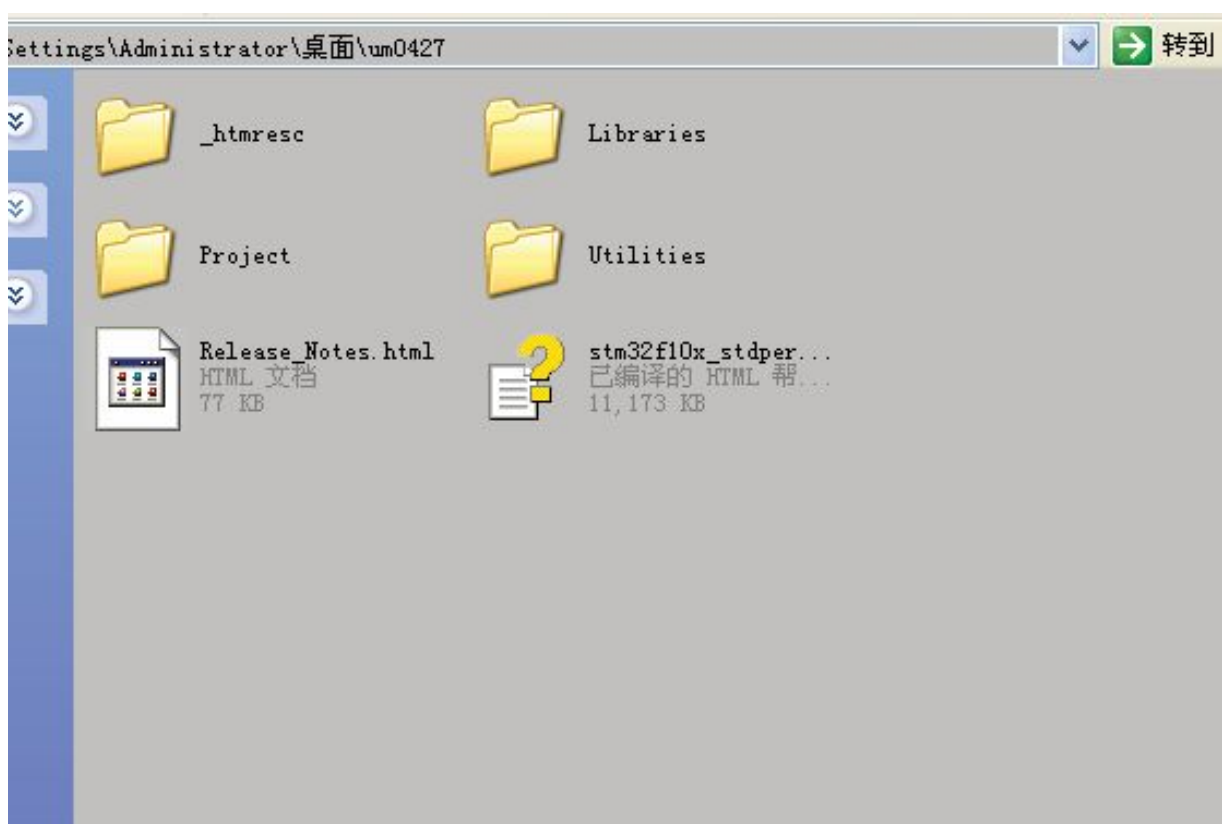


详解用 STM32 官方库来发自己的程序

这里用的库是 ST3.0.0 版本，解压缩后的文件名为 um0427, 库的源代码可以从 ST 的官网下载。

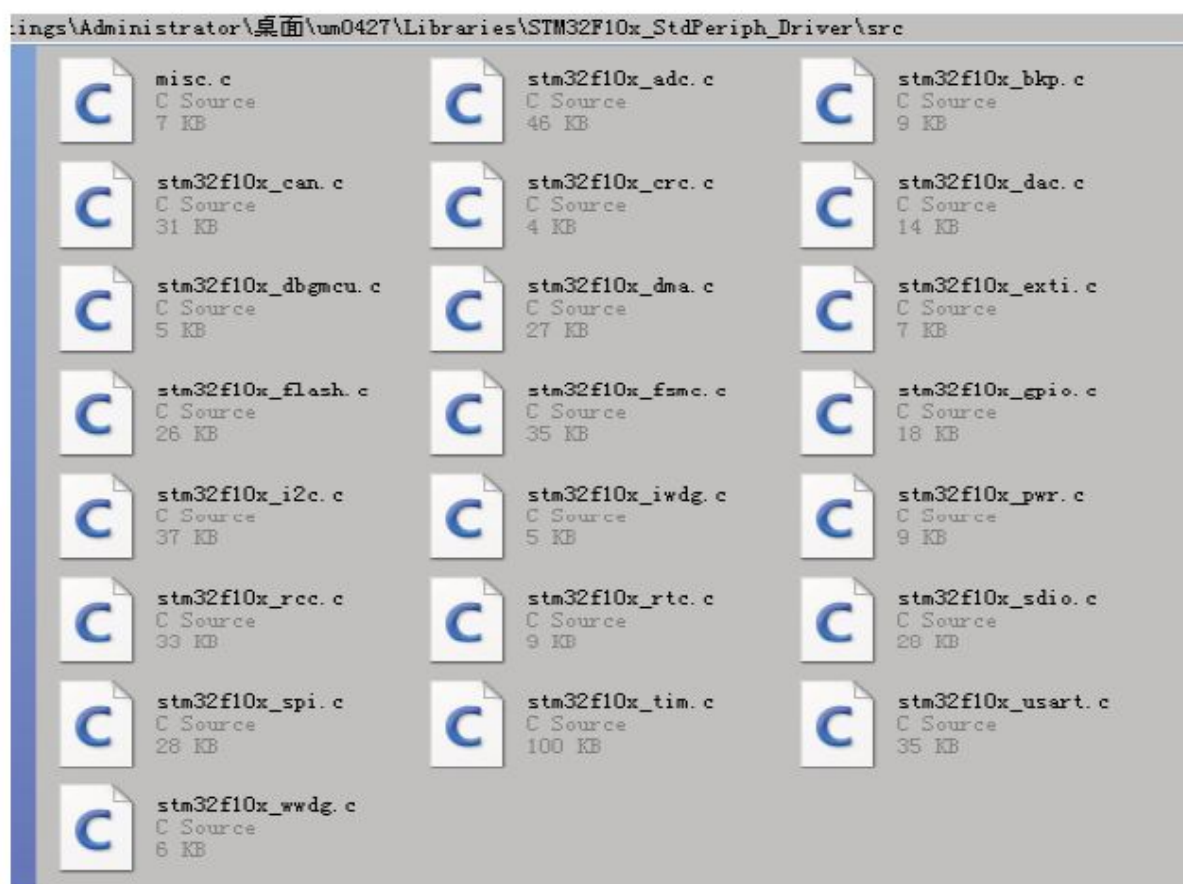
首先来分析下这个库的目录结构，如下图所示：



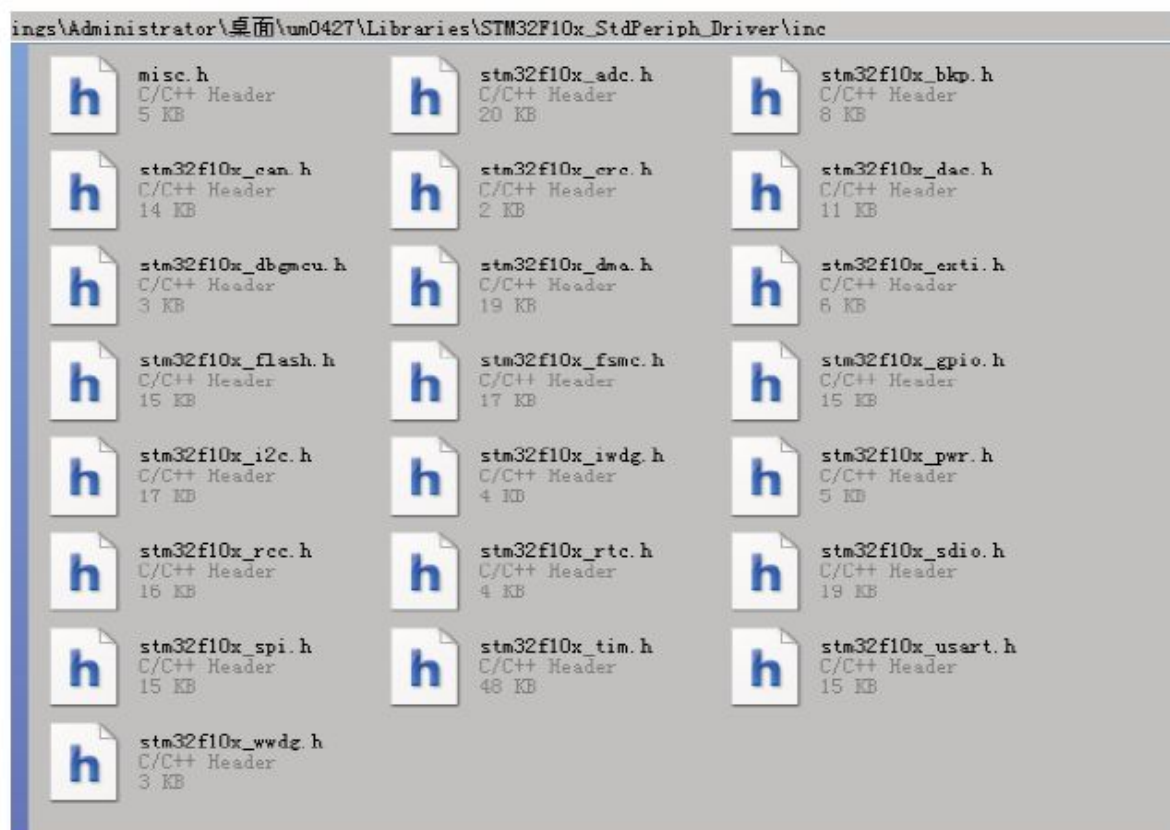
Libraries 文件夹下是驱动库的源代码跟启动文件。**Project** 文件夹下是用驱动库写的例子跟一个工程模板。还有一个已经编译好的 HTML 文件，主要讲的是如何使用驱动库来编写自己的应用程序，说得形象一点，这个 HTML 就是告诉我们：ST 公司已经为你写好了每个外设的驱动了，想知道如何运用这些例子就来向我求救吧。既然 ST 给我们提供的美味大餐（驱动源码）就在眼前，我又何必去找品尝大餐的方法呢，还不如直接一头直接扎进大餐中，大吃一顿再说（直接阅读库的源码）。但当我们吃的有点呛口的时候回去找下方法还是很好的。其他三个文件作用不大，我们可以不用管它。

