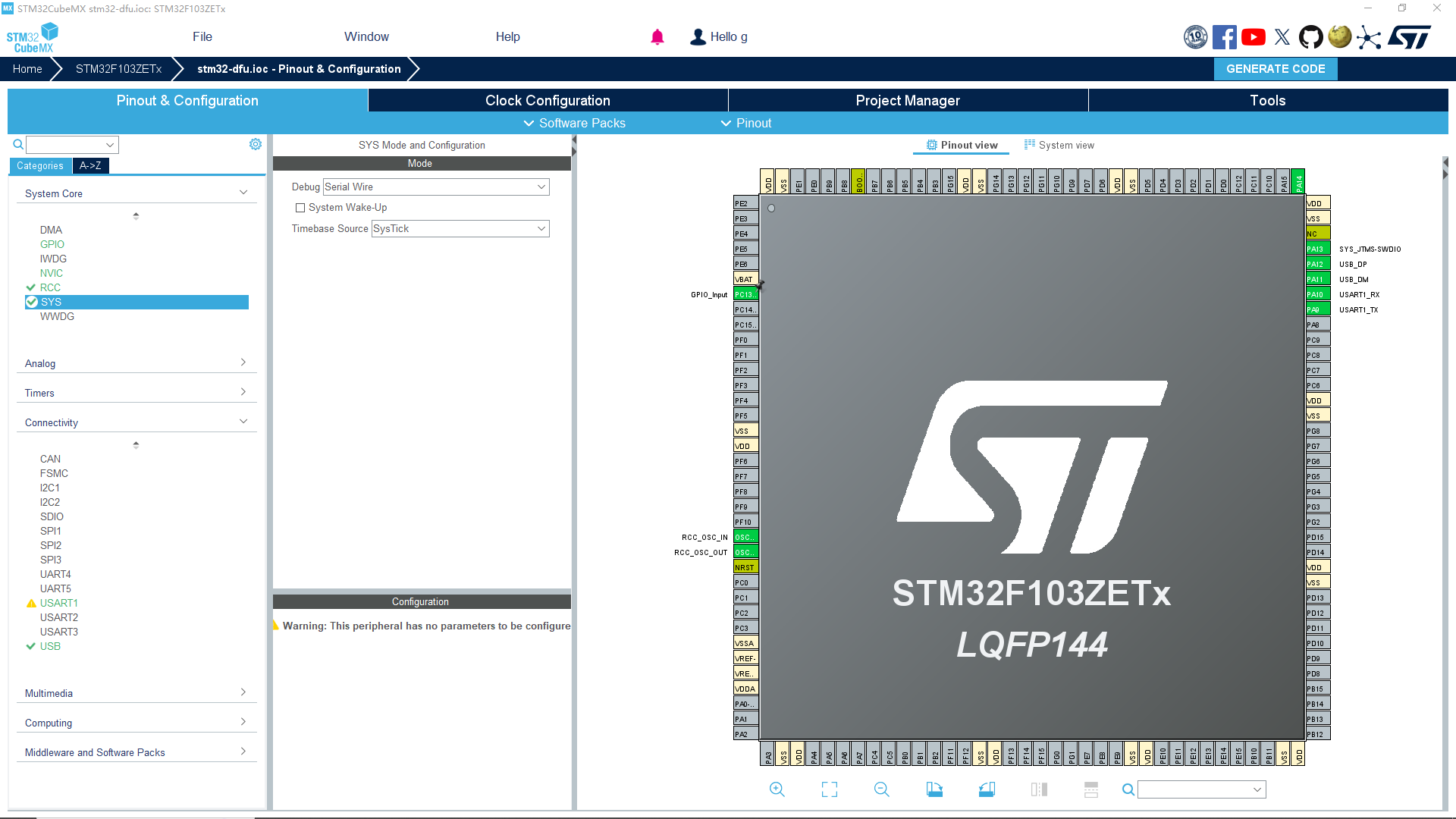
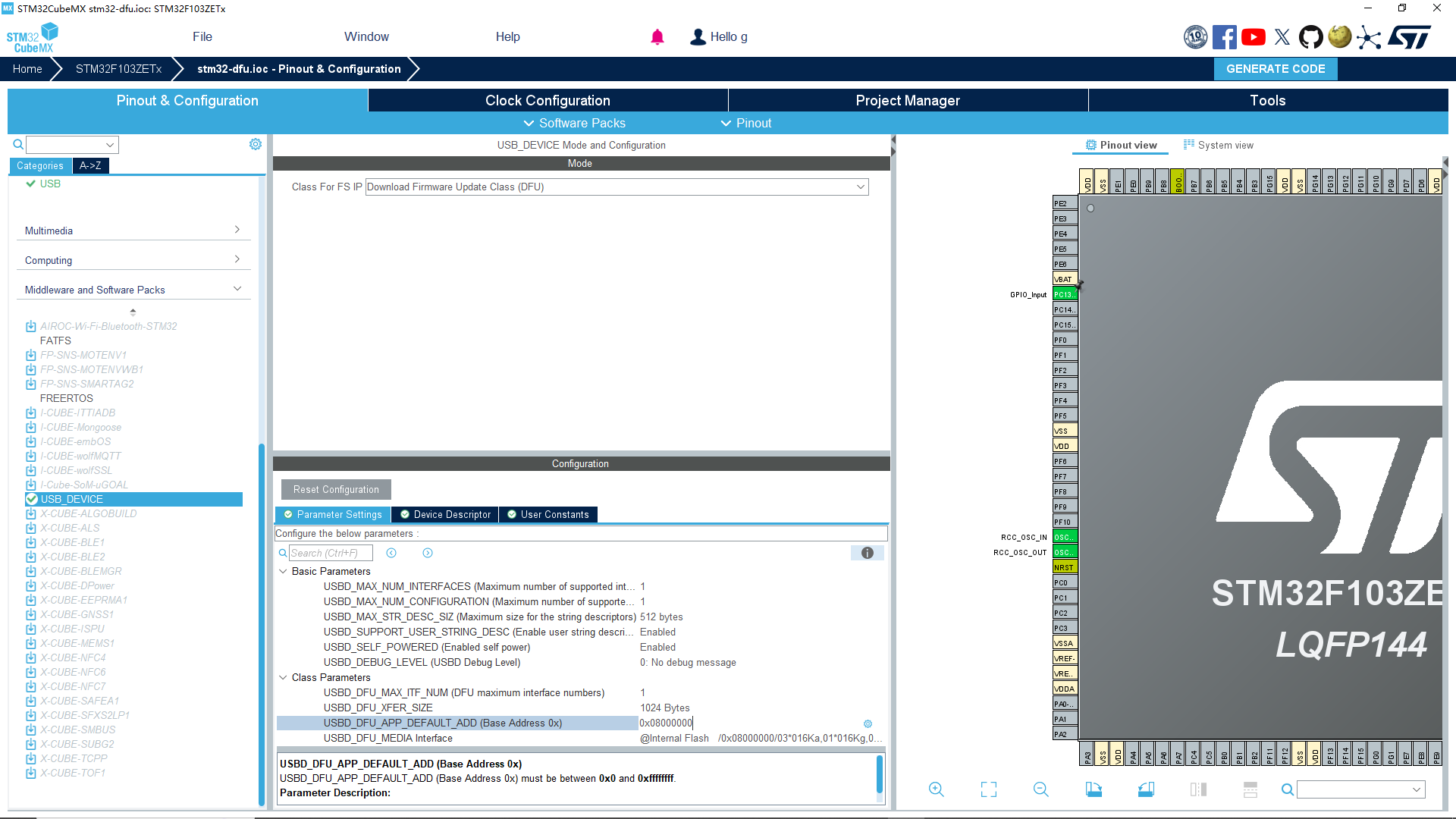
Stm32的usb提供dfu升级功能，有些芯片是内置的，可以直接通过dfu升级。有些没有内置dfu，可以通过usb设备提供的dfu类代码实现一个bootloader来更新应用程序。对于内置的dfu是不占用用户flash的，所以用户代码不需要做专门的修改。但是用户自己实现的dfu是要占用flash空间的，一般位于flash最开始的一部分，应用程序需要对应的往后挪，此时应用程序需要做一些修改，我们稍后介绍。

