 SysTick 中断的优先级为最低的可能值，及 15（或 0xF），其原

因与操作系统的调度有一定关系，在这里不过多解释。默认情况下，SysTick 的时钟脉冲频率为 72MHz，这也是 SystemCoreClock 这个宏展开后的值（即 72000000），所以，配置计数器初值为 SystemCoreClock / 1000 即可使 SysTick 以 1ms 为周期触发中断。当然，在 1ms 内 STM32 大概能够执行 10^4 条指令，因此几乎无需担心 SysTick 中断不能在这段时间内处理完成。如果配置成功，调用 SysTick_Config() 会返回 0，否则返回非零值，因此我们需要判断这个返回值是多少，并在初始化失败的情况下让程序陷入死循环。这是因为 SysTick 是一个非常基本的器件，初始化失败往往意味着许多功能无法完成，甚至可能暗示芯片存在其他重大问题。