##### # GPIO

GPIO（General-purpose input/output），通用型输入输出口的简称。

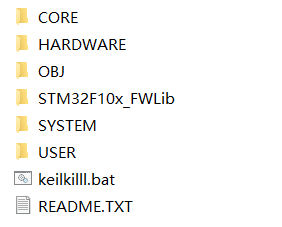
单片机需要实现功能，不可能只靠自己的芯片，是需要和其他部分一起合作的，这个地方就是到时候用线接起来的地方。

软件编程能够作用到硬件上是通过控制寄存器来完成功能的。

寄存器就是最基本的开关。寄存器的功能是存储二进制代码，它是由具有存储功能的触发器组合起来构成的。

触发器是一种具有两种稳态的用于储存的组件，可记录二进制数字信号“1”和“0”。一个触发器可以存储1位二进制代码，故存放n位二进制代码的寄存器，需用n个触发器来构成。

文件夹组成



- core存放固件库的核心文件和启动文件，不需要修改。

- hardware，存放的是外部硬件相关的驱动函数。

- 每一个源文件对应一个头文件。

- 他实现功能靠的是调用FWlib下面的固体库文件实现的。

- OBJ，存放编译过程文件和hex文件。

- C语言编译过程，预编译，编译，汇编，链接。

- 预编译：将所有的#include头文件以及宏定义替换成其真正的内容

- 编译：经过预处理之后的程序转换成特定汇编代码(assembly code)的过程

- 汇编：将上一步的汇编代码转换成机器码(machine code)

- 链接：将多个目标文以及所需的库文件(.so等)链接成最终的可执行文件(executable file)

- .so文件(动态链链接库）

- .o文件又称对象文件，是可执行文件。

- hex文件就是一种可以被单片机执行的文件。一般记事本也可以打开。

- FWlib，即firmware-library。存放的是ST官方提供的外设库函数源码文件。

- system提供的是共用代码，包括延时函数、串口函数等等。

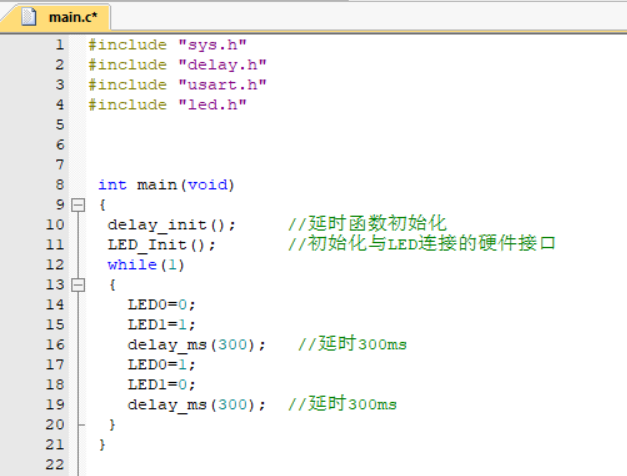
- USER存放的主要是main用户代码。其中后面是x的主要是系统时钟初始化函数的相关定义。

- stm32f10x.h包含了寄存器地址和结构体类型定义。后面是it的表示的是部分的中断服务函数，即interrupt。

- readme，注意事项。

软件的层次差不多就是，user -- hardware -- fwlib -- 寄存器。

main.c文件



因为main.c里包含了main函数，这个主函数是最重要的。

最上方是调用头文件。文件头的调用符号不同，冒号和尖括号。冒号表示查找顺序，先当前目录，先系统目录。

头文件里一般包含常数，函数，宏定义，变量声明。

main.c文件中，系统调用main主函数，main主函数调用其他函数，函数返回，最后由主函数结束。

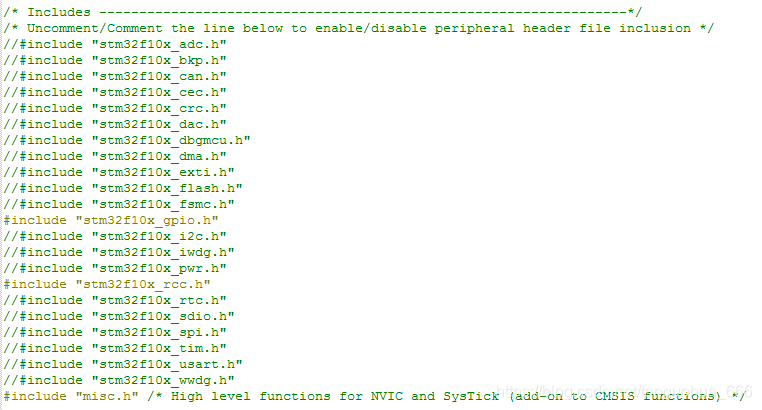
函数的形式

return\_type function\_name( parameter list )

