最简单的 C 程序（STM32 版的 helloworld）

下面开始讨论如何在 STM32 上写一个最简单的程序，会谈到程序执行的细节，原理，如何编程，如何编译及链接我们写的程序，如何通过 OpenOCD 把程序烧写到 STM32 芯片内部的 Flash 上，如何执行等。

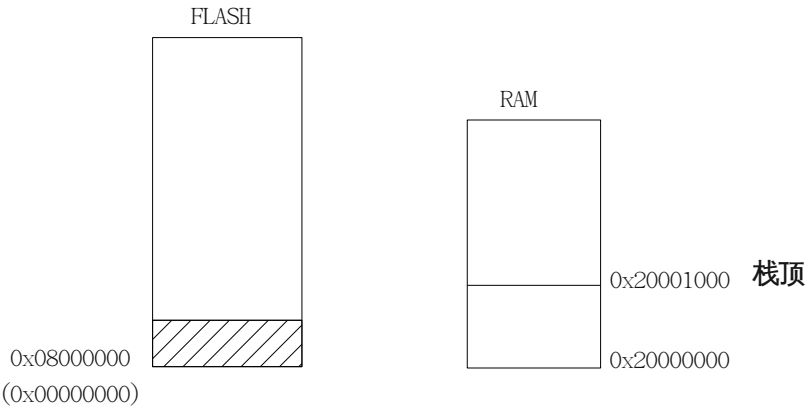
程序的运行方式

开始写之前，先说说最简单的 C 程序是如何运行的。

为了程序足够简单，我们可以让 CPU 直接从 FLASH 上取指令并执行，而且程序中没有全局变量，因此编译出来的目标文件中是数据段长度是 0，这样避免了初始化 RAM 的步骤，因为数据段是可读写的，如果目标文件中有数据段，我们就必须在程序的启动过程中，将数据段复制到 RAM 中，才能确保程序的正常工作。

程序的运行环境

我们来定义一下程序执行时的存储器映射（memory map）



FLASH 的阴影区域表示保存的程序镜像，程序执行过程中的栈当然只能是在 RAM 中，示意图中标出了栈顶指针。

复位后程序的运行流程

处理器复位后，就会开始“取值->执行“的循环了。因此，PC 寄存器在复位后的值就很关键了。

查查书吧，在《The Definitive Guide To ARM Cortex M3》中，我们可以看到：

在离开复位状态后，Cortex M3 做的第一件事就是读取下列两个 32 位整数的值：

- 从地址 0x0000,0000 处取出 MSP 的初始值
- 从地址 0x0000,0004 处取出 PC 的初始值—这个值是复位向量，LSB 必须是 1。然后从这个值所对应的地址处取值。

以上是 ARM 对 CortexM3 核定义的行为，那 ST 作为芯片的制造商，是如何实现的呢？再来查查 ST 的参考手册吧。

从 ST 的 Reference Manual 中，我们可以查到 STM32 系列处理器的引导模式设置和不同的模式下处理器的行为。我们所关心的就是最简单的情况，从内置的 Flash 启动，也就是 BOOT0=0 的情况。

刚才已经提到了，内置的 Flash 起始地址是 0x08000000，这岂不是意味着 Cortex M3 无法从 Flash 中取得复位后需要的 MSP 初始值和 PC 初始值了？STM32 对此的解决方案是地址别名，内置的 Flash 有两套地址空间，除了能从 0x08000000 访问外，从别名（从 0x00000000 开始）也能访问。

基于以上的分析，我们来总结一下我们程序的 image 该存放哪些信息。

- MSP（栈顶指针）初始值
- PC 初始值（LSB 必须为 1）
- 程序的代码段，数据段等

下面的代码就能达到我们的目的

```
__asm__(".word 0x20001000");
__asm__(".word main");
main()
{
```

在以上代码中，我们指定 MSP 为 0x20001000，即栈大小为（0x20001000-0x20000000=0x1000），对我们这么小的程序而言，4k 大小的栈应该绰绰有余了。需要指出的是，GNU 的 tool chain 能自动帮我们处理好 PC 初始值 LSB 必须为 1 的问题，我们只要保证“main”是一个 C 函数就行。其实，这个跟我们写 PC 上的程序还是有点区别的，函数名字其实是可以任意起的，这里起为“main”其实也是为了方便理解起见。毕竟，我们没有利用 tool chain 的启动代码，也就没有必要按 tool chain 的要求来命名主函数了。

接下来，就该分析实现功能的程序怎么干活的了。

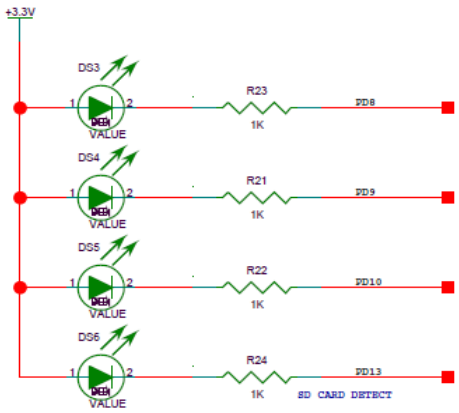
程序都干了些啥

程序再简单，也不能没有输出啊。可是在嵌入式系统中，可没有屏幕输出 hellowor1d，因此我们只能落入俗套，来玩点灯游戏吧。

为了点灯，程序需要操作那些外设呢？操作这些外设需要读写那些寄存器呢？

由于点灯程序“过于”简单，我们只需要操作 STM32 的 GPIO 就可达到目的。对 GPIO 的操作涉及到的几个寄存器我们可以查阅 ST 提供的使用手册得到具体信息，还要结合自己开发板的硬件电路来确定寄存器的值。在本人的开发板上，D 号 GPIO 的 9 脚连着一个 LED 灯，就用它了！

下图即为点灯相关的原理图，程序运行的效果就是 LED(DS4)不停的闪烁。



接下来分析要操作那些寄存器，并确定这些寄存器的值。

1. 使能 D 号 GPIO 端口的时钟
2. 配置 D 号 GPIO 端口的脚 9(PD9)为通用推拉输出模式(General purpose output push-pull)
3. 交替设置 PD9 的值为 0 和 1，控制 LED 灯的亮灭

寄存器的确定就要查 ST 的手册了

- APB2 peripheral clock enable register(RCC_APB2ENR)

地址：0x40021000 + 0x18

复位后的值：0x00000000

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
Res.	USART1EN	Res.	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	Reserved		IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.	AFIO EN
	rw		rw	rw	rw	rw			rw	rw	rw	rw	rw		rw

把(IOPD EN)设为 1 即可使能 GPIOD 的时钟

- Port configuration register high (GPIOx_CRH)(x=D)

地址：0x40011400 + 0x4

复位后的值：0x44444444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

根据我们的需求

- CNF9[1:0] = 00(General purpose output push-pull)
- MODE[1:0] = 01(输出模式，最大速度 10MHz)

- Port bit set/reset register (GPIOx_BSRR) (x=D)

地址：0x40011400 + 0x10

复位后的值：0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

亮灯：GPIOD9 输出低电平，BR9 = 1

灭灯：GPIOD9 输出高电平，BS9 = 1

至此，程序的实现细节都已经清除了，那就开始编程吧，happy programming!