1. 通过cubemx生成基础的代码

usb dfu需要的外设不多，一般是一个gpio用作按键、一个gpio用作led、一个uart打印log，当然不能少了USB，甚至gpio和uart都可以不要。我这里用一个按键按下表示进入dfu升级，不按则进入应用程序，Uart打印log。



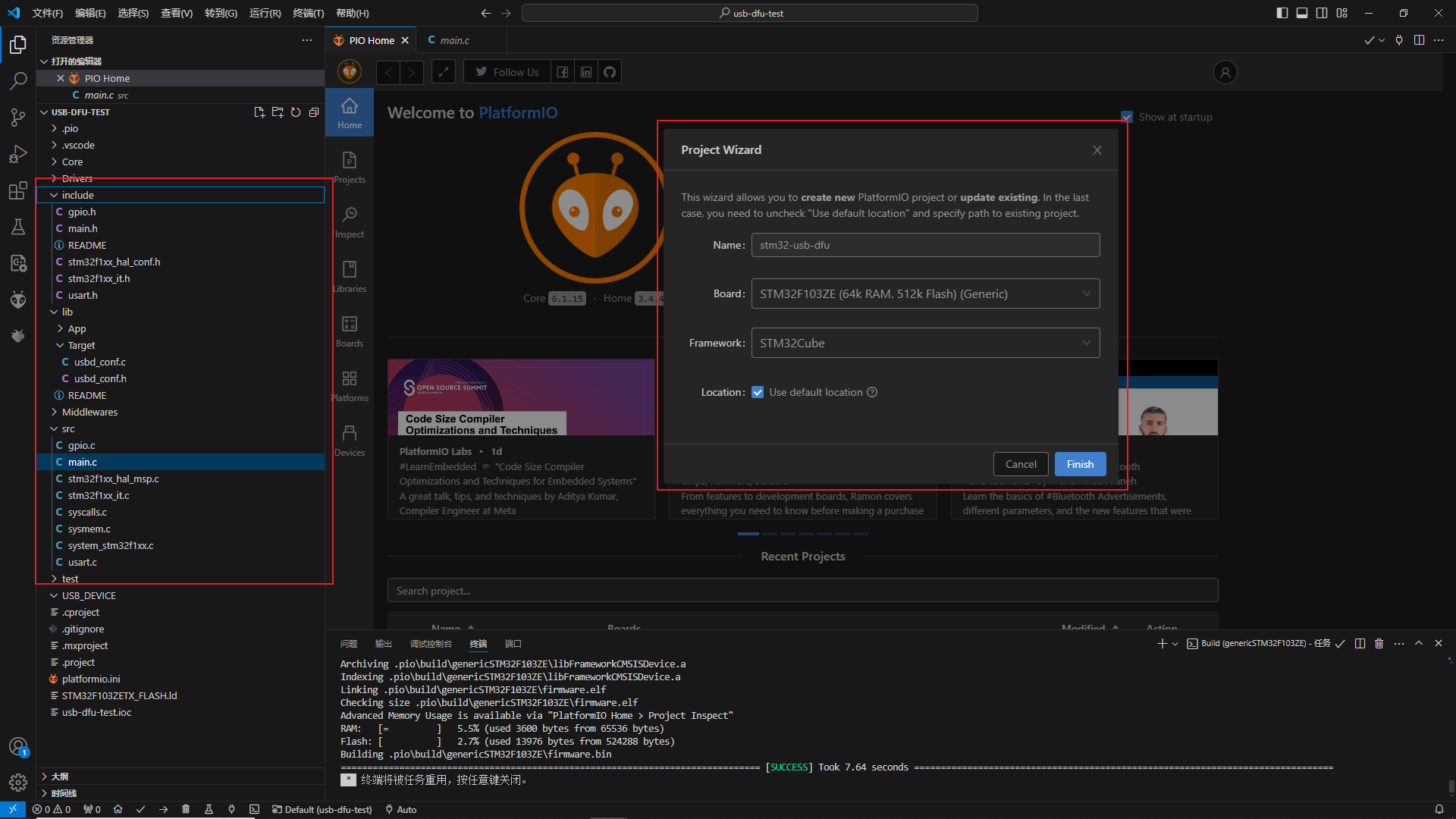
在Middleware and Software Packs下选择USB\_DEVICE，Mode选择DFU，ST这里的全称是Download Firmware Update，但是USB官方是Device Firmware Upgrade，首字母都一样，当然功能也一样。参数可以保持默认，后面在代码里改。