接下来我们重点来分析下 **Librarie** 文件夹下的内容。

Libraries\STM32F10x_StdPeriph_Driver 文件夹下有 inc（include 的缩写）跟 src（source 的简写）这两个文件，src 里面是每个片上外设的驱动程序，这些外设当中很多是芯片制造商在 Cortex-M3 核上加进去的，Cortex-M3 核自带的外设是通用的，放在 CMSIS 文件夹下。如下图所示：



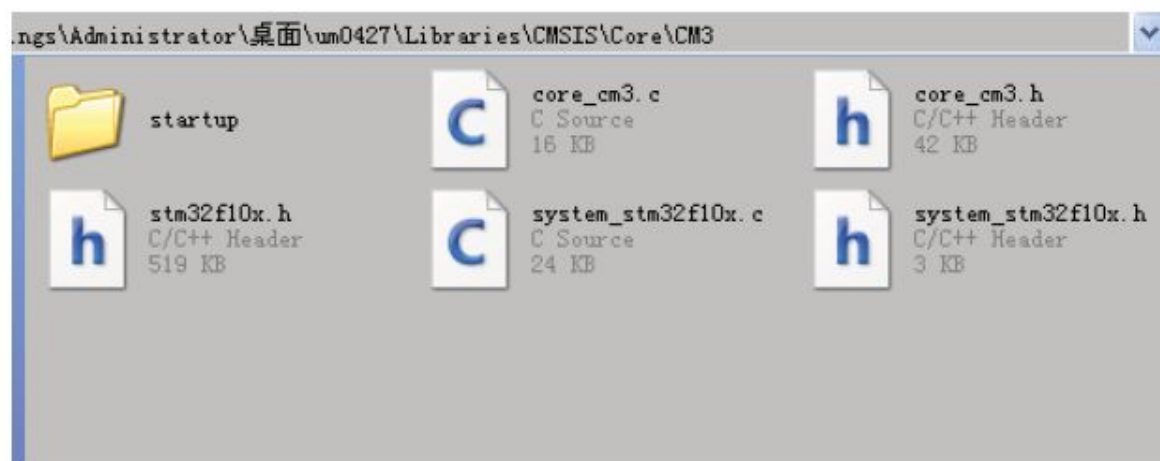
Libraries\STM32F10x_StdPeriph_Driver\inc 文件夹下是每个驱动文件对应的头文件。当我们的应用程序需要用到某个外设的驱动程序的话只需将它的头文件包含进我们的应用程序即可。



Libraries\CMSIS\Core\CM3\startup\arm 文件夹下是三个汇编编写的系统启动文件，分别对应于小（LD）中（MD）大（HD）容量 Flash 的单片机，在我们新建工程的时候需要将它包含到我们的工程中去。启动文件是任何处理器在上电复位之后最先运行的一段汇编程序。启动文件的作用是：1、初始化堆栈指针 SP，2、初始化程序计数器指针 PC，3、设置异常向量表的入口地址，4、配置外部 SRAM 作为数据存储器（这个由用户配置，一般的开发板可没有外部 SRAM），5、设置 C 库的分支入口 __main（最终用来调用 main 函数）。如若要详细了解启动文件的详细过程可参考如下网友的文章：<http://blog.ednchina.com/likee/138130/message.aspx>。搞不明白也没太大关系，我们新建工程的时候只要将它包含进来就可以了。



Libraries\CMSIS\Core\CM3 文件夹下除了放有 startup 启动文件外，还有这几个文件 core_cm3.c 、 core_cm3.h ， stm32f10x.h 、 system_stm32f10x.c ， system_stm32f10x.h。