{

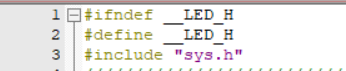
body of the function;

}

开启多余外设是增加功耗而没用。注意到conf.h管理所有库的头文件。

当多个文件需要包含各种外设头文件的时候，每个文件逐个添加头文件有点繁琐，把头文件全部通过include组织在一个文件中，并且先全部注释掉，工程中需要哪些头文件就把这些头文件进行去注释。

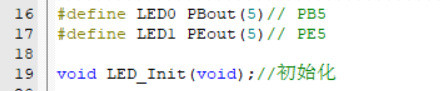
这个文件除了将stm32f10x系列芯片涉及到的外设头文件全部组织在一起之外，还有另外的断言语句。



if not defined条件编译，防止重复包含头文件时报错。

双下划线，避免这个宏标识符和其他定义重名。

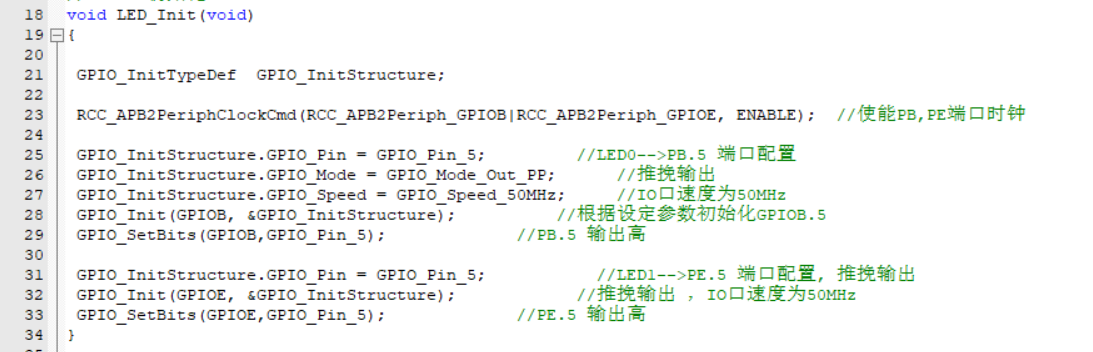
头文件里主要包含两个内容，宏定义和函数声明。



宏定义用来将一个标识符定义为一个字符串

函数声明给出了函数名、返回值类型、参数列表（重点是参数类型）等与该函数有关的信息，也称为函数原型宏定义

控制LED的GPIO步骤应该是引脚，功能模式，高低电平。



首先是定义了一个结构体的数据类型的定义，

- 开启了外设时钟，

- 选择了控制的引脚，

- 设置了引脚模式，

- 设置了引脚速率，

- 调用库函数初始化，

- 设置电平高低。

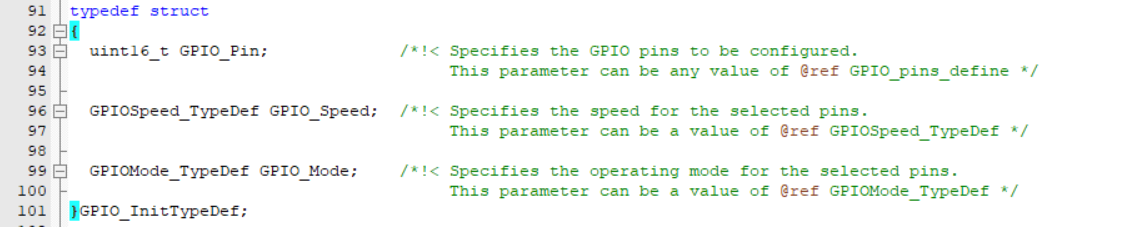
结构体。

结构体(struct)是由一系列具有相同类型或不同类型的数据构成的数据集合，也叫结构。

几种数据类型一起来修饰某个变量。

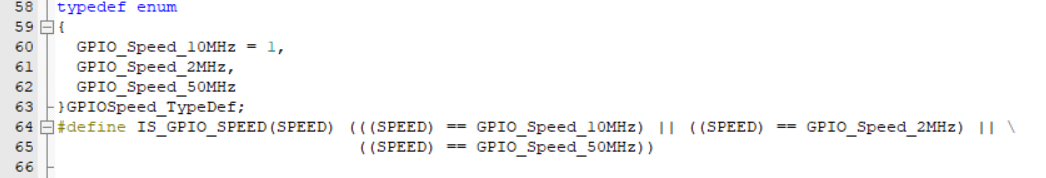
将不同类型的数据存放在一起，作为一个整体进行处理。

从结构体原型，得知三个成员，分别是引脚，最高频率，工作状态。



u8. u16，u32，无符号字符型，短型，长整型。

枚举，一个集合，变量的可取值范围。不赋值自动根据前一个加1。



- 引脚速度又叫输出驱动电路的响应速度。

- 主要控制输出速度。

- 选择的目的：保证效果的前提，降低噪声和功耗，电磁干扰。

- 比如USART，最大波特率115200，使用2M就可以，保证效果省电功耗小。

- 输出速度和配置速度不匹配，可以明显看出输出波形是不正常的。

- 工作模式。

- 模拟输入，一般用于ADC功能。连续量到离散量的转变。

- 浮空输入，如果没有连接其他设备，电平不定，高或者低都可能。可以做KEY的识别。

- 上拉输入下拉输入，就是浮空的两个边界。

