* 1. stm32开发基础
     1. stm32硬件介绍

STM32是意法半导体 (STMicroelectronics)公司推出的ARM Cortex-M内核单片机系列，按内核架构分为不同产品：主流产品（STM32F0、STM32F1、STM32F3）、超低功耗产品（STM32L0、STM32L1、STM32L4、STM32L4+）、高性能产品（STM32F2、STM32F4、STM32F7、STM32H7）。

本章所使用的型号为STM32F4DISCOVERY，如图所示，它基于STM32F407VGT6微控制器，包括一个板载的有嵌入式调试工具接口的ST-LINK/V2，ST MEMS 数字加速器，ST MEMS 数字麦克风，音频DAC集成了class D扬声器驱动器，LED 灯，按钮和一个USB OTG的micro-AB 连接器。

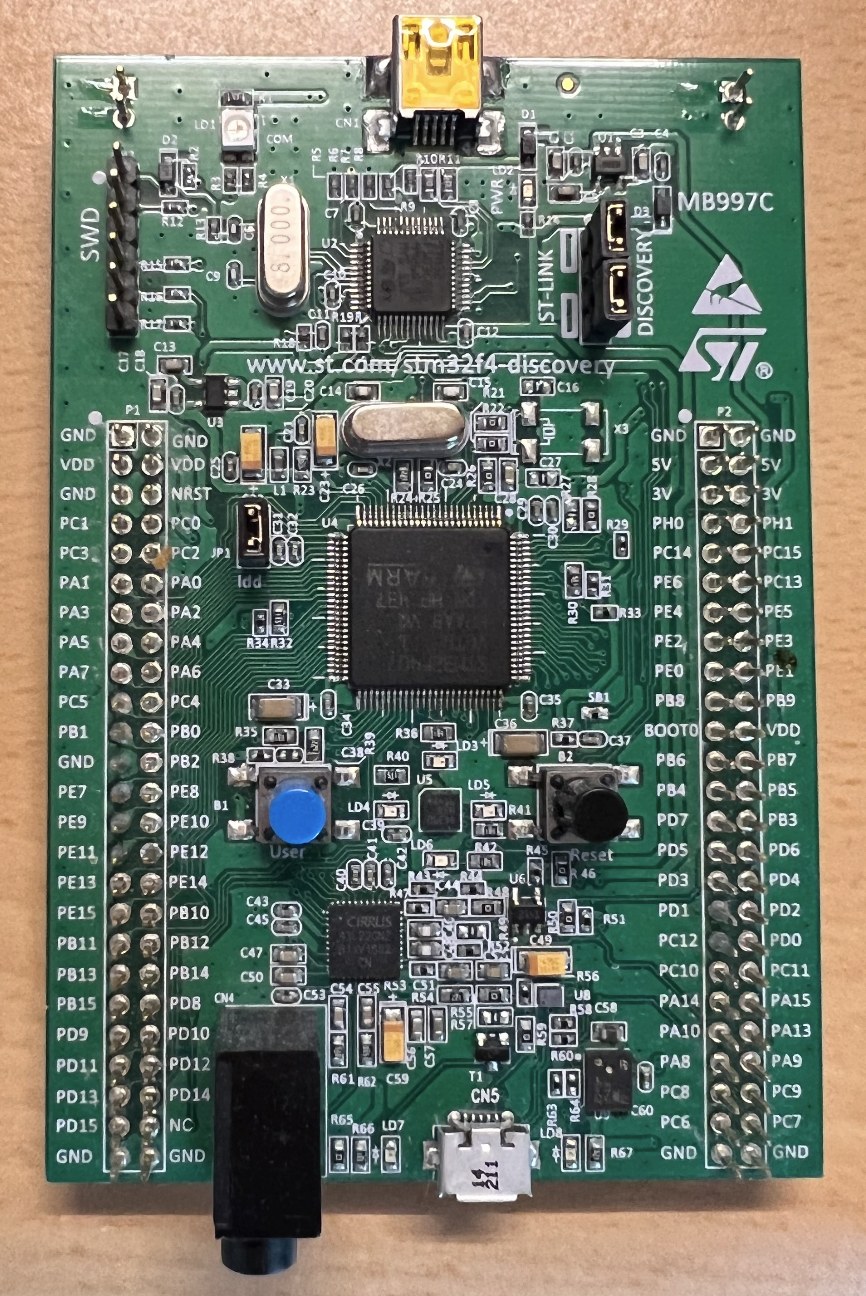


图 STM32F4DISCOVERY

板载ST-Link调试器：以前做嵌入式开发需要额外购买仿真器，它可以替代你目标系统中的MCU微控制单元，仿真它的运行。仿真器运行起来和实际的目标处理器一样，但是增加了其它功能，使你能够通过桌面计算机或其它调试界面来观察MCU中的程序和数据，并控制MCU运行。

音频DAC：DAC是数字模拟转换器的意思，是一种将数字信号转换为模拟信号的设备。

8个LED指示灯：LD1（红/绿）用于USB通讯；LD2（红）用于 3.3V 上电；4个用户LED灯，LD3（橙），LD4（绿），LD5（红）和LD6（蓝）；2 个USB OTG的LED灯，LD7（绿）VBus，LD8（红）过流指示。

2个按钮：分别为用户按键和复位键，图中左侧的按钮即为用户按键。

关于板子的更多信息，可以登录官方网站进行获取：https://www.st.com/zh/evaluation-tools/stm32f4discovery.html