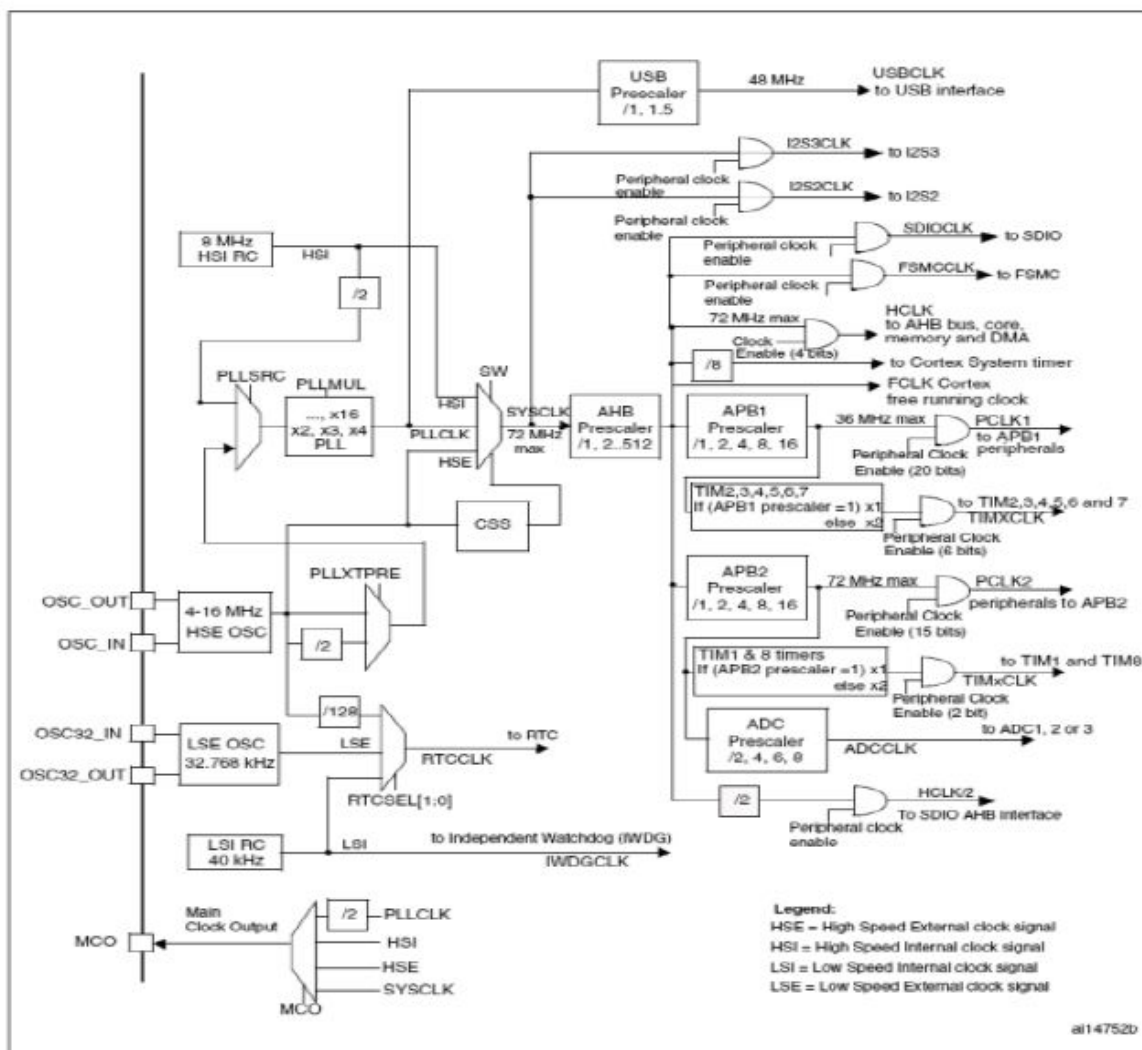


core_cm3.c 是 CMSIS Cortex-M3 核外设接入层的源文件，在所有符合 CMSIS 标准的 Cortex-M3 核系列单片机都适用，独立于芯片制造商，由 ARM 公司提供。它的作用是为那些采用 Cortex-M3 核设计 SOC 的芯片商设计的芯片外设提供一个进入 M3 内核的接口。至于这些功能是怎样用源码实现的，我们可以不用管它，我们只需把这个文件加进我们的工程文件即可。该文件还定义了一些与编译器相关的符号。在文件中包含了 **stdin.h** 这个头文件，这是一个 ANSI C 文件，是独立于处理器之外的，就像我们熟知的 C 语言头文件 **stdio.h** 文件一样。位于 RVMDK 这个软件的安装目录下，主要作用是提供一些类型定义，如：

```
036      /* exact-width signed integer types */
037 typedef      signed          char int8_t;
038 typedef      signed short    int int16_t;
039 typedef      signed          int int32_t;
040 typedef      signed          __int64 int64_t;
041
042      /* exact-width unsigned integer types */
043 typedef unsigned          char uint8_t;
044 typedef unsigned short    int uint16_t;
045 typedef unsigned          int uint32_t;
046 typedef unsigned          __int64 uint64_t;
047
```