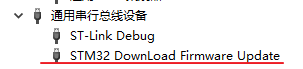
配置好时钟和工程相关导出代码。

1. 导入vscode开发

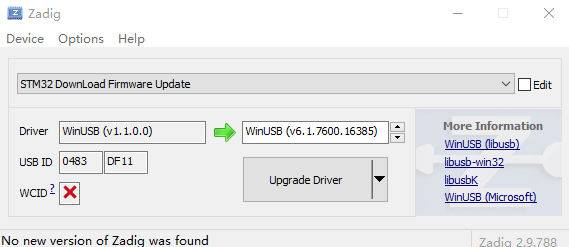
打开vscode新建platformio工程，打开工程。将Core\Inc下的文件放到include下，将Core\Src下的文件放到src下，将USB\_DEVICE下的两个文件夹放到lib下，下图我已经建好了，所以左边有代码。



此时编译下载后连接电脑应该就能识别为dfu设备了。

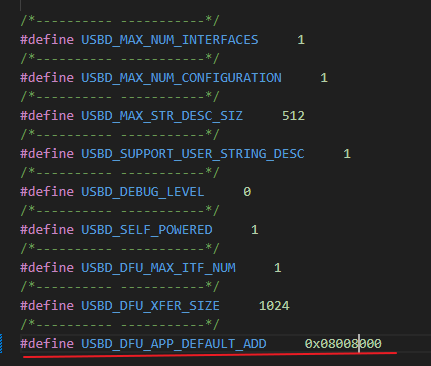


如果是一个带感叹号的设备，那可能是驱动问题，可以下载zadig安装驱动。

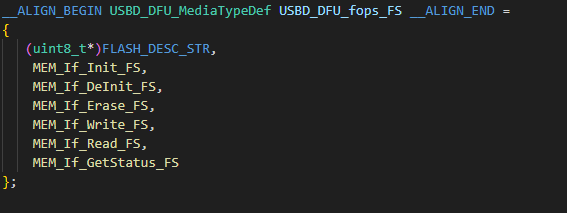


接下来我们修改代码，完善功能。

需要用户修改的usb相关文件包括：usbd\_conf.h、usbd\_dfu\_if.c.。在usbd\_conf.h中有一些usb相关的宏定义，我们需要修改USBD\_DFU\_APP\_DEFAULT\_ADD，这个指明用户程序的起始地址，这里我改为0x08008000，给bootloader预留32k（0x8000）的空间，这个可以根据实际情况调整，但必须为0x200的倍数，向量表的偏移对这个值有要求。



usbd\_dfu\_if.c是dfu类提供给用户的接口，需要用户根据自己的需求实现。



第一个成员是一个字符串宏定义，描述了内部flash的起始地址以及存储分配布局情况。

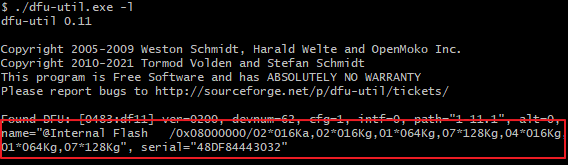


上面的大致意思是Flash地址为0x08000000，其中前3个16k的区域为只读，之后依次为1个16k、1个64k、7个128k、4个16k、1个64k、7个128k区域为可读可写可擦除。这些内容在cubemx中相关区域有介绍，有兴趣的可以去进一步了解。

还记得吗，我们的应用程序起始地址是0x08008000，0x8000是32k，位于了只读区域（3x16k=48k），所以需要修改一下这里，将3改为2，将后边的1改为2.。