* + 1. 实验环境准备

（1）下载vscode，下载网址：<https://code.visualstudio.com/>。

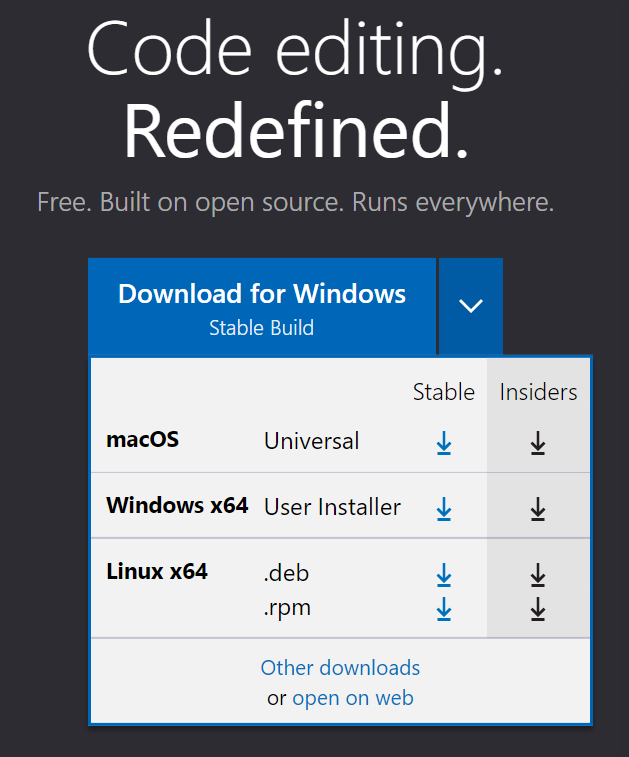


图 下载vscode

（2）在扩展商店中安装PlateformIO，如图所示。



图 安装PlateformIO

（3）安装后重启vscode，会多出一个图标，点击这个图标，然后在弹出的菜单里点Open，进入主界面。

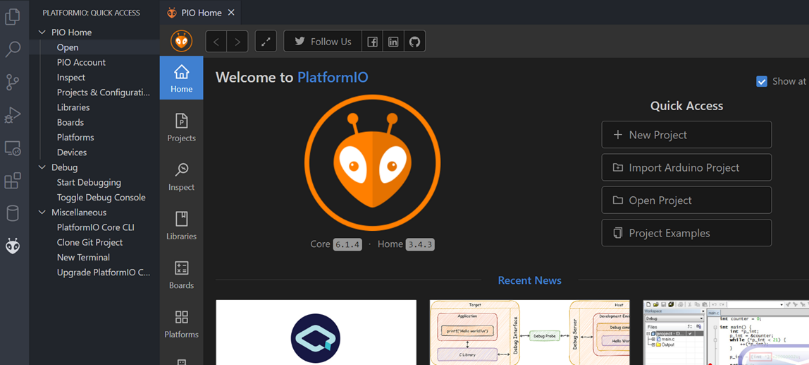


图 进入主界面

如果出现类似下图报错，解决办法是科学上网。

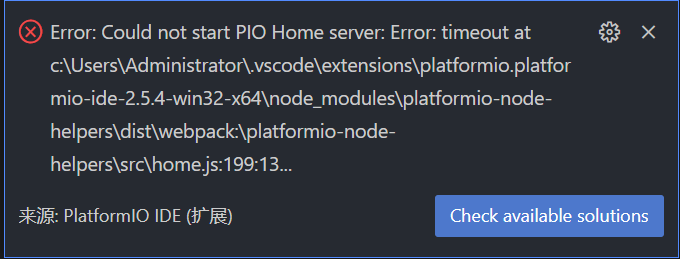


图 报错

（4）在主页点击New Project新建一个项目。

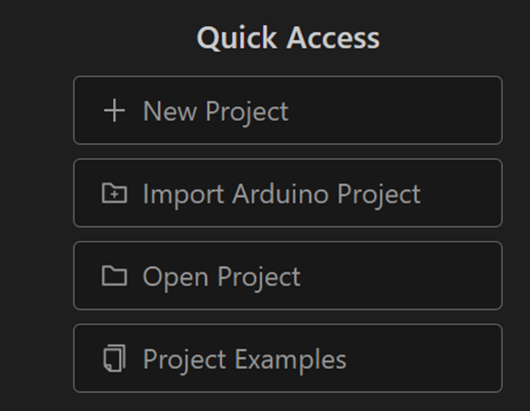


图 新建项目

（5）填写项目名称，选择ST STM32F4DISCOVERY开发板，选择STM32Cube框架，选择项目保存位置。

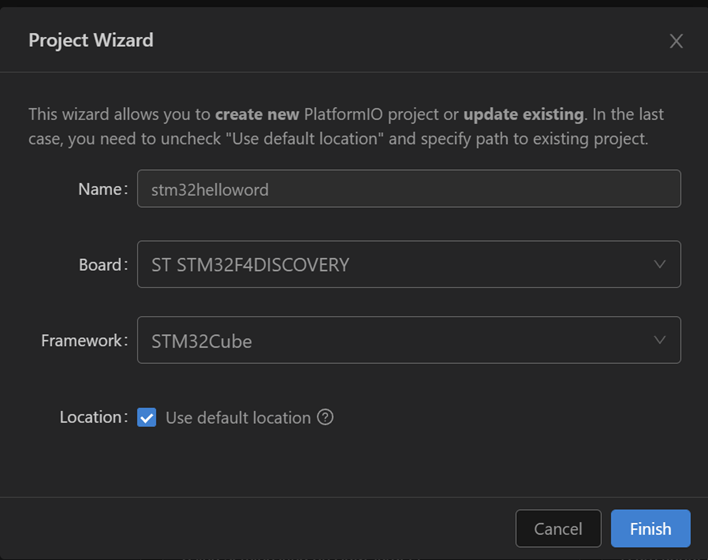


图 填写项目信息

（6）接下来会自动下载相关的开发环境，第一次使用时会比较久，耐心等待即可。

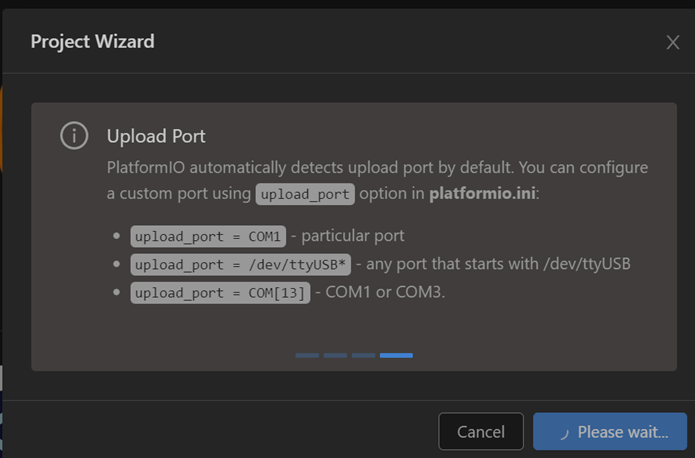


图 下载开发环境

（7）src下新建main.c文件，开始编写代码。

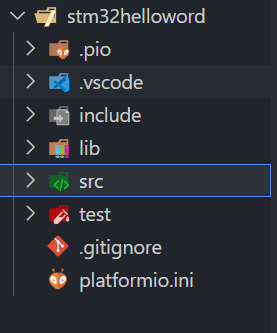


图 下载开发环境

编写程序结构如下：

int main(void) {

//1初始化

//2循环

while (1) {

}

return 0;