core_cm3.c 跟启动文件一样都是底层文件，都由 ARM 公司提供，遵守 CMSIS 标准，即所有 CM3 芯片的库都带有这个文件，这样软件在不同的 CM3 器件的移植工作就得以化简。**core_cm3.c** 里面包含了一些跟编译器相关的信息，如：RealView Compiler, ICC Compiler, GNU Compiler。**core_cm3.h** 这个文件实现了 CM3 内核里面的 NVIC 和 SysTick 这两个资源的所有功能，NVIC 是嵌套向量中断控制器，SysTick 是 CM3 内核里面的一个简单的定时器，其时钟由外部时钟源（STCLK）或内核时钟（FCLK）来提供，一般我们在编程的时候选择 FCLK 作为它的运行时钟，FCLK 由 SYSCLK 八分频得到。NVIC 的寄存器是以存储器映射的方式来访问的，所以 **core_cm3.h** 头文件中也包含了寄存器的存储映射和一些宏声明。

system_stm32f10x.c 的性质跟 **core_cm3.c** 是一样的，也是由 ARM 公司提供，遵守 CMSIS 标准。该文件的功能是根据 HSE 或者 HSI 设置系统时钟和总线时钟（AHB、APB1、APB2 总线）。系统时钟可以由 HSI 单独提供，也可以让 HSI 二分频之后经过 PLL（锁相环）提供，也可以由 HSE 经过 PLL 之后获得。具体可参考 STM32 的时钟树：（该图摘自 STM32 参考手册中文版 47 页）。



注意: **system_stm32f10x.c** 文件只是设置了系统时钟和总线时钟, 至于那些外设的时钟是在 **rcc.c** 这个文件中实现的。因为各个 SOC 厂商在 CM3 内核的基础上添加的外设工作的速率是不一样的, 有的是高速外设(时钟经过 APB2 高速总线获得), 有的为低速外设(时钟经过 APB1 低速总线获得), 所以这一功能的实现放在芯片驱动文件夹 **src/rcc.c** 下。这篇文档分析的是 ST(意法半导体)公司的 STM32, 对于其他公司的芯片可能不太一样, 但是, 不论是哪个厂商, 系统时钟都是由 ARM 公司实现, 为的是软件移植的方便。然后, 再从系统时钟里面分频来得到各个外设的时钟。**system_stm32f10x.c** 在实现系统时钟的时候要用到 PLL(锁相环), 这就需要操作寄存器, 寄存器都是以存储器映射的方式来访问的, 所以该文件中包含了 **stm32f10x.h** 这个头文件。

```
24
25  #include "stm32f10x.h"
26
```

stm32f10x.h 这个文件非常重要, 是一个非常底层的文件。以前我在学习其他单片机的时候只是纯粹地操作寄存器, 但不知道寄存器到底是个神马东东。但在这里我们可以学习到寄存器, 也就是内存, 我们访问寄存器也就是在访问内存。它先把内存地址先强制类型转换为指针, 再把该指针实现为一个宏。下面我们通过一个例子来说明这一过程(并不非常准确, 寄存器也是随便取的, 只是为了帮助理解)。

```
#define PORTA *((volatile unsigned long *) (0x4000000))

PORTA = 0xFFFF;           // 往端口A赋值
```

以前我们在写应用程序时, 只是这样写

```
PORTA = 0xFFFF; // 往端口A赋值
```

但为什么这样写就可以操作内存呢, 其实系统为我们提供的头文件已经帮我们做了很多工作了。假如换种方式来操作寄存器, 如下。这就再简单不过了。

```
*((volatile unsigned long *) (0x4000000)) = 0xFFFF;
```

学过 C 语言的朋友都明白：这是将一个十六进制值通过强制类型转换为一个指针，再对这个指针进行解引用操作。

假如我们在操作寄存器的时候都是用这种方法的话会有什么缺点没。当然有：1、地址容易写错 2、我们需要查大量的手册来确定哪个地址对应哪个寄存器 3、看起来还不好看，且容易造成编程的错误，效率低，影响开发进度。所以处理器厂商都会将对内存的操作封装成一个宏，即我们通常说的寄存器，并且把这些实现封装成一个系统文件，包含在相应的开发环境中。这样，我们在开发自己的应用程序的时候只要将这个文件包含进来就可以了。

stm32f10x.h 就是实现这么个功能的文件，并且它把这种功能发挥得更好。下面我们通过分析其源码来看看 STM32 是怎么样来实现其存储器映射的。要是有了 STM32 存储系统的知识的话，对这些源码理解的会更快。关于 STM32 存储器系统大家可以参考《ARM Cortex-M3 权威指南》中文版 宋岩译 第五章：存储器系统，重点看 83 页。要是把这章的内容都理解了，那么 **stm32f10x.h** 这个文件的源码也就理解的差不多了。