程序代码分析

我们写的第一个程序如此简单，干脆就把它贴在正文里了吧。

blink.c

```
#define GPIOD_CRH   (*((volatile unsigned long*)(0x40011400 + 0x4)))
#define GPIOD_BSRR  (*((volatile unsigned long*)(0x40011400 + 0x10)))
#define RCC_APB2ENR (*((volatile unsigned long*)(0x40021000 + 0x18)))

__asm__(".word 0x20001000");
__asm__(".word main");
main()
{
  unsigned int c = 0;

  RCC_APB2ENR = (1 << 5);  //IOPDEN = 1
  GPIOD_CRH = 0x44444414;

  while(1) {
    GPIOD_BSRR = (1 << 25);  // ON
    for(c = 0; c < 100000; c++);
    GPIOD_BSRR = (1 << 9);   // OFF
    for(c = 0; c < 100000; c++);
  }

}
```

simple.ld

```
SECTIONS {
        . = 0x0;
        .text : {
                *(.text)
        }
}
```

Makefile

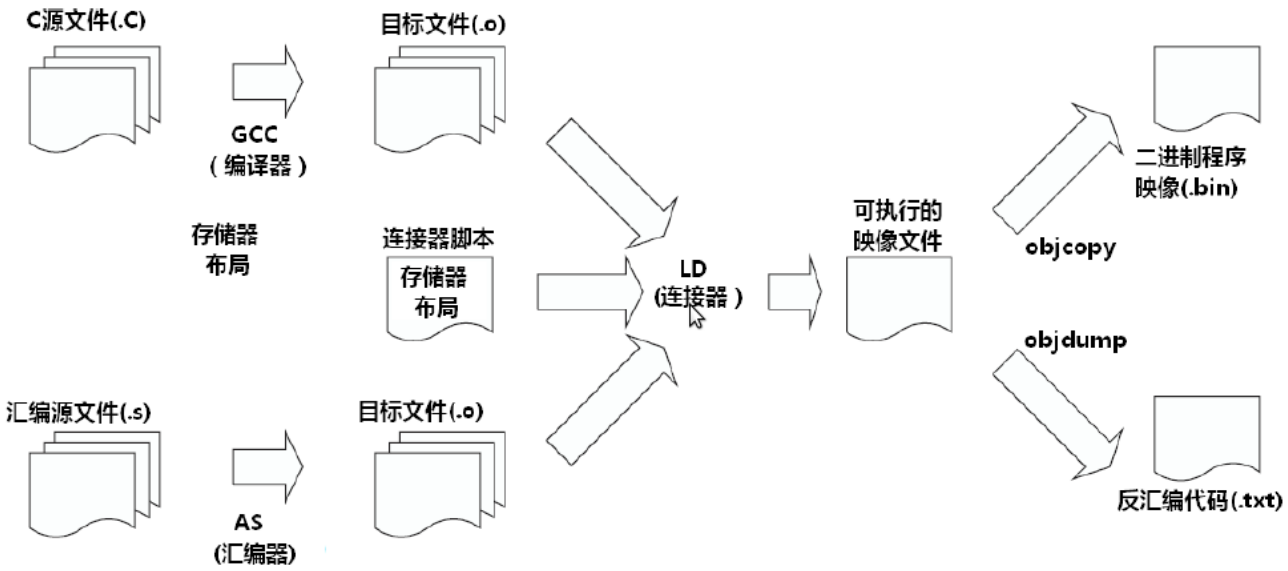
```
PREFIX := arm-none-eabi-
.PHONY: all clean

all: blink.bin
blink.o: blink.c
        $(PREFIX)gcc -mcpu=cortex-m3 -mthumb -nostartfiles -c blink.c -o blink.o
blink.out: blink.o simple.ld
        $(PREFIX)ld -T simple.ld -o blink.out blink.o
blink.bin: blink.out
        $(PREFIX)objcopy -j .text -O binary blink.out blink.bin
clean:
        rm -f *.o *.out *.bin
```

下面来分析一下这三个文件吧。

blink.c 中的实现我们在前面已经分析过了，需要注意的是程序中有两个 for 循环是用来延时的，STM32 执行的速度还是很快的，如果不加延时的话，LED 灯闪烁的频率会非常高，感兴趣的读者可以自己计算一下(时钟频率在以后的章节会有详细的介绍)。闪烁频率非常高的后果就是人眼觉察不到闪烁，而会觉得灯一直是亮着的。

simple.ld 是我见过的最简单的链接脚本了，“. = 0x0”指示程序链接的逻辑起始地址是 0x0，即程序中的所有符号做重定位时参考的起始地址是 0x0（重定位及相关知识可以参考《Linker and Loader》）。按照我们前面的分析，STM32 对内置 Flash 的访问有两套地址，因此我们这里把起始地址换成 0x08000000 也是可以的，感兴趣的读者可以自己试一试。

Makefile 是一个非常简单的 GNU make 脚本，《Managing Projects with GNU Make》这本书对 Makefile 的写法有非常详细和深入的探讨，读者有兴趣可以读一读，肯定有收获。这里需要注意的是编译命令中的选项“-mcpu=cortex-m3 -mthumb -nostartfiles”。Makefile 中描述的编译，链接以及二进制文件生成的步骤如下图（来源于《The Definitive Guide To The ARM Cortex M3》中文版）所示：



有了这三个文件，我们只需在终端敲“make”命令，make 程序就会帮忙把程序的镜像“blink.bin”做出来了。既然程序这么简单，编译出来的结果也不会太复杂，来一起分析一下吧。

先来反汇编一把

```
#arm-none-eabi-objdump -D blink.out
```

下面是结果的片段：

```
blink.out:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main-0x8>:
   0:   20001000        andcs   r1, r0, r0
   4:   00000009        andeq   r0, r0, r9

00000008 <main>:
   8:   b480            push    {r7}
   a:   b083            sub     sp, #12
   c:   af00            add     r7, sp, #0
   e:   f04f 0300       mov.w   r3, #0
  12:   607b            str     r3, [r7, #4]
  14:   f241 0318       movw    r3, #4120       ; 0x1018
  18:   f2c4 0302       movt    r3, #16386      ; 0x4002
.......
```

看到了吧，逻辑起始地址是 0x00000000，栈指针 MSP 是 0x20001000，PC 初始值是 0x00000009，GNU tool chain 自动帮忙处理好了 LSB 必须为 1 的问题。

细心的读者应该会去看看编译出来的目标文件中都有那些段了。在我的电脑上测试的结果中，只有”.text”，”.comment”，”.ARM.attributes”三个段，其中跟程序的执行相关的只有".text”段。在更复杂的程序中，还会有“.rodata”，”.data”和”.bss”段。另外用户也可以自己定义段。对这些不同段的处理，以后会有更详细的描述。

再来看看要烧到 Flash 中的二进制文件吧。

```
#od -t x1 blink.bin
0000000 00 10 00 20 09 00 00 00 80 b4 83 b0 00 af 4f f0
0000020 00 03 7b 60 41 f2 18 03 c4 f2 02 03 4f f0 20 02
0000040 1a 60 41 f2 04 43 c4 f2 01 03 44 f2 14 42 c4 f2
0000060 44 42 1a 60 41 f2 10 43 c4 f2 01 03 4f f0 00 72
0000100 1a 60 4f f0 00 03 7b 60 03 e0 7b 68 03 f1 01 03
0000120 7b 60 7a 68 48 f2 9f 63 c0 f2 01 03 9a 42 f4 d9
0000140 41 f2 10 43 c4 f2 01 03 4f f4 00 72 1a 60 4f f0
0000160 00 03 7b 60 03 e0 7b 68 03 f1 01 03 7b 60 7a 68
```

```
0000200 48 f2 9f 63 c0 f2 01 03 9a 42 f4 d9 d2 e7 00 bf
0000220
```