}

（8）点击下面的env按钮，在弹出的菜单中选择我们连接的板子

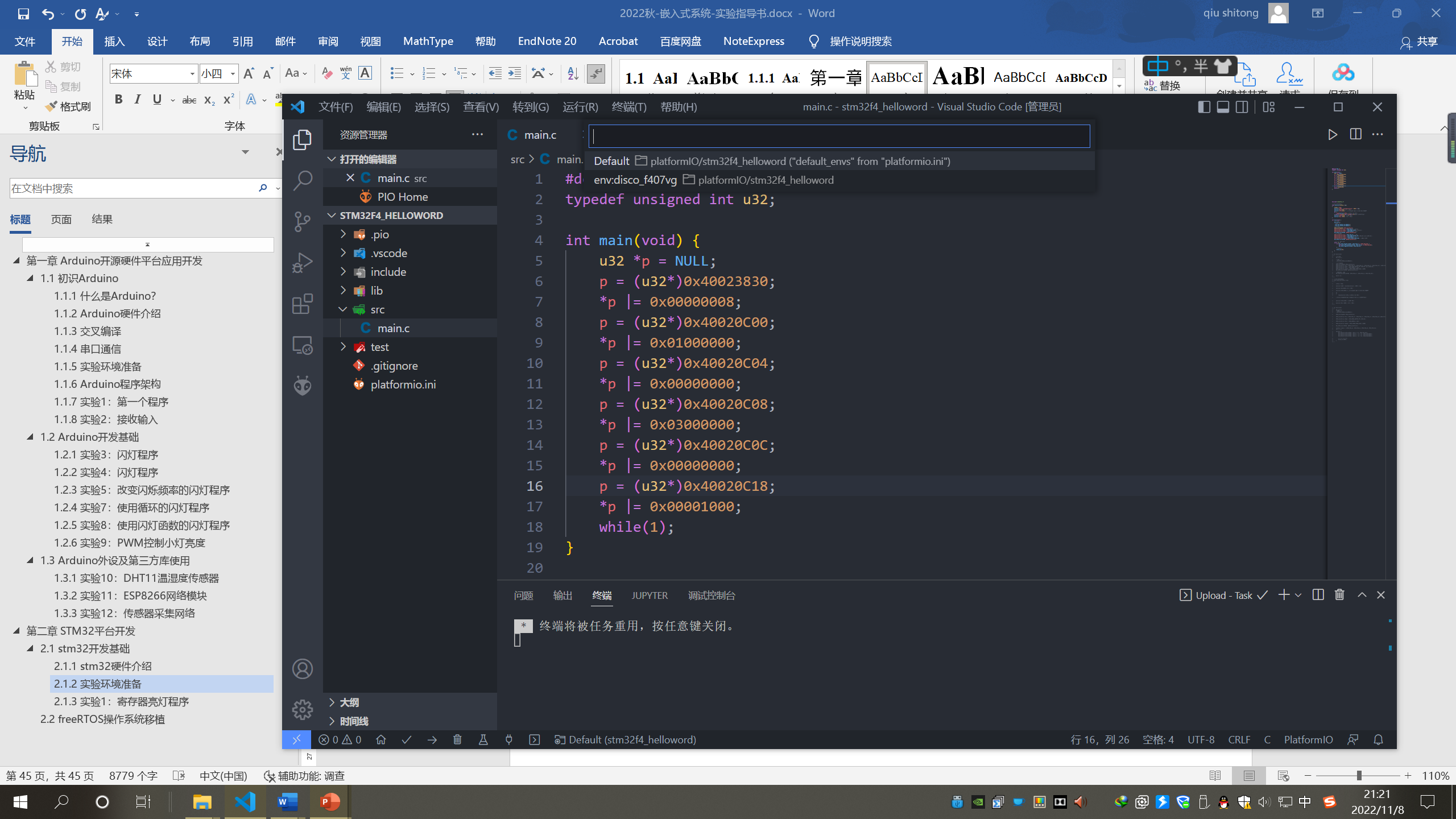


图 选择开发版

（9）点击上传按钮，将程序编译烧写到板子上

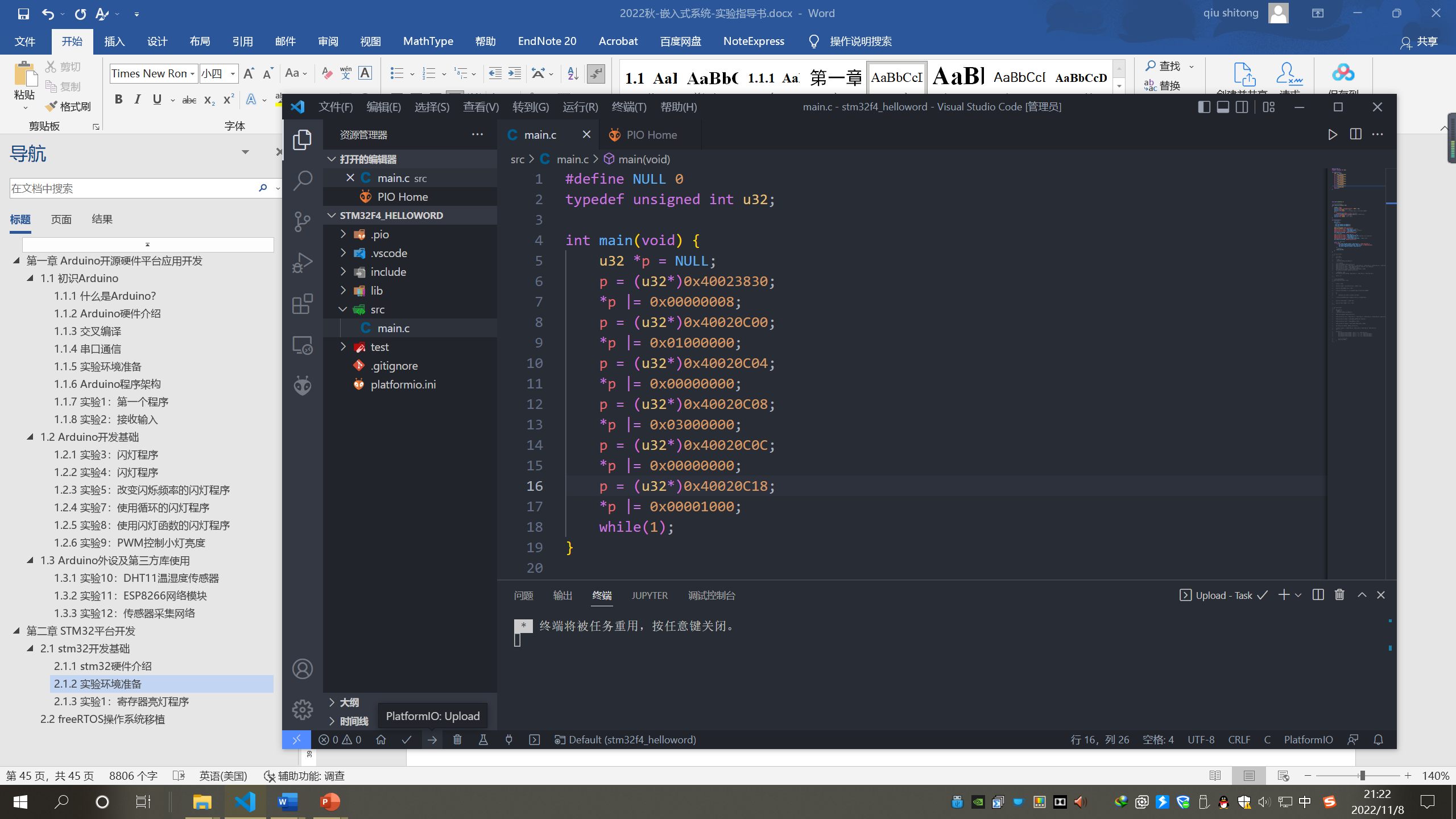


图 上传程序

* + 1. 实验1：寄存器亮灯程序

嵌入式设备要实现如同PC上的hello world，通常比较麻烦。但是实现IO口的反转反而比较简单。所以嵌入式的Hello World通常是实现IO口的反转，使与IO口相连的LED灯闪烁。