下面我们以 RCC 来说明这一功能的实现过程。RCC 是英文 Reset and Clock Control 的简写，是复位和时钟控制，是 STM32 片上的一个外设，这个外设里面包含了很多寄存器。ST 库把这些寄存器都定义在一个结构体里面，寄存器的长度都是无符号整型 32 位的，如下所示：

```
0686  /**
0687   * @brief Reset and Clock Control
0688   */
0689
0690  typedef struct
0691  {
0692      __IO uint32_t CR;
0693      __IO uint32_t CFGR;
0694      __IO uint32_t CIR;
0695      __IO uint32_t APB2RSTR;
0696      __IO uint32_t APB1RSTR;
0697      __IO uint32_t AHBENR;
0698      __IO uint32_t APB2ENR;
0699      __IO uint32_t APB1ENR;
0700      __IO uint32_t BDCR;
0701      __IO uint32_t CSR;
0702  } RCC_TypeDef;
0703
```



```

0105 #define      _I      volatile const      /*!< defines 'read only' permissions */
0106 #define      _O      volatile            /*!< defines 'write only' permissions */
0107 #define      _IO     volatile            /*!< defines 'read / write' permissions */

```

__IO 在 core_cm3.h 这个头文件中定义，uint32_t 在 stdint.h 这个头文件定义。
stm32f10x.h 这个文件中包含了 core_cm3.h 和 stdint.h 这两个头文件。如下所示：

```

0196 #include "core_cm3.h"
0197 #include "system_stm32f10x.h"
0198 #include <stdint.h>

```

从上可知 RCC_TypeDef 这个结构体中声明了很多的变量，这些变量的名字跟 RCC 里面的寄存器的名字是对应的。关于 RCC 寄存器的具体内容可以参考《STM32 参考手册中文》。但是，对变量的操作怎么转化为对寄存器的操作呢？接着往下看……………

STM32 是 32 位的 CPU，其寻址空间可以达到 4GB（2 的 32 次方等于 4GB），即从 0X00000000 到 0XFFFFFFF。这 4GB 的空间被分成了许多功能模块，其中地址 0X40000000 到 0X5FFFFFFF 这 512M 空间用于片上外设，即片上外设的寄存器区。详细内容可参考《ARM Cortex-M3 权威指南》中文版 宋岩译，第 83 页。以下是代码实现 RCC 寄存器是如何映射的。

声明片上外设的所有寄存器的起始地址，当要定义某些寄存器的地址的时候只需在这个地址上加上一定的偏移量即可。

```

0879 #define PERIPH_BASE      ((uint32_t)0x40000000)

```

声明高速寄存器的起始地址，因为外设要高速和低速之分，这从它们的时钟频率上可以体现出来，RCC 就属于高速的外设。

```

0886 #define AHBPERIPH_BASE  (PERIPH_BASE + 0x20000)

```

声明 RCC 寄存器的开始地址，从这个地址开始的一段内存都是 RCC 的寄存器，具体这段内存有多大，由上面的 RCC_TypeDef 这个结构体的长度确定。


```
0943 #define RCC_BASE (AHBPERIPH_BASE + 0x1000)
```

将 RCC 声明为一个 RCC_TypeDef * 型的指针，用于指向 RCC 寄存器的起始地址。这样，我们就可以通过 RCC 这个指针来访问寄存器了。如：RCC->CR = 0xFFFFFFFF;

```
1017 #define RCC ((RCC_TypeDef *) RCC_BASE)
```

现在我们把上面这个例子的代码整理在一起，整体来看下。

```
043 typedef struct
044 {
045     __IO uint32_t CR;
046     __IO uint32_t CFGR;
047     __IO uint32_t CIR;
048     __IO uint32_t APB2RSTR;
049     __IO uint32_t APB1RSTR;
050     __IO uint32_t AHBENR;
051     __IO uint32_t APB2ENR;
052     __IO uint32_t APB1ENR;
053     __IO uint32_t BDCR;
054     __IO uint32_t CSR;
055 } RCC_TypeDef;
056
057 #define PERIPH_BASE ((uint32_t)0x40000000)
058 #define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
059 #define RCC_BASE (AHBPERIPH_BASE + 0x1000)
060 #define RCC ((RCC_TypeDef *) RCC_BASE)
061
062 RCC->CR = 0xFFFFFFFF; // 用户代码
```

STM32 有非常多的寄存器，这里只是以外设 RCC 来作为例子。注意，这些源码在 **stm32f10x.h** 头文件中并不是紧靠在一起的，而是每个功能模块都分开的，如全部寄存器的结构声明放在一起，外设存储器映射地址声明放在一起，外设的指针声明放在一起。我们这里把他们放在一起是为了好理解。具体情况可查看源码。**stm32f10x.h** 头文件还包含了寄存器的位声明，要知道 STM32 是具有非常强大的位处理功能的，具体这里不详述，可参考《ARM Cortex-M3 权威指南》第五章的 5.5 位带操作部分。