- 所有端口都有外部中断能力。为了使用外部中断线，端口必须配置成输入模式。当我们闭合上拉电阻的开关，断开下拉电阻的开关的时候，也就是此时为上拉通路导通。根据我们最基础的电路知识，如果I/O口断开不接外设，或者接了外设但是电路也断开，此时整条回路是断路的时候，电阻相当于不存在，此时O点的电位，也就被钳制在了VDD(供电电压正极)，于是O点就处于高电平。同理，闭合上拉电阻的开关，断开下拉电阻的开关，那么在电路断开的时候，根据电路知识，电阻相当于不存在，O点的电压就就被钳制在了VSS(GND)，也就是O点处于低电平。

- 开漏输出，即漏级开路输出。设置1，开漏浮空，设置0，拉低电平。

- 推挽输出，输出高电平和低电平时，都具有驱动能力。

- 复用开漏，片内外设的TX，MOSI。

- 复用和普通的区别就在于输出的是数据寄存器中的数据，还是来自外设的数据。

- 复用推挽，片内外设如IIC的SCL，SDA

led.c文件

IO口时钟。

在配置 STM32 外设的时候，任何时候都要先使能该外设的时钟。

通过右键选择跳进了他的定义里可以发现，这个gpio普遍都是挂靠在apb2总线上的，在固件库中对挂载在 APB2 总线上的外设时钟使能是通过函数 RCC\_APB2PeriphClockCmd()来实现的。

所以说如果要开启时钟的话就是直接选择宏定义的这一句。

. 表示结构体的成员。

STM32F1有很多的内置外设，这些外设的外部引脚都是与 GPIO 复用的。

也就是说，一个 GPIO如果可以复用为内置外设的功能引脚，那么当这个 GPIO 作为内置外设使用的时候，就叫做复用。

STM32 的每个 IO 端口都有 7 个寄存器来控制。

他们分别是：配置模式的 2 个 32 位的端口配置寄存器 CRL 和 CRH；2 个 32 位的数据寄存器 IDR 和 ODR；1 个 32 位的置位/复位寄存器BSRR；一个 16 位的复位寄存器 BRR；1 个 32 位的锁存寄存器 LCKR。

7个和gpio相关的寄存器可以分成4类，分别是配置，数据，位控制和锁定。

配置的意思就是说选定Gpio的特定功能，最基本的就是选择是输入还是输出。

数据的意思就是保存了输入的电平或者是说要输出的电平。

位控置的意思就是说设置某个引脚的数据是一还是零，控制输出的电平。

锁定寄存器就是设置之后就不能修改他的配置。

每个I/O端口位可以自由编程，然而I/0端口寄存器必须按32位字被访问(不允许半字或字节访问)。GPIOx\_BSRR和GPIOx\_BRR寄存器允许对任何GPIO寄存器的读/更改的独立访问；这样，在读和更改访问之间产生IRQ时不会发生危险。