开发板上可供编程的LED灯只有4个，我们需要找到这些LED灯与哪些IO口相连。

开发板的原理图在PCB SCH文件夹中。

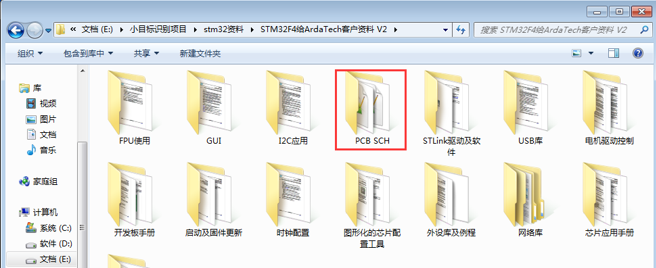


图 PCB SCH文件夹

我们可以查看pdf文件里的原理图。



图 原理图

LD3～LD6为四个LED灯。

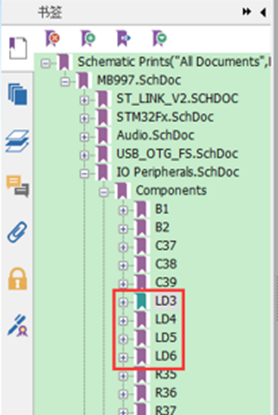


图 目录

可以看到几个LED灯与IO口PD12～PD15相连。灯的另外一头接地，说明IO口给高电平亮灯。

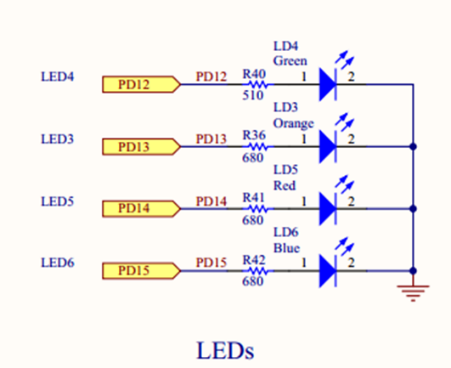


图 用户LED灯电路图

这样我们就需要查阅芯片开发手册，看如何配置IO口。我们看开发参考手册即可。



图 芯片应用手册

开发参考手册总共33章，控制IO口我们只需要看第2、5、6章。分别是内存与总线架构、复位与时钟控制、通用I/O。

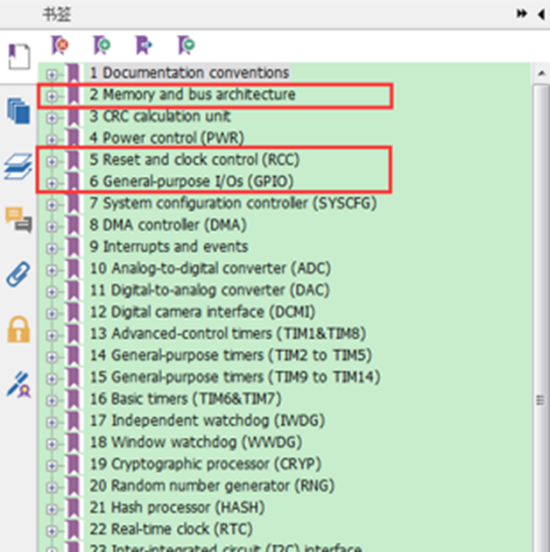


图 查看开发参考手册

因为我们要控制IO口所以要看通用IO这一章节。但使用任何设备都需要时钟信号，只有给相应设备时钟信号，设备才能运转。设备都是挂接在总线上，因此我们需要看复位与时钟控制、内存与总线架构这两章节。另外，片内外设都是统一编址的（使用内存地址），我们还需要知道设备的起始地址。

首先我们可以看2.3节(内存映射)，通过表中我们可以看出GPIOD（D系列IO口）挂接在AHB1高速总线上。 GPIOD的控制寄存器的地址范围是0X4002 0C00 ～ 0x4002 0FFF。RCC（复位与时钟控制）寄存器的地址范围是0x4002 3800 ～ 0x4002 3BFF。

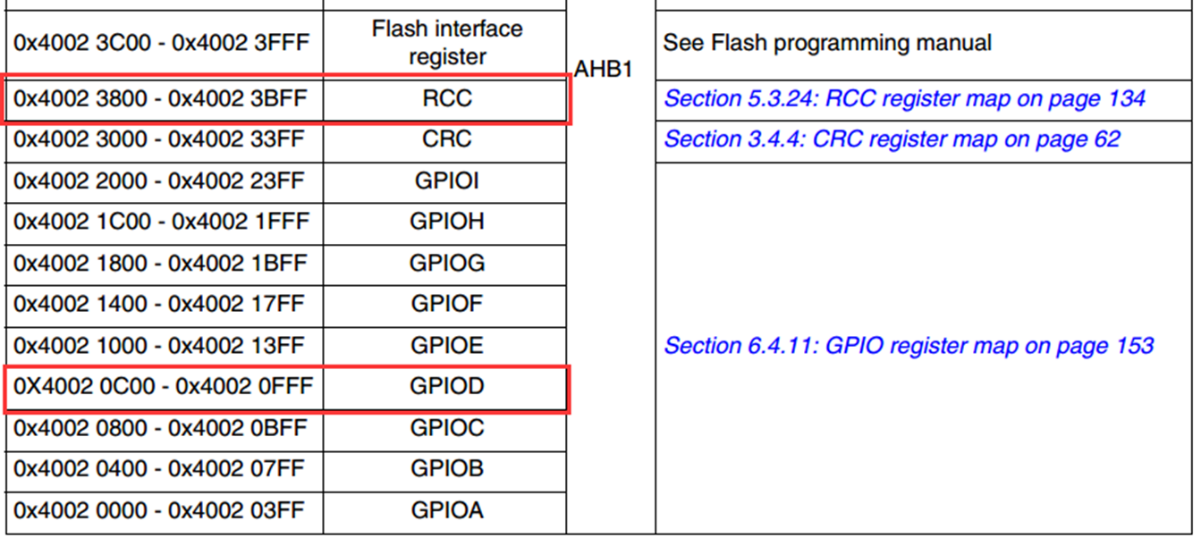
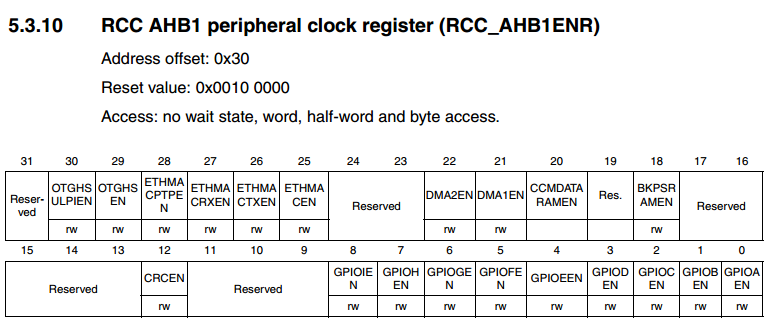


图 内存映射表

我们再可以直接看5.3节（复位与时钟寄存器部分），因此我们只需要将寄存器的第3位置为1就可以让D系列IO口的时钟使能。那么具体代码中该如何操作呢？



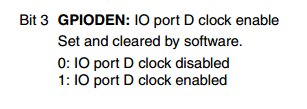


图 复位与时钟寄存器

RCC的起始地址为： 0x4002 3800，RCC\_AHB1ENR寄存器的偏移地址为：0x30，RCC\_AHB1ENR的内存地址为：0x4002 3830（ 0x4002 3800 + 0x30），那么只需将起始地址为0x4002 3830的一个字的内容置为0x0010 0008（复位值为0x0010 0000）