上面的 **RCC->CR = 0xFFFFFFFF;** 是我们以前学习单片机时直接操作寄存器的方法，但 ST 官方库为我们想到了一个更绝妙的方法，就是将写到寄存器里面的值都实现为一个宏，宏名即是实现什么功能的英文描述，达到了让人一看宏名就知道往这个寄存器里面写这个宏是实现什么功能，非常的方便。当我们往寄存器里面写一个值时，实际上是操作了寄存器的所有位的值，但更绝的是在这个库里面将实现每个位功能的值也实现为一个宏。这些功能的实现都包含在相应外设的头文件中，如 RCC 则为 `stm32f10x_rcc.h`，ADC 则为 `stm32f10x_adc.h`。有很多网友觉得这样很多，眼花缭乱，看不过来，干脆就不用这些宏，而是自个去查看数据手册，逐位查看寄存器每位的值，再将这些值写到寄存器里面去，就像 **RCC->CR = 0xFFFFFFFF;** 这也是网上流行的直接操作寄存器，而不使用库的方法。但请大家认真想想，是真的没有使用库吗？且这样做还要花费大量的精力，实在是不讨好，聪明的人都应该把好钢用到刀刃上。在每个外设里面的驱动文件里面则实现了该外设的所有功能，这些功能的实现大量运用了这些宏。不过也有很多朋友还是自个重新实现这些功能。但这样做也有缺点：1、加长了产品的开发时间，耗费了大量不必要的精力；2、重新实现这些功能就得自个去查阅数据手册，对数据手册的理解也就不一定能达到百分百的正确，除非你觉得你比开发这些芯片的工程师理解的好。所以，我们还是应该使用官方给的库，要站在巨人的肩膀上开发更好的应用程序。假如还是劈头盖脸的从最基础的部分做起，这在真正的产品开发中是要不得的。聪明的工程师都会善于利用别人搭好的舞台来唱自个的戏。大家不愿意用库，究其原因：一个可能是摆脱不了以前学习单片机的方法；另一个可能是没见过这么大的代码量（注：ST 的库文件还是比较庞大的），因为学单片机的朋友很少是像计算机那样搞纯软件的，动辄就上万行的代码，所以初次见到这么庞大的代码还是有点抗拒的，但要是你学过 LINUX 的话且看过其内核的源码的话，这库简直就是小菜一碟啦。当初我用 LINUX+QT+FFMPEG 在 ARM9 平台上做 MP4 项目的时候，要研究的代码比这个都不知要复杂多少倍。

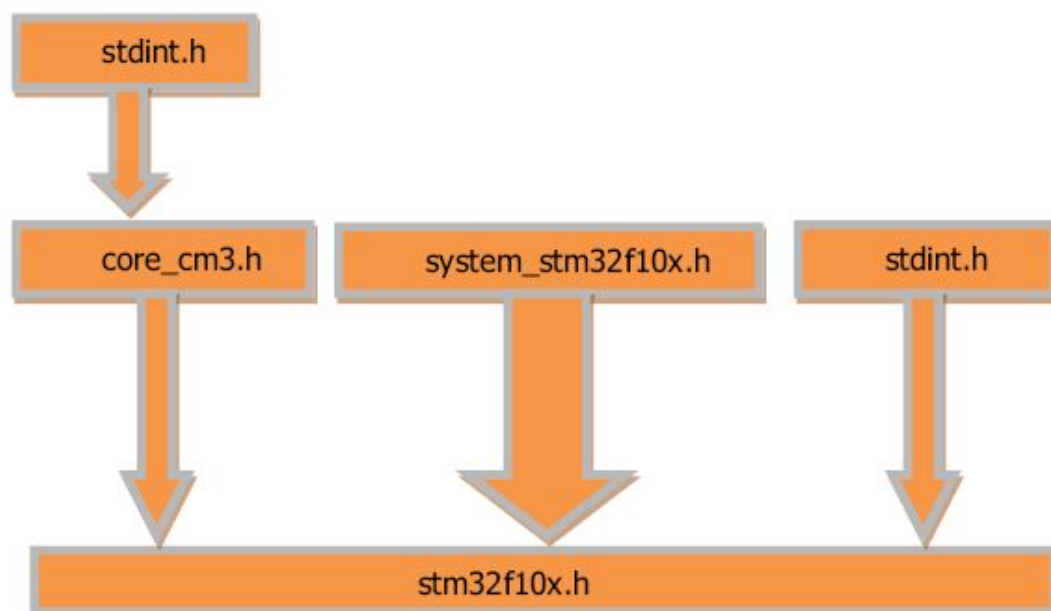
下面我们简单举一个 GPIO 的例子来帮助我们理解上面的这段话。我们知道对 I/O 口的操作分为两种：一种是读数据，一种是写数据。但是 I/O 口数据的操作有总线操作和端口操作，总线操作即是在 16 个端口里面操作数据，而端口操作是在这 16 个端口里面的某一个端口操作数据（注：GPIOA 有 16 个端口，为 0~15）。以下源码来自 `stm32f10x_gpio.h`，只是库里头的 I/O 所有功能函数的声明，它们的具体实现在 `stm32f10x_gpio.c` 这个文件中，这些功能函数的实现都是通过操作寄存器来实现的。当我们要用到时，只需在相应的数据手册下查看下要用到的寄存器即可，没必要重新