仅供好奇心超强的读者参考，注意字节序是 little-endian 哦。

程序的烧写和运行

程序写好了，也编译完成了，接下来就该把程序二进制文件烧写到芯片内置的 Flash 中并运行测试一下了。

烧写程序有很多种选择，本文使用的是 OpenOCD+OpenJTAG 的组合。OpenOCD 在 linux 下的使用请读者自己 google 吧。

具体烧写步骤如下：

1. 启动 OpenOCD

```
#openocd -f openocd.cfg -f stm32.cfg
```

2. 在 shell 中执行以下命令，完成 Flash 的擦除和烧写

```
#telnet localhost 4444
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> halt
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x81000000 pc: 0x0800002a msp: 0x200000d8
> poll
background polling: on
TAP: stm32.cpu (enabled)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x81000000 pc: 0x0800002a msp: 0x200000d8
> flash protect_check 0
device id = 0x10016414
flash size = 512kbytes
successfully checked protect state
> stm32x mass_erase 0
stm32x mass erase complete
> flash write_bank 0 /tmp/blink.bin 0
not enough working area available(requested 16384, free 16336)
wrote 144 bytes from file /tmp/blink.bin to flash bank 0 at offset 0x00000000 in 0.175022s (0.803 kb/s)
> reset
JTAG tap: stm32.cpu tap/device found: 0x3ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x3)
JTAG tap: stm32.bs tap/device found: 0x06414041 (mfg: 0x020, part: 0x6414, ver: 0x0)
```

3. 开始测试一把

```
>reset
```

怎么样？灯亮了么？如果没亮的话，请仔细检查一下步骤是不是正确，寄存器的值是否正确。

稍微复杂点的 C 程序（加强版的 **helloworld v1**）

看完简单版的 helloworld 后，来看个稍微复杂点的吧。这个程序的功能其实和上一个程序一样简单，也是使 LED 闪烁，它的复杂性体现在对代码不同段的管理上。

由于大伙已经基本熟悉 STM32 程序的编写和运行了，本节的解释就不会再那么详细了。

本程序与上一个程序的不同之处在于，本程序的目标文件中有数据段。前面提到过，数据段必须要复制到RAM中，而这个复制的步骤必须要靠自己写程序来完成。在电脑上写程序，程序员不必自己完成这个步骤，而会由操作系统提供的装载器（loader)完成。

由于有存储器间数据的搬移以及PC在不同存储器上的跳转，而且我们编译链接生成的仍然是一个二进制文件，带来的结果必然是不同存储器上的数据要拼接成一个二进制文件。这意味着链接器脚本(*.ld)不会再像上一节那么简单了。

先来分析一下编译器生成的目标文件中都有些什么段，链接器对各个段(section)如何放置，以及如何对里面的符号进行重定位。

每个目标文件中都有一系列的段，每个段都有一个名字和尺寸信息，大多数段还包含有数据。如果一个段包含的数据在程序运行时必须载入到存储器中，则该段是可载入的（loadable)；如果一个段没有包含数据，但是必须在存储器中预留一块区间，则该段是可分配的（allocatable)。那些既不可载入，又不可装载的，一般都是跟代码调试有关的信息。

arm交叉编译器生成的目标文件中，一般会含有.text .data .rodata .bss .comment .ARM.attributes等不同的段。.comment和.ARM.attributes我们不用关心，它们都是提供一些信息便于我们使用调试器的。当然，我们也可以自定义一些段，根据我们的需要做特殊处理。

.text：代码段

此段中的数据是指令序列，一般是只读的。

.rodata：只读数据段

此段中的数据是只读的变量。如C语言中用const关键字定义的全局变量即是此类。

.data：有初始值的数据段

在C语言中定义一个全局变量并赋初值，则该变量会存在此段中。

.bss：无初始值的数据段

未赋初值的全局变量存放在此段。

上面提到了三种数据段，值得注意的是，只有全局变量才会出现在段中，局部变量为栈变量，随时生成，随时销毁，并没有对应的段存在。

从以上对各段属性的说明中，我们不难得出链接器对各个段的处理方法。需要保存在二进制目标文件中的只有".text",".rodata"和".data"三个段，对".bss"段的处理，我们只需要记住该段的大小，在程序启动过程中对其进行清零就可以了。按链接器手册中的定义，代码段，只读数据段，有初始值的数据段为可装载段(loadable section)，".bss"那样无初始值的数据段叫可分配段(allocatable section)。

链接器的输入是编译器生成的若干个目标文件(*.o)，输出是一个大的映像文件(*.out)。链接器将输入的目标文件中各个段进行拼接，对其中的一些符号进行重定位，最终得到一个大的映像文件。可见，链接器必须从链接脚本中得到两方面的信息

1. 如何拼接

   在我们的例子中，我们需要拼接各目标文件的.text, .rodata, .data, .bss，用户自定义段，最终生成的目标文件中含有.text, .data, .bss和自定义段。因为.text和.rodata属性一致，可把它们合在一个段内。最终生成的各个段首位相连，我们需要注意的是，有些段需要从4字节对齐的地址开始，因此各个段之间可能会存在填充字节。

   简而言之，在这个步骤中，我们对输入段（input sections，即输入目标文件中的段）进行处理，拼接成期望的输出段（output sections，即目标映像文件中的段）。“input section”和“output section”就是连接器手册中使用的术语。

   跟拼接有关的信息如下：

   - 需要保留哪些输入段和输入段的属性

- 输出段的属性

2. 如何重定位

   把原始目标文件拼接成一个大目标文件并不是链接器的全部，程序编译过程中有些符号的地址并不能确定，比如某些外部变量，编译过程中无法确定它的地址，链接器还要负责解析这些变量的地址，因为这些地址只有在链接阶段才能最终确定。由于变量在输入段中的偏移是已知的，因此要确定重定位后的地址，链接器只需知道输出段运行时的起始地址即可。

在以上的分析中，可以看出链接脚本要处理的核心问题就两个："可执行映像文件中各段的布局"和"运行时各个段在存储器中的地址"。对本程序而言，链接脚本必须要能告诉程序代码：

- 数据段在可执行映像文件中的偏移是多少
- 数据段有多大
- 程序应该把数据段复制到内存的什么地址

可以看出，上面反复出现"偏移"、"地址"，一个是指在存储器上的保存位置，一个是指程序运行时的地址（即程序运行时的虚拟地址）。为此，有必要先统一一下术语。在链接脚本的帮助手册中，定义了两种地址：

1. LMA(Load Memory Address): 表示段被保存在存储器上的地址
2. VMA(Virtual Memory Address): 表示代码运行时的地址

大多数情况下（不需要段搬移的情况），这两类地址是相同的。比如目标文件由操作系统装载器载入的情况，或者像上面那个例子，没有操作系统载入器，但没有数据段需要搬移，指令直接从 FLASH 上读取并执行。但在本例中，由于数据段需要从 FLASH 中搬移到 SRAM 中，所以数据段的 LMA 和 VMA 并不一样。代码段存储在 FLASH 上的地址为 LMA，搬移到 SRAM 中的地址为 VMA。在后面的描述中，经常会出现这两类地址。链接器脚本的一个重要作用就是管理各个段的 LMA 和 VMA，并在必要的情况下，把有关信息传给程序代码使用。下面会看到实例：

```
//stm32f103vet6.ld

MEMORY
{
  FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 512k
  SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

/* Section Definitions */
SECTIONS
{
  .text :
  {
    KEEP(*(.isr_vector))
    *(.text)
    *(.rodata)
    . = ALIGN(4);
    _etext = .;
  } > FLASH

  .data : AT (_etext)
  {
    _data = .;
    *(.data)
    . = ALIGN(4);
    _edata = . ;
  } > SRAM

  /* .bss section which is used for uninitialized data */
  .bss (NOLOAD) :
  {
    _bss = . ;
    *(.bss)
    . = ALIGN(4);
    _ebss = . ;
  } > SRAM

  _end = . ;
}
```