最后我们再看6.3节（IO口功能描述）：

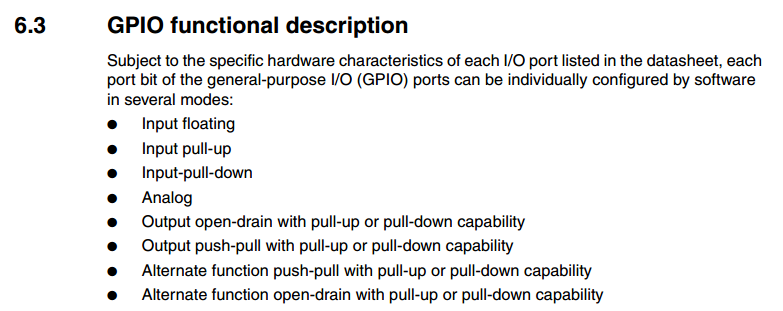


图 IO口功能描述

再直接看6.4（GPIO寄存器部分），如果我们要控制LED4，那么只需要将MODER12字段设置成01即可。

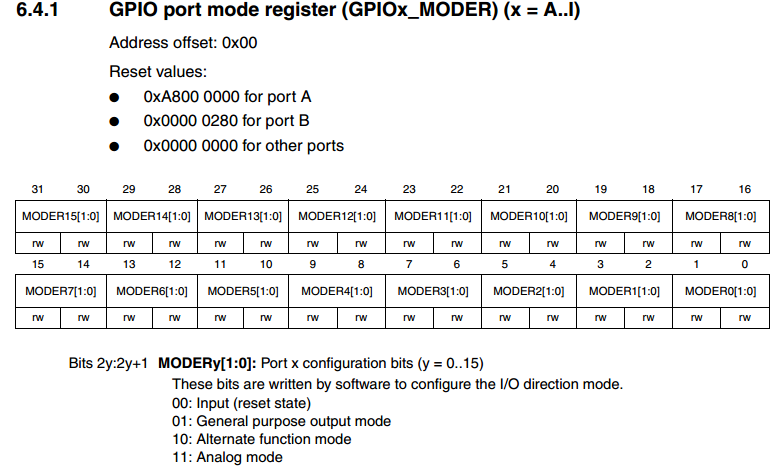


图 IO接口模式寄存器

我们使用推挽输出模式，即OT12置为0。

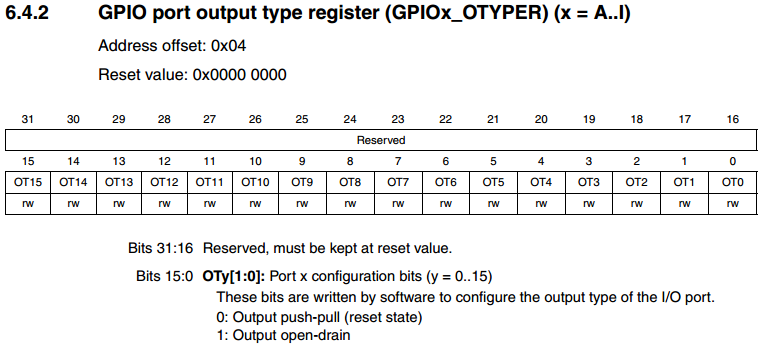


图 输出模式寄存器

我们只是点亮LED灯，任意速度即可，我们这里可以把OSPEEDR12字段置为11。

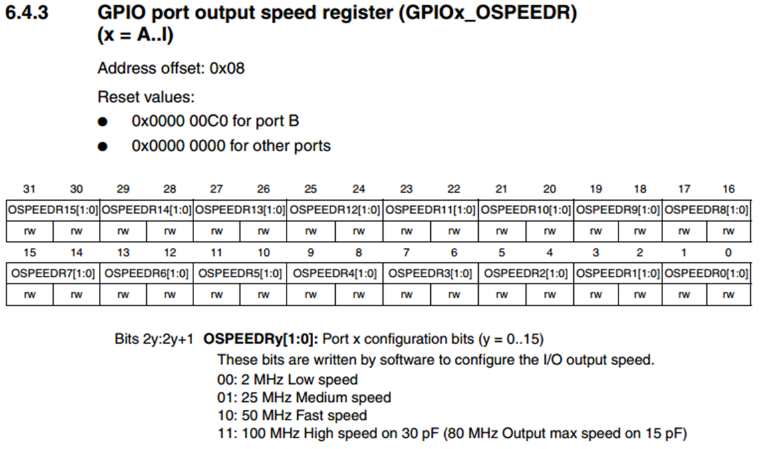


图 接口速度寄存器

我们只是点亮LED灯，不用上拉也不用下拉。PUPDR12字段设置为00。

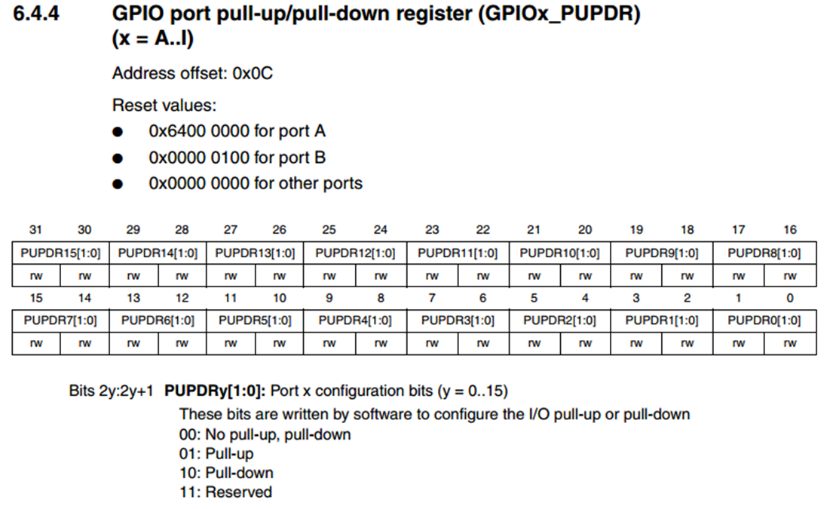


图 上拉/下拉寄存器

我们只做输出用，所以不管输入寄存器。这里ODR12置为1即12号针脚输出高电平。

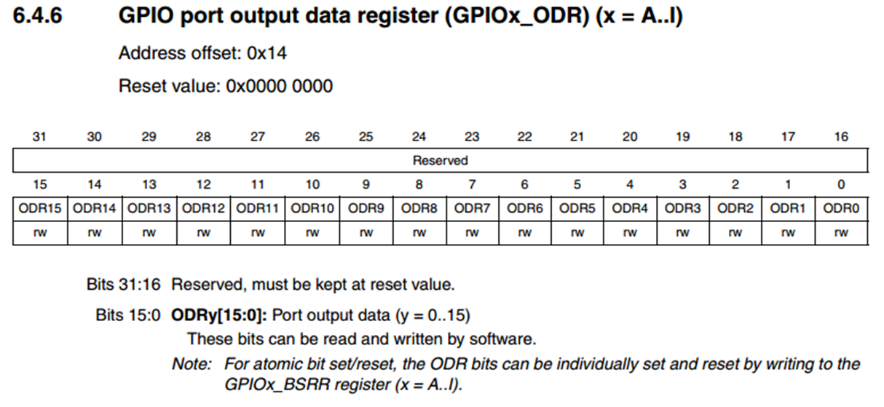


图 输出数据寄存器

然而ODR寄存器并不方便设置。通常使用BSRR寄存器设置高低电平。

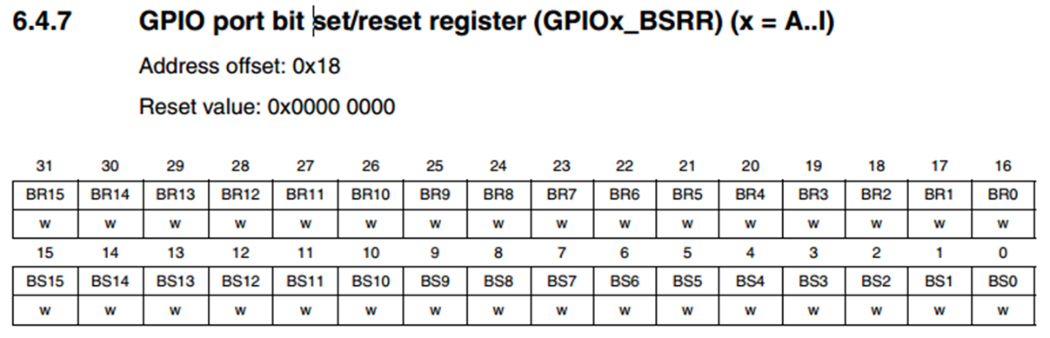


图 BSRR寄存器

看下图，因此BS12字段置为1则输出高电平，BR12字段置为1则输出低电平。

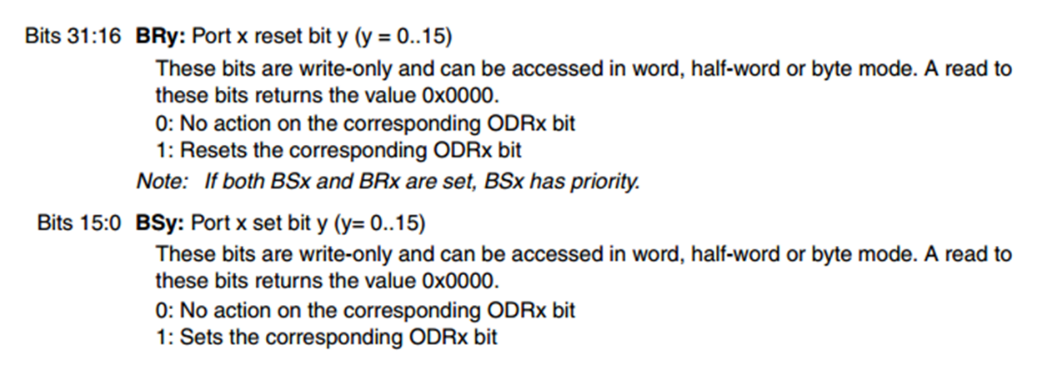


图 BSRR寄存器设置

那么点亮LED灯应该分为以下几步：

1. GPIOD的时钟使能
2. MODER12置为01，输出模式
3. OT12置为0，推挽输出
4. OSPEEDR12置为11，高速模式
5. PUPDR12置为00，不上拉也不下拉
6. BS12置为1，即ODR12置为1，输出高电平

对应的内存操作为：

1. 地址： 0x4002 3830 内容： 0x0010 0008
2. 地址： 0x4002 0C00 内容： 0x0000 0000
3. 地址： 0x4002 0C04 内容： 0x0000 1000
4. 地址： 0x4002 0C08 内容： 0x0300 0000
5. 地址： 0x4002 0C0C 内容： 0x0000 0000
6. 地址： 0x4002 0C18 内容： 0x0000 1000

具体代码如下：

#define NULL 0

typedef unsigned int u32;

int main(void) {

u32 \*p = NULL;

p = (u32\*)0x40023830;

\*p |= 0x00000008;

p = (u32\*)0x40020C00;

\*p |= 0x01000000;

p = (u32\*)0x40020C04;

\*p |= 0x00000000;

p = (u32\*)0x40020C08;

\*p |= 0x03000000;

p = (u32\*)0x40020C0C;

\*p |= 0x00000000;