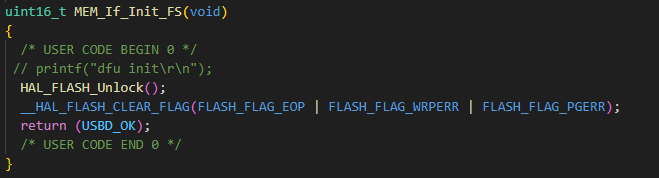


这个字符串信息会在电脑的应用程序中用到。

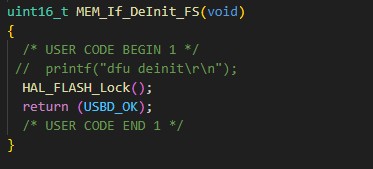


下面具体实现操作flash读写擦除相关接口函数。

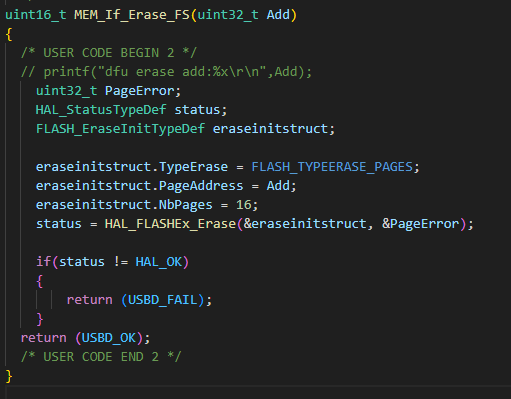
1、初始化函数解锁flash，清除一些标志，为操作flash做准备；



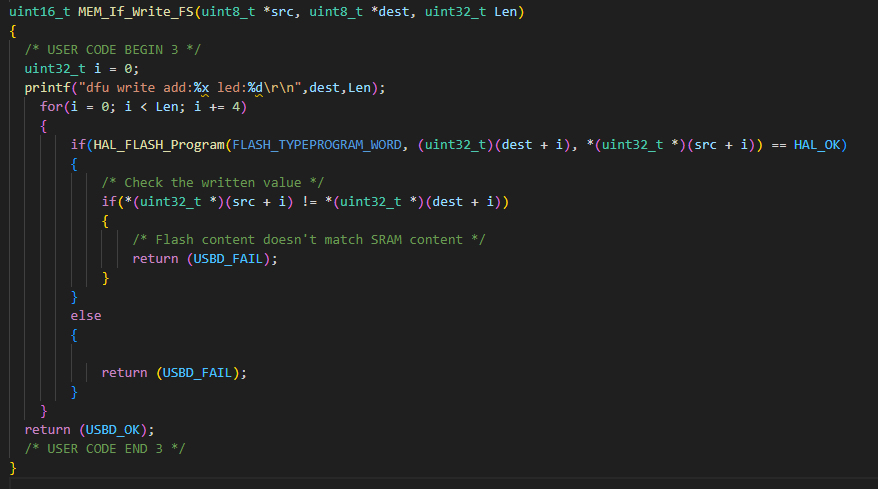
2、deinit锁定flash，防止误操作；



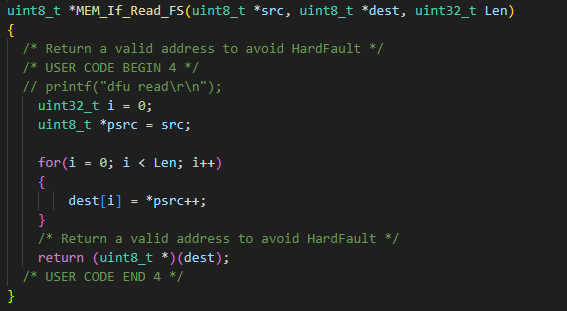
3、flash在写之前必须擦除，这个函数只传入了地址，没有长度，所以擦除的页数要保证覆盖整个应用程序，当然为了保险可以擦除整个应用flash区域，就是费时间了点。实测发现这个函数会多次进入，我还没搞清楚这里详细的机制，这里实际可能同一个区域多次擦除了，后续研究明白了再补充吧。