链接脚本的精确语法定义可以参考手册，这里只是对该脚本的功能做具体分析。

首先，链接脚本定义了两个存储器区间（MEMORY)，下面输出段定义中会包含对它们的引用，表明某一输出段位于某一存储器区间。其实我们不用存储器区间定义这个机制也可实现同样的功能，但是使用它会使描述简单而清晰。

脚本的核心部分是 SECTIONS 部分，从手册中，我们可以查到 SECTIONS 命令的标准格式规范：

```
SECTIONS
{
    section-command
    section-command
    ...
}
```

sections-command 是如下四个之一

- ENTRY 命令
- 符号赋值
- 输出段定义
- 重叠定义

其中，"ENTY 命令“和”符号赋值“可以出现在”输出段定义“内部。
上面的脚本中用到了“符号赋值”和“输出段定义”两种元素，“重叠定义”一般较少使用，“ENTRY 命令”以后有机会再具体分析。

先来分析简单的--”符号赋值“。前面提到过，链接器脚本必须能够提供一些地址相关的信息给程序代码，其实就是通过这个机制来实现的。程序代码中引用一些外部符号，在链接器脚本中给这些外部符号赋值。在链接阶段，链接器负责处理这些外部符号的对应关系。”符号赋值“提供了一种链接器给程序代码提供信息的途径。上面我们看到一个特殊的符号：".”，这个特殊符号叫“位置计数符”，出现在不同的位置会有不同的含义。位置计数符表示的是在包含它的元素的地址空间内到当前偏移的地址，如果出现在 SECTIONS 中，它表示相对于把各段拼接后的地址(LMA)；如果在“输出段定义”内部，则表示在该段内部的地址（VMA)。

来看核心部分吧—输出段定义的精确格式如下：

```
section [address] [(type)] :
  [AT(lma)] [ALIGN(section_align)]  [SUBALIGN(subsection_align)]
{
    output-section-command
    output-section-command
    …
} [>region] [AT>lma_region] [:phdr :phdr …] [=fillexp]
```

大家可以看到好多可选元素，一般情况下输出段定义不会用到这么多。需要注意的一个细节是，所有 LMA 都是出现在 AT 关键字之后的。

Output-section-command 可以是以下四个中的一个：

- 符号赋值
- 输入段描述
- 数值（用于在输出段中插入一些数据）
- 特殊关键字

最常用的当然就是“符号赋值”和“输入段描述”了。

输入段描述

“输入段描述”的作用就是告诉链接器如何把输入文件映射到内存布局中。一个输入段描述包含一个文件名，和一系列括号包着的输入段名，文件名和输入段名都可以包含通配符：

file-name [(sections_name1 section_name2 ...)]

最常用的输入段描述是把所有文件中的某个段全部输出到一个输出段中，例如，把所有文件中的.text 段输出到输出文件中对应的命令如下

*.(.text)

这种描述应付我们例子中的需求已经够用了。使用统配符的副作用是各个文件中的各个段最终在输出文件中的排序就由链接器来决定了，这在多数情况下是无所谓的。如果需要显示指定某个文件的某个段出现在最靠前的位置，就必须显示指定文件名了，如下所示：

data.o(.text)

刚提到了输入段在输出文件中的排序问题，事实上，在使用通配符的情况下，最终排序的结果是由链接器处理的顺序决定的，如果想显示指定顺序，除了显示指定文件名之外，可以指定链接器处理的顺序，如按字母顺序。

总结一下，从逻辑的角度上来看，上例中链接器脚本的总体结构如下所示

```
存储区间定义
 {
     区间 1
     区间 2
     ...
 }
输出段定义
 {
     段 1
     {
          输入段描述 1
          输入段描述 2
          ...
     } > 区间 1
     符号定义
     ...
     段 2
     {
          ...
     } > 区间 n
     ...
}
```

基于以上的背景知识，先以.text 段为例分析一下对输入段的处理。

.text 段会依次包含所有输入文件中的.isr_vector 段，.text 段，.rodata 段，在.isr_vector 上加 KEEP 是为了防止链接器的垃圾收集功能启用时忽略.isr_vector 段，实际上，isr_vector 段的内容相当关键。先分析一下吧：

从”The Definitive Guide to The ARMCortexM3"上，我们可以查到：”在 0 地址处提供 MSP 的初始值，然后紧跟着就是向量表（向量表在以后还可以被移至其它位置），向量表中的数值是 32 位的地址，而不是跳转指令。向量表的第一个条目指向复位后应执行的第一条指令“。前面提到过，STM32 中的 FLASH 可以从 0x00000000，或 0x08000000 起始的地址访问。isr_vector 中的内容就对应 MSP 初始值，还有向量表的内容。我们这里的程序这么简单，就没必要实现向量表移动了。

现在，我们先不去管 CortexM3 为实现中断和异常机制而引入的一系列机制，先来看看中断向量表的定义。

编号为 1-15 的对应系统异常，大于 15 的则都是外部异常，具体的异常定义可以参考 CortexM3 的手册。实现中断控制其实还是很复杂的，但对简单的程序来说，中断优先级可以先不用考虑，我们只需要搞清楚存储器上需要保存哪些值就足够了。来看看上电后的向量表吧：

地址	异常编号	值（32 位整数）
0x0000,0000	-	MSP 的初始值
0x0000,0004	1	复位向量（PC 初始值）
0x0000,0008	2	NMI 服务例程的入口地址
0x0000,000c	3	Hard Fault 服务例程的入口地址
…	…	其它异常服务例程的入口地址

这个表中只列出了向量表的前三项，正好对应我们的程序代码。

```
__attribute__ ((section(".isr_vector")))
pfnISR VectorTable[] =
{
  (pfnISR)(0x20010000),  // The initial stack pointer is the top of SRAM
  ResetISR,              // The reset handler
  NMIException,
  HardFaultException
};
```

可以看出，硬件上电后，PC 会自动跳转到 ResetISR()函数。

链接器脚本对其它段的定义与对.text 段的定义类似，就不做详细分析了。

符号赋值

脚本中有 5 个关键的符号定义：_etext, _data, _edata, _bss, _ebss，这几个符号本身没有意义，而且也不占用输出段的空间，它们用做标志数据，真正起作用的是它们的地址，能用来完成向程序代码提供链接器有关信息的功能。注意这些符号表示的地址都是输入段内相应偏移在运行时的地址。

- _etext

  该变量存在输出文件中.text 段的运行地址结尾，由于.text 段的链接地址和运行地址是一致的，而且.data 段的起始正是.text 段的结尾，因此它的地址能代表数据段在输出目标文件上的链接起始地址。

- _data, _edata

  这两个变量分别保存在.data 段的开头和末尾。它们地址的差值能提供段的大小信息，正是我们所需要的。

- _bss, _ebss

  这两个变量保存在.bss 段的开头和结尾，程序根据这两个变量的地址来定位需要清零的区域。

刚才提到了 ResetISR()函数是系统上电后执行的第一段代码，因此完成数据段内容的搬移非它莫属了。

```
//*****************************************************************************
//
// The following are constructs created by the linker, indicating where the
// the "data" and "bss" segments reside in memory.  The initializers for the
// for the "data" segment resides immediately following the "text" segment.
//
//*****************************************************************************
extern unsigned long _etext;
extern unsigned long _data;
extern unsigned long _edata;
extern unsigned long _bss;
extern unsigned long _ebss;

void ResetISR(void)
{
  unsigned long *src, *dst;

  //
  // Copy the data segment initializers from flash to SRAM.
  //
  src = &_etext;
  dst = &_data;
  while (dst < &_edata) {
    *dst++ = *src++;
```

```
  }

  //
  // Zero fill the bss segment.
  //
  for(dst = &_bss; dst < &_ebss; dst++) {
    *dst = 0;
  }

  //
  // Call the application's entry point.
  //
  main();
}
```