p = (u32\*)0x40020C18;

\*p |= 0x00001000;

while(1);

}

上传程序之后，便可以看到LD3被点亮。

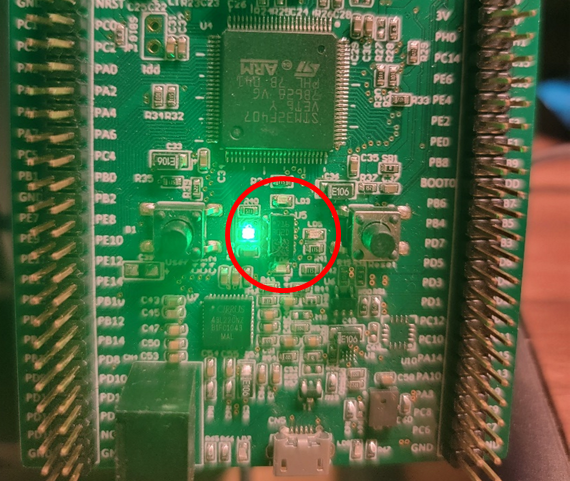


图 点亮LD3

* + 1. 实验2：HAL库函数亮灯程序

经过上面的实验我们知道操作寄存器不需要调用外部库，但是这样显然是非常麻烦的一件事，而且可移植性和可读性很差。那么接下来推荐使用STM32Cube框架库函数来实现。

结合前一个实验的内容，点亮小灯的步骤可以分为四步：

1. 第一步是初始化HAL库，用的是HAL\_Init()这个API。HAL就是硬件抽象层，它帮我们隐藏了硬件接口细节，不用再去查找芯片的内存结构使用寄存器了。
2. 第二步是时钟使能，因为使用任何外部设备都需要时钟信号，只有给相应设备时钟信号，设备才能运转，这里使用了\_\_HAL\_RCC\_GPIOD\_CLK\_ENABLE()这个API，它其实是一个宏定义，红色的部分是可以改变的，比如我们要给GPIOA时钟信号，就要把D改成A。
3. 第三步是初始化GPIO，用来设置GPIO接口的输入输出模式，使用HAL\_GPIO\_Init(GPIOD, &GPIO\_InitStruct)这个API，第一个参数是指定GPIO的系列，比如GPIOD，它也是一个宏定义，第二个参数是一个结构体，用来配置接口的详细参数，我们后面再讲。
4. 第四步是对完成初始化的GPIO接口进行操作，其中一个最基础的API是HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_13, GPIO\_PIN\_SET)，这个API的三个参数都是宏定义，第一个参数是指定GPIO的系列，第二个参数是指定具体接口，比如13号接口，第三个参数代表输入高电平或者低电平，比如设置为SET就代表高电平，RESET则代表低电平。

对于上述步骤中的宏定义和结构体相关的定义，可以查看stm32f4xx\_hal\_gpio.h（在编辑器里跳转查看源码），比如步骤（3）中的宏定义结构如下:

typedef struct

{

uint32\_t Pin; /\*!< Specifies the GPIO pins to be configured.

This parameter can be any value of @ref GPIO\_pins\_define \*/

uint32\_t Mode; /\*!< Specifies the operating mode for the selected pins.

This parameter can be a value of @ref GPIO\_mode\_define \*/

uint32\_t Pull; /\*!< Specifies the Pull-up or Pull-Down activation for the selected pins.

This parameter can be a value of @ref GPIO\_pull\_define \*/

uint32\_t Speed; /\*!< Specifies the speed for the selected pins.