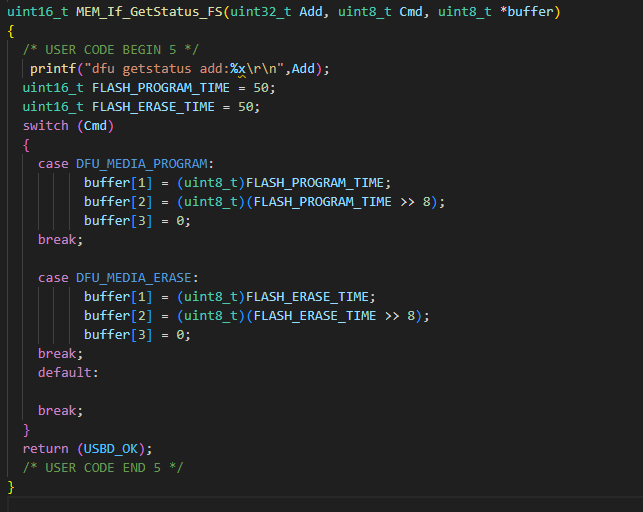
4、写函数写入我们上传的固件到flash，实现程序的升级。



5、读函数读取flash内容

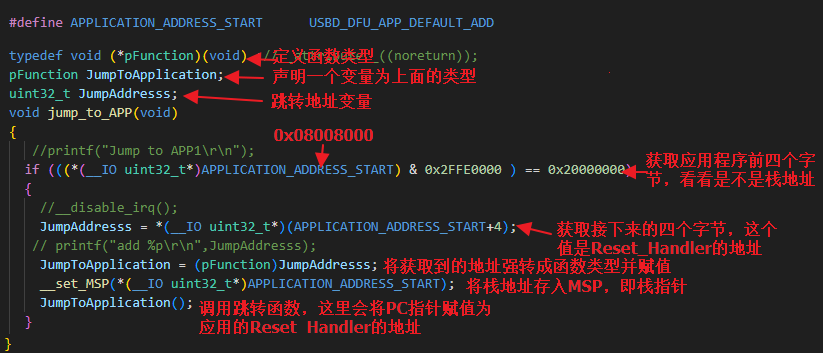


6、获取状态函数获取操作过程中的状态，这里可以自定义擦除和编程的时间，buffer[1-3]保存以毫秒为单位的时间值。实测这个函数也会多次调用，但这个时间具体怎么影响程序还没仔细研究，后续补充吧。



到此，bootloader已经能够实现程序的升级了，但是主程序还什么都没有，即便升级成功了也办法跳到应用程序，即便复位也还是执行升级程序，所以接下来我们实现跳转程序。

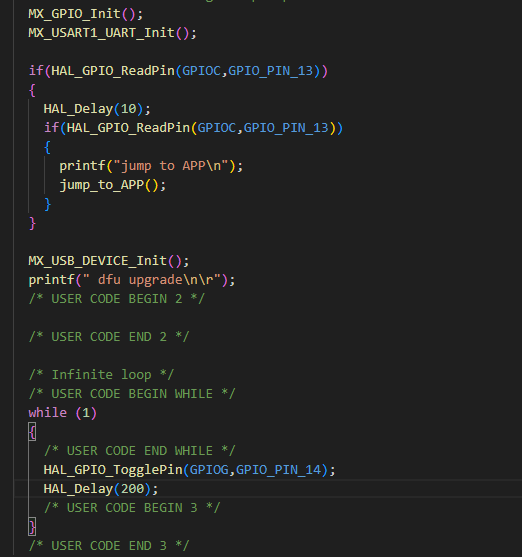
先实现跳转函数，



这里其实就做了两件事，重设栈指针和PC，一切OK就能跳到应用程序了。如果指定的地址没有程序，或者不是栈地址就直接进入dfu状态。

这里注意，在该函数内部调用printf会导致跳转失败，此时如果给typedef void (\*pFunction)(void)加\_\_attribute\_\_((noreturn))声明后又可以了。所以，调试的时候可以加上\_\_attribute\_\_((noreturn))以便查看log。

在主程序里，先初始化gpio和usart，然后看有没有按下按键，我这里按下是低电平，所以没按if成立，进入跳转程序，如果按下，if不成立，开始执行后边USB设备的初始化并快闪一个指示灯，等待程序的升级。