代码说明了一切。ResetISR（）函数完成了数据搬移及.bss 段的清零之后，就调用 main()函数，开始干正事了。

好了，干正事的代码大伙应该很容易看懂了吧，这里就不再赘述了。把程序编译出来，烧到开发板上，让三个灯闪起来吧。

再复杂点的 C 程序（加强版的 helloworld v2）

在这个程序中，我们会先执行 FLASH 上的一小段程序，把指令以及一些数据都从 FLASH 上拷贝到 RAM 中，然后再跳转到 RAM 上执行。对于这个拷贝的步骤，有人就要问了，为啥要把指令读入 RAM 之后再执行呢？在 RAM 上执行有啥特别的好处吗？

就这个问题，基于我个人的理解，有两个方面的原因。

1. 在 RAM 上取指然后执行，速度会比较快。
2. 指令全放在 RAM 中，程序的设计会更灵活，限制更少。STM32 是哈佛结构，也就是数据总线和指令总线分离，从硬件角度上来讲，把指令全放在 FLASH 中，数据全放在 RAM 中没有任何问题。但这样做还是会给软件设计带来一些不便。程序员必须负责合理安排不同的存储区域，而且在程序设计阶段就必须对不同的存储区域有合理的规划。

大家如果开发过 ARM9 上的 bootloader 的话，应该都有过类似的经验。这类 bootloader 的实现中，开始运行的第一步就是把代码段，数据段全部移到 RAM 中，尤其是对有些从 NAND FLASH 启动的开发板，这一步是无法避免的。

这个例子比前一个例子只是多了一个代码段的拷贝，看起来并不算复杂，关键是对中断向量表及中断服务例程的特殊处理，读者可以做为练习自己实践一下。给出的参考例程中，中断向量表和中断服务例程还是运行在 FLASH 上的，感兴趣的话可以把除了 ResetISR（）的中断服务例程也放在 SRAM 中。自己试试吧！

总结：通过三个点灯程序，大家基本能够熟悉链接器脚本的使用，中断向量表的组织，真正干活的程序如何访问寄存器。接下来就可来点复杂的了。

串口程序

写了三个点灯程序了，换个输出方式吧。拿到一个开发板后，通常要做的第一件事就是把串口调通。毕竟和 LED 灯比起来，串口能提供的信息要丰富的多了。

STM32 提供了三个 USART(Universal synchronous asynchronous receiver transmitter)端口 USART1, USART2, USART3，以及两个 UART(Universal asynchronous receiver and transmitter)端口 UART4, UART5。该端口的功能还是很丰富的。除了能用于异步通信之外，还可提供诸如同步通信，智能卡接口，红外信号接收等。在通信的实现上，可以支持中断接受，DMA 数据传送等高级功能。在这里，我们没必要使用这些高级功能，最简单的异步发送和接受就够了。USART1 能达到的最高速率是 4.5Mbits/s，其他端口的最大速度是 2.25Mbit/s。

硬件准备

实现双向异步通信最少需要两个管脚，接收脚(Receive Data In, RX)和接收脚(Transmit Data Out, TX)。

- RX：接收脚是串行数据的输入口，数据恢复过程中，为了区分有效数据和噪声，采用了过采样技术。
- TX：当传送器没有使能时，该输出口可以作为通用输入输出端口(GPIO)使用。当传送器使能但没有数据需要传送时，该端口为高电平。

我们先来指定一下我们期望实现的端口参数吧

- 波特率为 9600bps

- 数据位 8，停止位 1，无硬件握手，无流量控制

在开发板上，可以查到 USART1 是引出的，就用它了。

软件出马

来看看实现串口数据发送需要干些啥吧。

1. 设置 USART_CR1 的 UE 位为 1 来使能 USART
2. 设置 USART_CR1 的 M 位来定义数据位长度
3. 设置 USART_CR2 中的相关位来定义停止位长度
4. 设置 USART_CR3 中的相关位来设置使能 DMA 模式
5. 设置 USART_BRR 来选择期望的波特率
6. 设置 USART_CR1 中的 TE 位来发送一个 idle 帧作为发送的开始
7. 写数据到 USART_DR 寄存器，STM32 会把数据发送出去。
8. 重复步骤 7，直到数据全部发送完毕。把最后一个数据发送之后，等待 TC 位为 1。TC 位为 1 显示最后一帧的发送已结束，这个步骤是为了保证数据完整的被发送。

可以看出，上面的步骤中，都是操作寄存器，寄存器的值怎么确定当然是要查手册了。其中最复杂的是波特率寄存器值的计算。

查阅手册，可以发送器和接收器的波特率是相同的，波特率的计算公式如下：

$$Tx/Rx\ baud = \frac{fck}{16 * USARTDIV}$$

fck 是外设的输入时钟（USART2，3，4，5 是 PCLK1，USART1 为 PCLK2，这就是为什么 USART1 的最高速度跟其它端口不一样的原因）。USARTDIV 是一个无符号的定点小数，下面就来看看这个定点小数是怎么由 USART_BRR 寄存器来定义的：

USART_BRR 寄存器中定义了 DIV_Mantissa 和 DIV_Fraction，分别用来定义 USARTDIV 的整数部分和小数部分

$$USARTDIV = DIVMantissa + DIVFraction/16$$

在实际的计算中，难免会遇到取整，这个时候就必须要注意误差不能太大，在 ST 的手册中有对不同波特率和时钟频率下误差的分析。

下面来计算一下对我们现在的需求，USART_BRR 寄存器的值应该是多少吧。值得注意的是，STM32 重启后默认的 fck 是 8MHz，在这里先不管时钟设置的细节，因为时钟的设置还是相对比较麻烦的，下一个例子会专门予以讨论。

在开发板上，fck=8MHz, 我们期望的 baud = 9600, 则 USARTDIV=52.083，进而 DIV_Mantissa=0d52=0x34，DIV_Fraction=0.083*16=0x1，则 USART_BRR=0x341

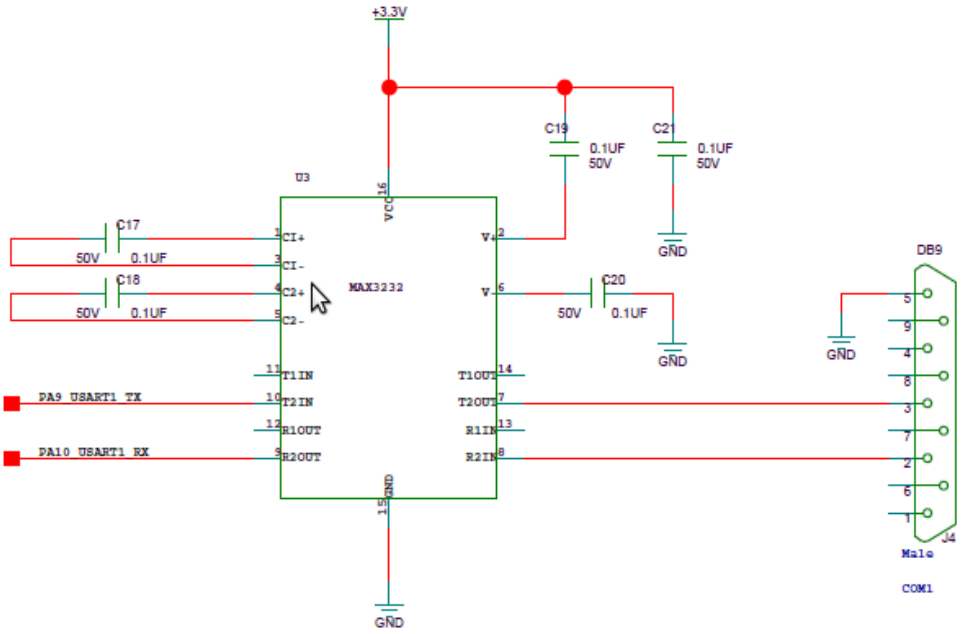
好了，寄存器的值确定下来没有问题了，查一下寄存器的地址，写代码实现吧，这次再来一个版本的 helloworld：在串口上输出一行 helloworld。