This parameter can be a value of @ref GPIO\_speed\_define \*/

uint32\_t Alternate; /\*!< Peripheral to be connected to the selected pins.

This parameter can be a value of @ref GPIO\_Alternate\_function\_selection \*/

}GPIO\_InitTypeDef;

有了前面的铺垫，使用HAL库函数亮灯程序代码如下：

#include"stm32f4xx.h"

int main(void) {

//初始化

HAL\_Init();

//时钟使能

\_\_HAL\_RCC\_GPIOD\_CLK\_ENABLE();

//初始化GPIO

GPIO\_InitTypeDef GPIO\_InitStruct;

GPIO\_InitStruct.Pin = GPIO\_PIN\_13 | GPIO\_PIN\_15;// LED3/6

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;//正常输出模式：推挽输出

GPIO\_InitStruct.Pull = GPIO\_NOPULL;//不拉

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;//低速模式即可

HAL\_GPIO\_Init(GPIOD, &GPIO\_InitStruct);

//点亮LED3和LED6

HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_13 | GPIO\_PIN\_15, GPIO\_PIN\_SET);

while (1);

}

* + 1. 实验3：使用按钮控制亮灯

STM32F4DISCOVERY有一个用户按钮，我们可以利用按钮来控制LED灯亮灭。与LED灯一样，按钮也是属于外设，在开发版手册第35页有它的电路图，可以看到按钮连接的是PA0，而且当按钮按下的时候是高电平。

接下来我们看一下如何去实现，首先我们额外添加对于按钮的GPIO初始化，把它设置为输入模式，然后在循环中通过HAL\_GPIO\_ReadPin函数去读取按钮的电平，当按钮从高电平变为低电平的时候，我们就可以认为一次按钮操作，然后通过HAL\_GPIO\_TogglePin函数来翻转LED灯的电平，从而实现对LED灯的开关控制。在两次判断之间需要添加一定的延时防止读取的过快，这里给大家提供了一个延时函数，它是利用系统节拍定时器来实现的。具体代码如下：

#include"stm32f4xx.h"

//延时函数（毫秒）

void delay\_ms(int32\_t nms) {

int32\_t temp;

SysTick->LOAD = SystemCoreClock / 8000 \* nms;

SysTick->VAL=0X00;//清空计数器

SysTick->CTRL=0X01;//使能，减到零是无动作，采用外部时钟源

do {

temp=SysTick->CTRL;//读取当前倒计数值

} while((temp&0x01)&&(!(temp&(1<<16))));//等待时间到达

SysTick->CTRL=0x00; //关闭计数器

SysTick->VAL =0X00; //清空计数器

}

int main(void) {

//初始化

HAL\_Init();

//时钟使能

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOD\_CLK\_ENABLE();

//初始化按钮

GPIO\_InitTypeDef GPIO\_InitStruct;

GPIO\_InitStruct.Pin = GPIO\_PIN\_0;// Button

GPIO\_InitStruct.Mode = GPIO\_MODE\_INPUT;//输入模式

GPIO\_InitStruct.Pull = GPIO\_NOPULL;//不拉

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct);

//初始化LED灯

GPIO\_InitStruct.Pin = GPIO\_PIN\_13;// LED3

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;//正常输出模式：推挽输出

GPIO\_InitStruct.Pull = GPIO\_NOPULL;//不拉

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;//低速模式即可

HAL\_GPIO\_Init(GPIOD, &GPIO\_InitStruct);

//通过按钮控制小灯

while (1) {

if (HAL\_GPIO\_ReadPin(GPIOA, GPIO\_PIN\_0) == GPIO\_PIN\_SET) {

while(HAL\_GPIO\_ReadPin(GPIOA, GPIO\_PIN\_0) == GPIO\_PIN\_SET);

HAL\_GPIO\_TogglePin(GPIOD, GPIO\_PIN\_13);

}

//消除抖动

delay\_ms(10);

}

}

* + 1. 实验4：使用按钮控制亮灯（中断版本）

之前的实验中我们的程序都是在循环里面完成的，其实是通过程序查询的方式来判断按钮的状态，这样的效率比较低，通常情况下我们都应该使用中断的方式来实现。要使用中断，我们需要先建立stm32f4xx\_it.h和stm32f4xx\_it.c来管理中断函数，添加到工程中，目录结构如图所示。

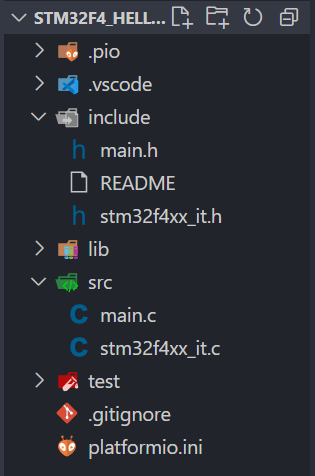


图 目录结构

stm32f4xx\_it.h中的内容为：

/\* Define to prevent recursive inclusion -------------------------------------\*/

#ifndef \_\_STM32F4xx\_IT\_H

#define \_\_STM32F4xx\_IT\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

/\* Includes ------------------------------------------------------------------\*/

#include "main.h"

/\* Exported types ------------------------------------------------------------\*/

/\* Exported constants --------------------------------------------------------\*/

/\* Exported macro ------------------------------------------------------------\*/

/\* Exported functions ------------------------------------------------------- \*/

void NMI\_Handler(void);

void HardFault\_Handler(void);

void MemManage\_Handler(void);

void BusFault\_Handler(void);

void UsageFault\_Handler(void);

void SVC\_Handler(void);

void DebugMon\_Handler(void);

void PendSV\_Handler(void);

void SysTick\_Handler(void);

void EXTI0\_IRQHandler(void);

#ifdef \_\_cplusplus

}

#endif

#endif /\* \_\_STM32F4xx\_IT\_H \*/

stm32f4xx\_it.c中的内容为：

/\* Includes ------------------------------------------------------------------\*/

#include "main.h"

#include "stm32f4xx\_it.h"

/\* Private typedef -----------------------------------------------------------\*/

/\* Private define ------------------------------------------------------------\*/

/\* Private macro -------------------------------------------------------------\*/

/\* Private variables ---------------------------------------------------------\*/

/\* Private function prototypes -----------------------------------------------\*/

/\* Private functions ---------------------------------------------------------\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Cortex-M4 Processor Exceptions Handlers \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* @brief This function handles NMI exception.

\* @param None

\* @retval None

\*/

void NMI\_Handler(void)

{

}

/\*\*

\* @brief This function handles Hard Fault exception.

\* @param None

\* @retval None

\*/

void HardFault\_Handler(void)

{

/\* Go to infinite loop when Hard Fault exception occurs \*/

while (1)

{

}

}

/\*\*

\* @brief This function handles Memory Manage exception.

\* @param None

\* @retval None

\*/

void MemManage\_Handler(void)

{

/\* Go to infinite loop when Memory Manage exception occurs \*/

while (1)

{

}

}

/\*\*

\* @brief This function handles Bus Fault exception.

\* @param None

\* @retval None

\*/

void BusFault\_Handler(void)

{

/\* Go to infinite loop when Bus Fault exception occurs \*/

while (1)

{

}

}

/\*\*

\* @brief This function handles Usage Fault exception.