到这里usb dfu bootloader已经搞定了，下面我们需要做一个对应的应用程序。

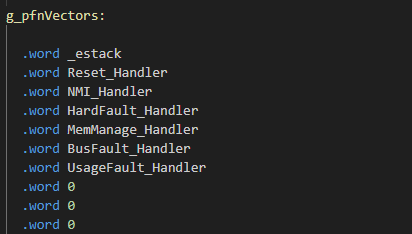
注意！！！！！！

注意！！！！！！

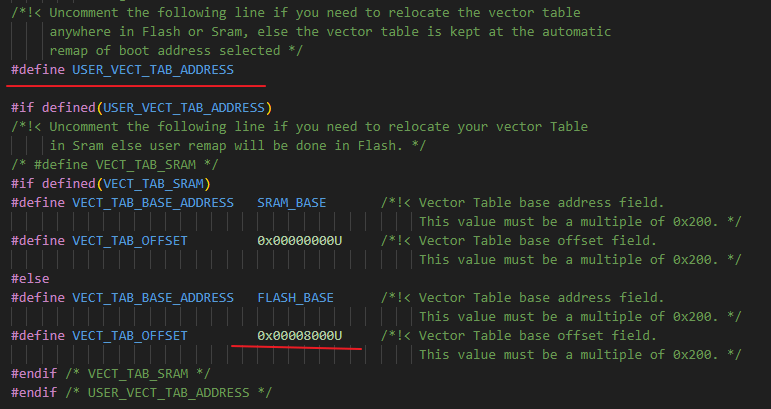
注意！！！！！！

下面的修改是应用程序，千万别搞错了。

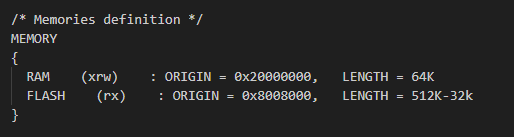
应用程序需要修改两个地方，一个是中断向量表偏移。默认中断向量表位于0x00000000地址处。



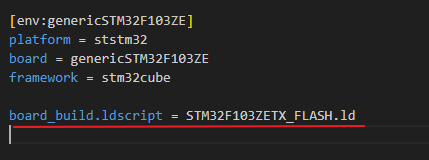
有了bootloader后，应用程序偏移了，所以中断向量表也相应偏移了，所以应用程序需要指明偏移地址，否则程序按照默认的跳转就会出问题。这个修改在system\_stm32f1xx.c文件中，宏定义#define USER\_VECT\_TAB\_ADDRESS默认是注释掉的，去掉注释，然后修改宏#define VECT\_TAB\_OFFSET的值为0x00008000U，这是我们设计bootloader时定的值。这里的注释注明了这个值必须是0x200的倍数。



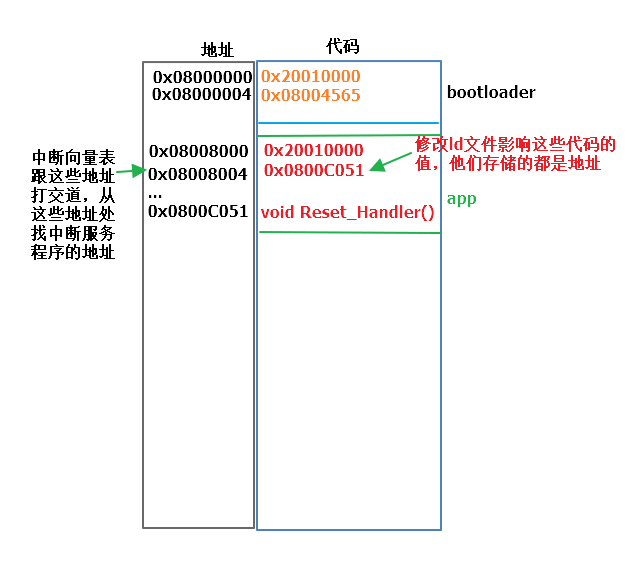
另一个需要修改的是链接文件，工程中有个STM32F103ZETX\_FLASH.ld文件，这个文件内部定义了程序的分布情况，同样需要修改flash的起始地址为0x08008000，告诉链接器程序从哪个地址开始的，以便为函数、变量重定位地址。



如果用的platformio，需要在platformio.ini中指定一下ld文件，否则用的是platformio自己下载的库中的ld文件。



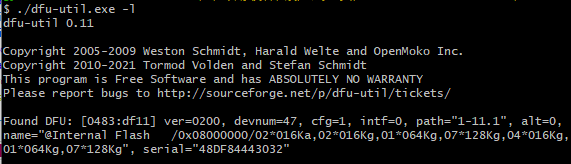
到此，我们所有的修改都完成了，总结一下，修改向量表偏移是告诉程序从哪里找向量表，修改ld文件中flash起始地址告诉编译器程序存在哪里，以便链接的时候以此为基础生成代码。有没有很绕，我刚开始晕死了，后来我每改一个条件生成一个bin文件然后看二进制才搞懂了，你要是也不太明白可以看看bin文件的变化，你会明白的。