USART1 寄存器起始地址是 0x40013800，USART_SR 的偏移是 0x00，USART_DR 的是 0x04，USART_BRR 的是 0x08，USART_CR1 的是 0x0C，USART_CR2 的是 0x10，USART_CR3 的是 0x14，往里面填值吧。

另外还需要注意的是，要像前面的例子设 IO 口输入输出模式那样，设置串口线对应管脚的工作模式。如下面的原理图所示，Tx 脚复用 GPIOA 的 9 脚，Rx 脚复用 GPIOA 的 10 脚。我们必须像设置 GPIO 口输入输出模式那样，设置 Tx 和 Rx 脚的工作模式。需要注意的是，Rx 脚为输入模式，与 GPIO 口设置的可选模式相同，而对 Tx 脚这样的输出管脚，需要设置专门的工作模式(Alternate function output Push-pull 或 Alternate function output Open-drain)，而不能设置为 General purpose output 模式。对串口输出而言，需要选择 Alternate function output Push-pull，据此可确定 GPIOA_CRH 寄存器的值。

最后，千万不要忘了使能 USART1，GPIOA 的时钟，就像前面几个例子，需要使能 GPIO 端口的时钟那样。

在这里还跟读者分享一个调试串口的经验，如果遇到串口无输出的情况，可以把 PC 端的串口软件的波特率设一个比较低的值，如果是因为波特率的原因导致无输出的话，PC 端较低的波特率就会输出乱码，这样就可以确定问题的根源，为进一步分析提供线索了。



RS232

STM32 总线介绍

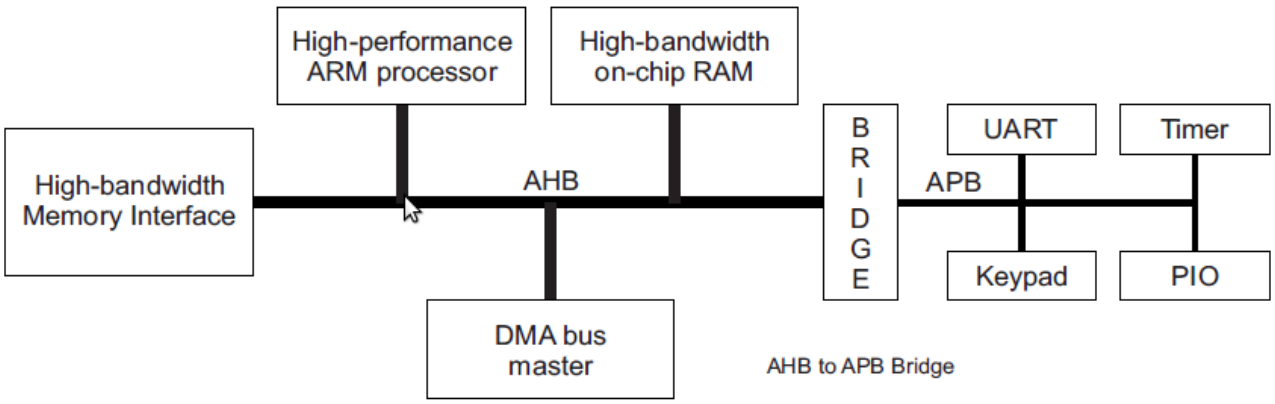
AMBA 总线规范

AMBA(Advanced Microcontroller Bus Architecture) 是 ARM 公司为高性能微控制器定义的片上通信标准，在标准中，定义了三种总线：

1. Advanced High-performance Bus (AHB)
2. Advanced System Bus (ASB)
3. Advanced Peripheral Bus (APB)

从名字就可看出各个总线的特性，其中 ASB 与 AHB 的作用类似，都可用于连接系统中的高速模块，但 AHB 比 ASB 更适合高性能的场合，因此，很多 SoC 中只使用了 AHB。而 APB 则被用于低功耗的外设模块。

符合 AMBA 规范的微控制器一般包含一个高性能的系统骨干总线，用于处理器，片上存储器和其它 DMA 设备的互联，而且还连接一个桥接器，将 APB 上个低速外部设备连接到 AHB 上。下图是一个典型的符合 AMBA 规范的系统：



APB 本身主要用于低带宽周边外设之间的连接，AHB-to-APB 桥是 APB 上唯一的主模块，也是 AHB 上的从模

块，其主要功能是锁存来自 AHB 的地址、数据和控制信号，并提供二级译码以产生 APB 外围设备的选择信号，从而实现 AHB 协议到 APB 协议的转换。

对学习 STM32 来说，我们没有必要深究 AMBA 的规范细节，只需要知道 STM32 中那些模块连接在 AHB 上，哪些模块连接在 APB 上，如何使能它们的时钟就行了。对好奇的读者，可以参考 ARM 公司的 AMBA 规范。

STM32 中的总线

STM32 中的 AHB 与其它 ARM 核的 SoC 比起来没啥特别之处，我们需要特别关心的是 APB 总线上都连接了那些设备。

APB 在 STM32 中分为两种：APB1 和 APB2，分别用来连接低速外设和高速外设。

- 连接在 APB1 上的设备有：

CAN, USB, I2C1, I2C2, UART2, UART3, UART4, UART5, SPI2, SPI3, Watchdog, Timer2, Timer3, Timer4, Timer5

- 连接在 APB2 上的设备有：

GPIO_A-E, USART1, ADC1, ADC2, ADC3, Timer1, Timer8

这些信息来源于 STM32 的 Datesheet。搞清楚各个模块连接那个总线，对实现精确的时钟控制非常重要。前面例子中已经提到过，要操作一个模块，务必先使能该模块连接总线的时钟。

时钟例程

STM32 中的时钟

其实，时钟模块是应该最先仔细学习的内容，它是很多模块工作的基础，但前面的例子一直避免接触它们，主要是因为其复杂性和不直观性。经过前面几个例子的实验，现在可以通过设计一个时钟相关的程序来熟悉它。

本例的目的是使 STM32 全速（72MHz）工作，并通过串口输出消息，同时控制 3 个 LED 交替亮灭。注意本程序将会使用和以前程序一样的延时函数，这样就可以很直观的通过比较灯两灭的速度来比较 CPU 运行的速度了。