去自个实现。我们在应用程序中需要用到 I/O 时只需将 `stm32f10x_gpio.h` 这个头文件包含进来就可以了。

```
292 void GPIO_DeInit(GPIO_TypeDef* GPIOx);
293 void GPIO_AFIODeInit(void);
294 void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
295 void GPIO_StructInit(GPIO_InitTypeDef* GPIO_InitStruct);
296 uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
297 uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
298 uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
299 uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
300 void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
301 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
302 void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction BitVal);
303 void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
304 void GPIO_PinLockConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
305 void GPIO_EventOutputConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
306 void GPIO_EventOutputCmd(FunctionalState NewState);
307 void GPIO_PinRemapConfig(uint32_t GPIO_Remap, FunctionalState NewState);
308 void GPIO_EXTILineConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
```

看到这里，假如你还是坚持要直接操作寄存器且要自个去重新实现功能函数而不使用库的话，那我这个文档没能说服你，我有罪，阿门!!!

最后，我们再来梳理下这个库里面的关键的头文件之间的关系。



stdint.h

是 ANSI C 头文件，位于 RVMDK 这个开发环境的安装目录，是开发环境自带的，其功能是提供一些数据类型的定义。

core_cm3.h、system_stm32f10x.h

是由 ARM 公司为 SOC 厂商提供的基于 Cortex-M3 核的外设接口层，独立于芯片厂商符合 CMSIS 标准。CMSIS 标准是 ARM 公司联合其他 SOC 厂商制定的，所以基于 Cortex-M3 核的 SOC 都必须遵守，这样不管是哪个公司生产的芯片都可以用 ARM 公司提供的这几个文件，方便了软件上的开发。这两个头文件分别对应一个 C 文件，我们在新建工程的时候需要把这两个 C 文件添加进来。还要将启动文件也加进来。

stm32f10x.h

实现了存储器映射和寄存器的声明，是一个非常重要的头文件。这个头文件包含了 core_cm3.h、system_stm32f10x.h、stdint.h 这三个头文件。



综上：在我们的应用程序中只需将 **stm32f10x.h** 这个头文件包含进来即可，这件就可以通过 **stm32f10x_conf.h** 这个头文件方便的选择某些外设的驱动程序。这个配置头文件里面包含了每个外设驱动的头文件，如下图所示：

```
25 /* Includes -----
26 /* Uncomment the line below to enable peripheral
27 /* #include "stm32f10x_adc.h" */
28 /* #include "stm32f10x_bkp.h" */
29 /* #include "stm32f10x_can.h" */
30 /* #include "stm32f10x_crc.h" */
31 /* #include "stm32f10x_dac.h" */
32 /* #include "stm32f10x_dbgmcu.h" */
33 /* #include "stm32f10x_dma.h" */
34 /* #include "stm32f10x_exti.h" */
35 /* #include "stm32f10x_flash.h" */
36 /* #include "stm32f10x_fsmc.h" */
37 /* #include "stm32f10x_gpio.h" */
38 /* #include "stm32f10x_i2c.h" */
39 /* #include "stm32f10x_iwdg.h" */
40 /* #include "stm32f10x_pwr.h" */
41 /* #include "stm32f10x_rcc.h" */
42 /* #include "stm32f10x_rtc.h" */
43 /* #include "stm32f10x_sdio.h" */
44 /* #include "stm32f10x_spi.h" */
45 /* #include "stm32f10x_tim.h" */
46 /* #include "stm32f10x_usart.h" */
47 /* #include "stm32f10x_wwdg.h" */
48 /* #include "misc.h" */ /* High level functions
```

我们需要哪个驱动，只需将它的注释去掉即可。

stm32f10x_conf.h 包含在 **stm32f10x.h** 中，如下图所示：

```
6970 #ifdef USE_STDPERIPH_DRIVER
6971     #include "stm32f10x_conf.h"
6972 #endif
```

注意：宏 **USE_STDPERIPH_DRIVER** 我们在新建工程的时候在  这个工具的  选项卡中声明，如下所示：

```
Define: USE_STDPERIPH_DRIVER, STM32F10X_HD
```

M3 的库就简单介绍到这，至于具体到每个外设的库的应用还需大家多花时功力呀，我这可不能一一详述，但在接下来的模块教程中会详细讲到芯片中每个外设资源的使用。