\* @param None

\* @retval None

\*/

void UsageFault\_Handler(void)

{

/\* Go to infinite loop when Usage Fault exception occurs \*/

while (1)

{

}

}

/\*\*

\* @brief This function handles SVCall exception.

\* @param None

\* @retval None

\*/

void SVC\_Handler(void)

{

}

/\*\*

\* @brief This function handles Debug Monitor exception.

\* @param None

\* @retval None

\*/

void DebugMon\_Handler(void)

{

}

/\*\*

\* @brief This function handles PendSVC exception.

\* @param None

\* @retval None

\*/

void PendSV\_Handler(void)

{

}

/\*\*

\* @brief This function handles SysTick Handler.

\* @param None

\* @retval None

\*/

void SysTick\_Handler(void)

{

HAL\_IncTick();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* STM32F4xx Peripherals Interrupt Handlers \*/

/\* Add here the Interrupt Handler for the used peripheral(s) (PPP), for the \*/

/\* available peripheral interrupt handler's name please refer to the startup \*/

/\* file (startup\_stm32f4xx.s). \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* @brief This function handles External line 0 interrupt request.

\* @param None

\* @retval None

\*/

void EXTI0\_IRQHandler(void)

{

HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_0);

}

要配置按钮的中断事件，我们可以先查看开发参考手册的202页：

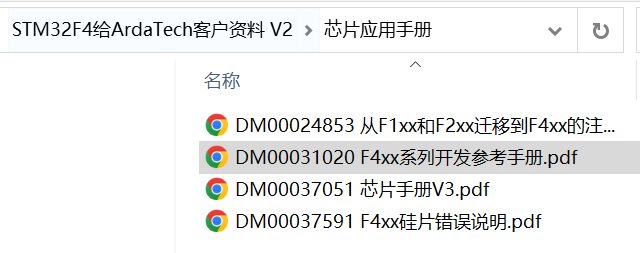


图 开发参考手册

如图，到按钮的IO口PA0与0号中断线相连，因此我们需要在管理中断的stm32f4xx\_it.c中添加相应的函数来处理PA0上的中断事件，即代码中的EXTI0\_IRQHandler函数。

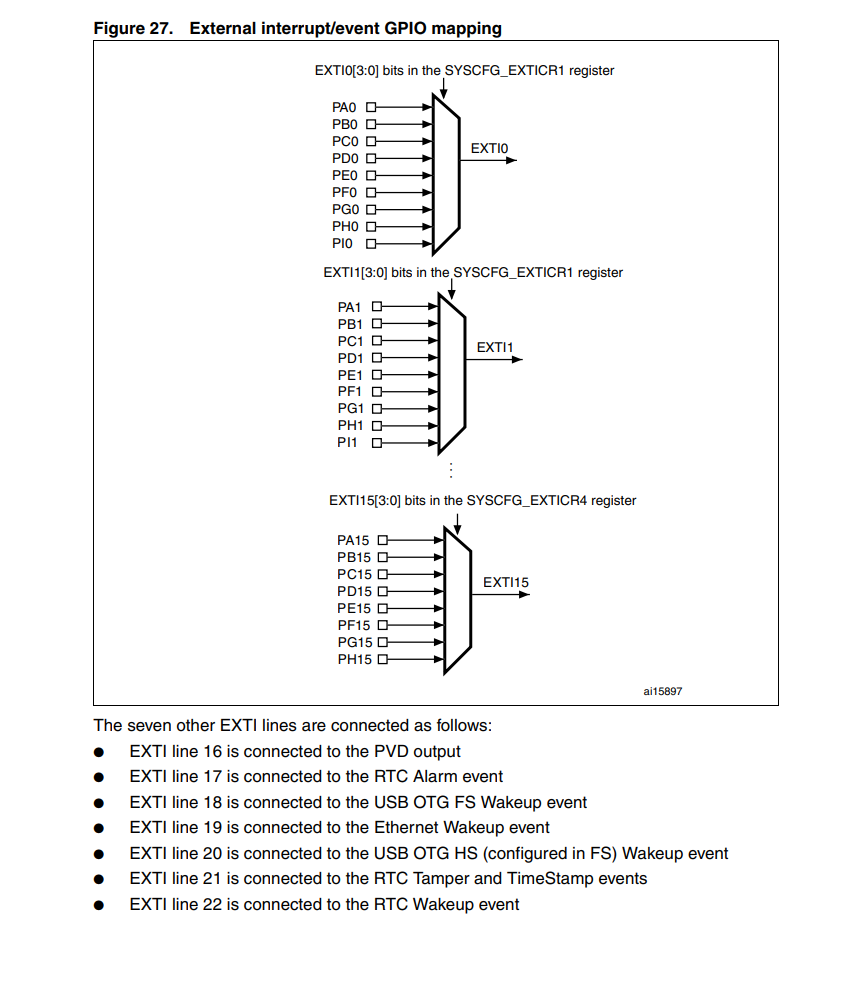


图 中断线

配置完毕之后，我们可以来看一下如何使用中断来响应按钮的事件，和之前一样，我们需要对按钮和LED进行初始化，对按钮的初始化需要一些修改，把模式改为下降沿触发，然后配置中断线0的优先级并启动。初始化完成后，我们还要编写中断回调函数，在函数中翻转LED的电平。

main.h代码如下：

/\* Define to prevent recursive inclusion -------------------------------------\*/

#ifndef \_\_MAIN\_H

#define \_\_MAIN\_H

/\* Includes ------------------------------------------------------------------\*/

#include "stm32f4xx\_hal.h"

/\* Exported types ------------------------------------------------------------\*/

/\* Exported constants --------------------------------------------------------\*/

/\* Exported macro ------------------------------------------------------------\*/

/\* Exported functions ------------------------------------------------------- \*/

#endif /\* \_\_MAIN\_H \*/

main.c如下：

#include"main.h"

//初始化LED灯

void led\_init() {

\_\_HAL\_RCC\_GPIOD\_CLK\_ENABLE();//时钟使能

GPIO\_InitTypeDef GPIO\_InitStruct;

GPIO\_InitStruct.Pin = GPIO\_PIN\_13;// LED3

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;//正常输出模式：推挽输出

GPIO\_InitStruct.Pull = GPIO\_NOPULL;//不拉

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;//低速模式即可

HAL\_GPIO\_Init(GPIOD, &GPIO\_InitStruct);

}

//初始化按钮

void button\_init() {

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE(); //时钟使能

GPIO\_InitTypeDef GPIO\_InitStruct;

GPIO\_InitStruct.Pin = GPIO\_PIN\_0;// Button

GPIO\_InitStruct.Mode = GPIO\_MODE\_IT\_FALLING;//下降沿触发中断

GPIO\_InitStruct.Pull = GPIO\_NOPULL;//不拉

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct);

HAL\_NVIC\_SetPriority(EXTI0\_IRQn, 2, 0);//设置最低中断优先级，避免阻塞其他中断

HAL\_NVIC\_EnableIRQ(EXTI0\_IRQn); //启用中断线0

}

//中断回调函数

void HAL\_GPIO\_EXTI\_Callback(uint16\_t GPIO\_Pin) {

if(GPIO\_Pin == GPIO\_PIN\_0) {

HAL\_GPIO\_TogglePin(GPIOD, GPIO\_PIN\_13);

}

}

int main(void) {

//初始化

HAL\_Init();

led\_init();

button\_init();

while (1) {}

}