前面串口的例子中已经提到过，系统重启后默认的时钟是 8MHz，实际上，STM32 的系统时钟可以由以下三个时钟源驱动：

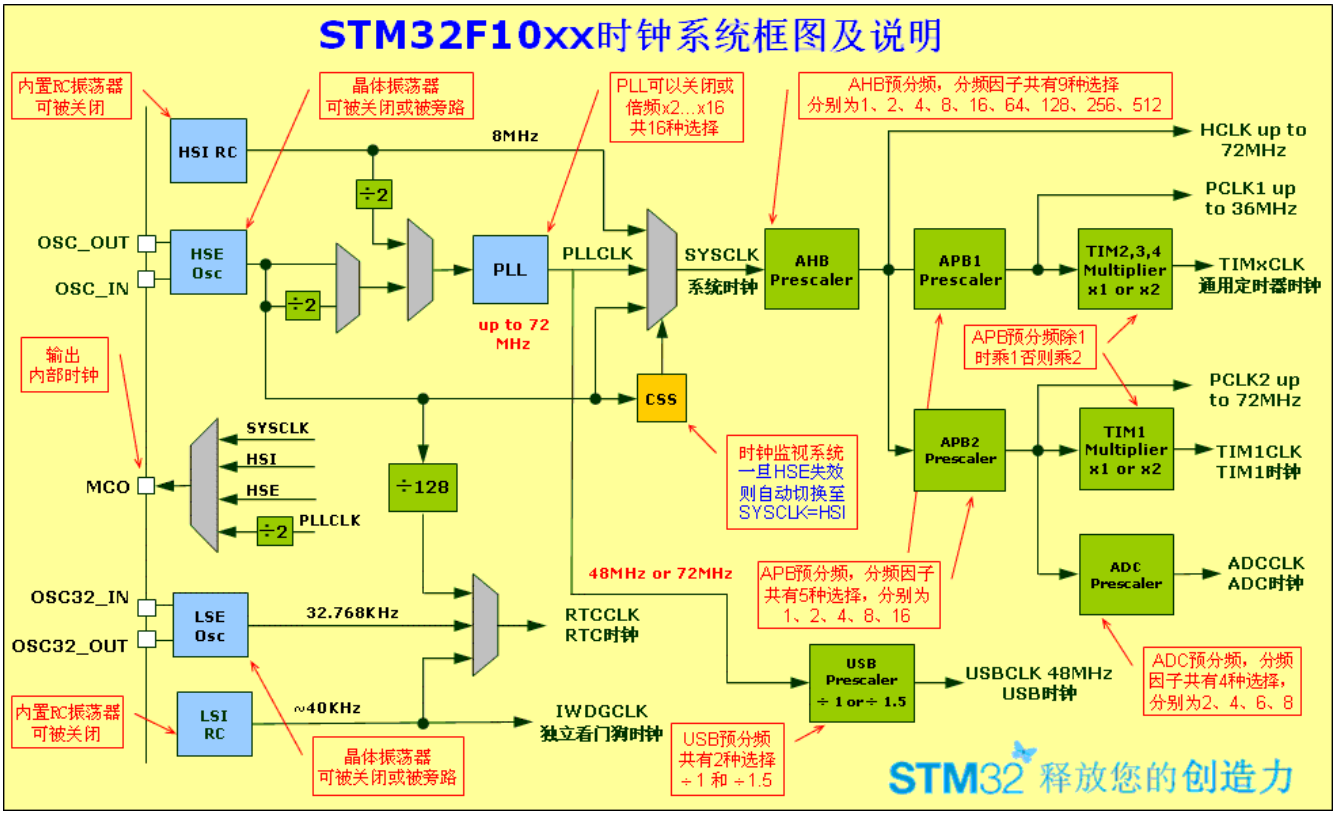
1. 高速内部 RC 振荡器(HSI oscillator clock)，频率为 8MHz
2. 高速外部振荡器(HSE oscillator clock)
3. 锁相环时钟(PLL clock)

除此之外，还有两个二级时钟源

1. 40kHz 的内部 RC 振荡器(LSI)，用来驱动看门狗。而且，该时钟也可用做实时时钟，实时时钟用来将处理器从停止/休眠状态唤醒。
2. 32.768kHz 的低速外部晶振（LSE)，也可以用来驱动实时时钟。

各个时钟都可以独立的打开或关闭，进而降低系统的功耗。系统重启后，默认的系统时钟就是高速内部 RC 振荡器。

下图来自 ST 的培训资料，这张图的信息非常丰富，各个时钟的频率限制，配置都有很清晰的说明。



下面先对时钟模块的输出，即生成的各个时钟的用途做简要分析。

- SYSCLK
  系统时钟，为 PLL 的输出，是 HCLK，PCLK1，PCLK2，TIMxCLK，TIM1CLK，ADCCLK 的基础
- HCLK
  AHB Clock，AHB 总线的时钟
- PCLK1
  APB1 low speed clock，外设总线的低速时钟
- PCLK2
  APB2 high speed clock，外设总线的高速时钟
- TIMxCLK
  Timer2, Timer3, Timer4 的时钟
- TIM1CLK
  Timer1 的时钟
- ADCCLK
  ADC 模块的时钟
- USBCLK
  USB 时钟，必须是 48MHz
- RTCCLK
  RTC 时钟
- IWDGCLK
  独立看门狗时钟
- MCO
  输出时钟

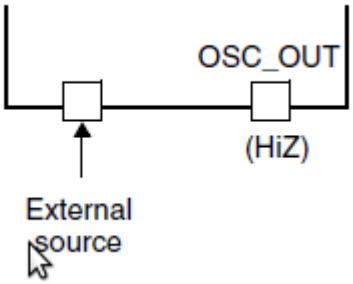
在进行时钟配置的时候，务必注意各个时钟的频率限制，图中已经注明了一些，还需要注意的是在使用 HSI 作为 PLL 的输入时，SYSCLK 最大值为 36MHz。