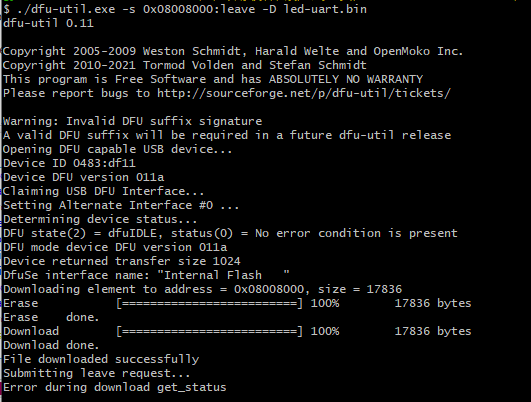
编译好准备下载吧。让我很不爽的是st官方的两个工具都不能用，dfusedemo完全找不到usb设备，stm32cubeprogrammer可能添加了新的东西，能读不能擦写。所以，我找了dfu-util这个命令行工具，这个工具就能用。看来st官方还添加了其他东西，后续研究好了再补充。

这里只演示最基本的操作，其他的大家自己尝试吧。

./dfu-util.exe -l 列出dfu设备

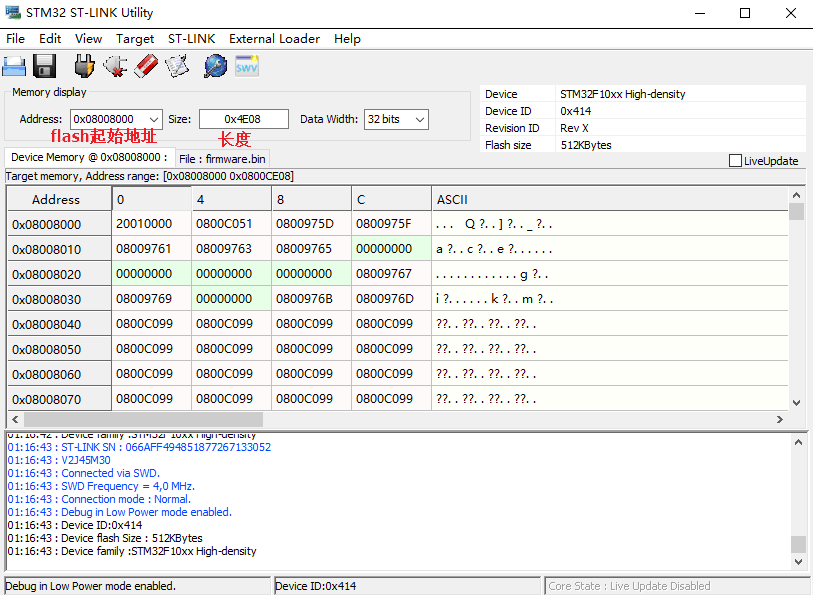


./dfu-util.exe -s 0x08008000:leave -D led-uart.bin 这个意思是在0x08008000地址处下载led-uart.bin这个文件，leave表示下载完退出dfu模式，其实就是复位了系统，具体进不进应用程序还得看主程序设计。



到此，我们基于stm32 usb dfu的bootloader基本上就完成了，其实通过这个实验我们可以实现其他各种接口的bootloader，uart、iic、spi、can、甚至可以通过一个gpio实现（电调就是单线升级），原理都一样，只不过是获取数据的形式有区别罢了。这个实验还是挺综合的，能够学到不少东西，建议学习嵌入式的小伙伴都做做，一定会受益匪浅。

附：调试的时候可能会用到stlink等调试器，用调试器可以很方便的读取flash，这样就可以很方便的验证我们做的对不对。



同样，我们可以将程序下载到指定的地址处，如果确定应用程序OK，我们可以把应用程序直接下载到0x08008000地址处，然后调试跳转函数。