因为 GPIOB 和 GPIOE 的 IO 口的初始化参数都是设置在结构体变量GPIO\_InitStructure 中，因为两个 IO 口的模式和速度都一样，所以我们只用初始化一次，在GPIOE.5 的初始化的时候就不需要再重复初始化速度和模式了。

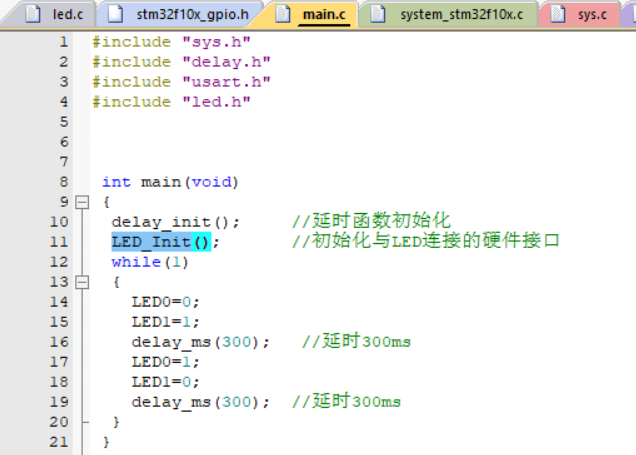
GPIO 相关的函数我们先讲解到这里。

虽然 IO 操作步骤很简单，这里我们还是做个概括性的总结，操作步骤为：

- 1） 使能 IO 口时钟。调用函数为 RCC\_APB2PeriphClockCmd()。

- 2） 初始化 IO 参数。调用函数 GPIO\_Init();3） 操作 IO。操作 IO 的方法就是上面我们讲解的方法。

main.c文件



main()函数非常简单，先调用 delay\_init()初始化延时，接着就是调用 LED\_Init()来初始化GPIOB.5 和 GPIOE.5 为输出。

最后在死循环里面实现 LED0 和 LED1 交替闪烁，间隔为 300ms。

操作 IO 口输出高低电平的三种方法。

通过位带操作 PB5 输出高低电平从而控制 LED0 的方法如下：

LED0=1; //通过位带操作控制 LED0 的引脚 PB5 输出高电平

LED0=0; //通过位带操作控制 LED0 的引脚 PB5 输出低电平

库函数操作方法如下：GPIO\_SetBits(GPIOB, GPIO\_Pin\_5); //设置 GPIOB.5 输出 1,等同 LED0=1;

GPIO\_ResetBits (GPIOB, GPIO\_Pin\_5); //设置 GPIOB.5 输出 0,等同 LED0=0;

库函数操作就直接调用两个函数即可控制 IO 输出高低电平。

我们也通过直接操作寄存器BRR 和 BSRR 的方式来操作 IO 口输出高低电平，方法如下：GPIOB->BRR=GPIO\_Pin\_5; //设置 GPIOB.5 输出 1,等同 LED0=1;

GPIOB->BSRR=GPIO\_Pin\_5; //设置 GPIOB.5 输出 0,等同 LED0=0;

编译部分

- Translate是编译当前改动的源文件，在这个过程中检查语法错误。但并不生成可执行文件

- Build是只编译工程中上次修改的文件及其它依赖于这些修改过的文件的模块，同时重新链接生成可执行文件。如果工程之前没编译链接过，它会直接调用Rebuild All。另外在技术文档中，Build实际上是指increase build，即增量编译。

- Rebuild是不管工程的文件有没有编译过，会对工程中所有文件重新进行编译生成可执行文件，因此时间较长。

可以看到没有错误，也没有警告。

这里我们解释一下，编译结果里面的几个数据的意义：

- Code：表示程序所占用 FLASH 的大小（FLASH）。

- RO-data：即 Read Only-data，表示程序定义的常量，如 const 类型（FLASH）。

- RW-data：即 Read Write-data，表示已被初始化的全局变量（SRAM）

- ZI-data：即 Zero Init-data，表示未被初始化的全局变量(SRAM)

有了这个就可以知道你当前使用的 flash 和 sram 大小了。所以，一定要注意的是程序的大小不是.hex 文件的大小，而是编译后的 Code 和 RO-data 之和。