时钟源

接下来再看看时钟源有关的管脚。前面提到的三个系统时钟驱动信号，只有 HSE 有对应的外部管脚。而两个二级时钟源中，32.768KHz 的外部晶振需要两个管脚。

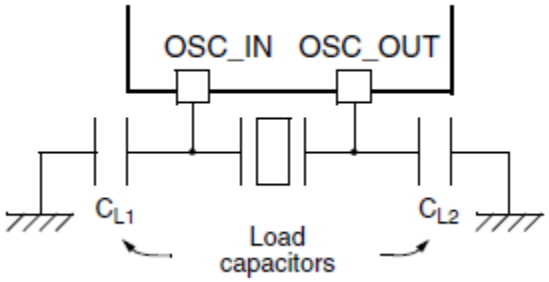
HSE 信号有两种生成方式

- 直接输入时钟信号



在这种模式下，OSC_OUT 管脚必须是高阻态。

- 外接晶振/陶瓷谐振器生成时钟信号



这种模式是最常见的用法，在进行 HSE 相关的时钟配置之前，一定要保证晶振起振并进入稳定状态。

LSE 的管脚解法与 HSE 的类似，这里不再赘述。

时钟的稳定性

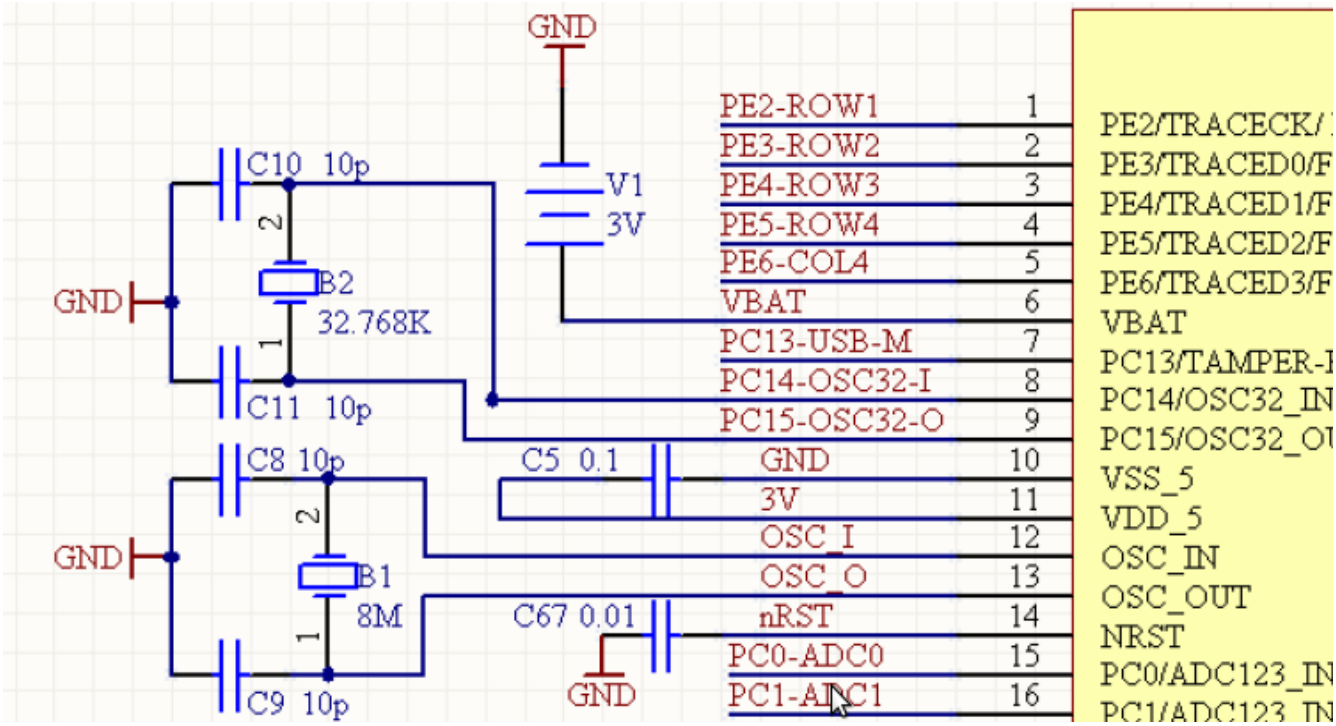
STM32 内部的时钟源（HSI, LSI)虽然使用方便，但精度没有外部时钟高，在实际使用中，如果使用到了内部时钟源，需要进行校准。这方面的详细信息可以参考 ST 的手册。

STM32 时钟的配置步骤

搞清楚了 STM32 中的各个时钟后，接下来了解一下如何配置这些个时钟的频率以及如何使能它们。在前面的例子中，我们只是简单的提到了要使能某某时钟，在这一节会更精确的阐述各个时钟的配置和使能。

先来看看开发板上时钟相关的电路。

OSC_IN，OSC_OUT 外接了晶振/陶瓷谐振器，晶振的频率为 8MHz，OSC32_IN 和 OSC32_OUT 同样外接了晶振/陶瓷谐振器，晶振频率为 32.768kHz。可以看出，为了得到 72MHz 的系统时钟，PLL 的倍频系数必须设为 9。



来看时钟初始化的详细步骤吧：

1. 打开外部高速时钟晶振 HSE
2. 等待外部高速时钟晶振稳定
3. 配置时钟相关的参数
   - PLL 的输入时钟源设为 HSE
   - PLL 的倍频系数
   - PCLK1，PCLK2 相对于系统时钟的降频系数，即 APB1 和 APB2 的频率
   - HCLK 相对系统时钟的降频系数，即 AHB 的频率
4. 使能 PLL
5. 等待 PLL 稳定
6. 选择 PLL 作为系统时钟
7. 等待硬件确认系统时钟已经是 PLL 时钟

通过以上步骤，我们就可以成功的把默认 8MHz 的系统时钟换成 72MHz，由 PLL 倍频生成的时钟了。接下来配置串口输出信息，配置 GPIO 点亮 LED 的步骤跟前面的例子就没啥区别了。唯一需要注意的是 PCLK2 和 PCLK1 变了，即串口的时钟源发生了变化，跟波特率相关的寄存器值需要重新计算。而且，系统时钟是以前的 8 倍，延时函数的执行时间更短，LED 亮灭的频率更高了。

马上写代码试试看吧。

使用 ST 的固件库编程

通过前面几个例子的实践，可以发现直接操作寄存器来编程还是相当枯燥的，而且很容易出错。如果能把对寄存器操作达到一些目的的代码封装成比较通用的函数，无疑会大大方便我们的编程工作。谁能提供最可靠的封装函数呢？当然是 ST 了。实际上，ST 确实提供这样的函数，官方的名字叫通用外设库（Standard Peripheral Libarary)。很多人觉得 ST 提供的外设库效率不够高，其实大可不必这么担心。使用 ST 提供的库，可以让我们把更多的精力集中在需要我们创新的地方，而不是去枯燥的调寄存器。这些机械、无聊的事情就交给 ST 来完成吧。

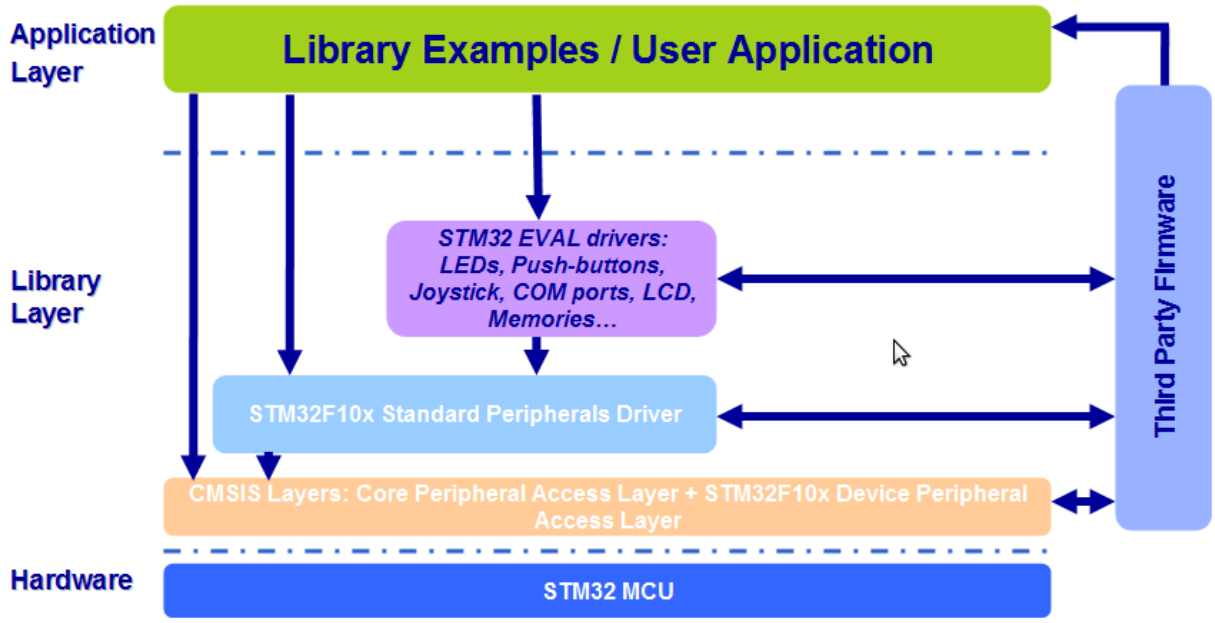
标准外设库简介

通用外设库（STM32F10x Standard Peripherals Libray)的详细介绍可以参考 ST 提供的文档，这里只是简要介绍一下对理解本例子很重要的信息。

首先要强调的一点是，ST 提供的标准外设库是完全符合 CMSIS（Cotex Microcontroller Software Interface Standard）的。CMSIS 是 ARM 公司针对 CortexM 系列处理器定义的一个与芯片制造商无关的，独立的硬件抽象

层，基于符合 CMSIS 的固件库开发程序，代码复用的好处是显而易见的。只要其它芯片供应商的 CortexM 处理器也提供符合 CMSIS 的固件库，我们基于 ST 的固件库开发的程序就很容易移植过去。关于 CMSIS 的详细介绍可以参考 ARM 提供的稳定行。

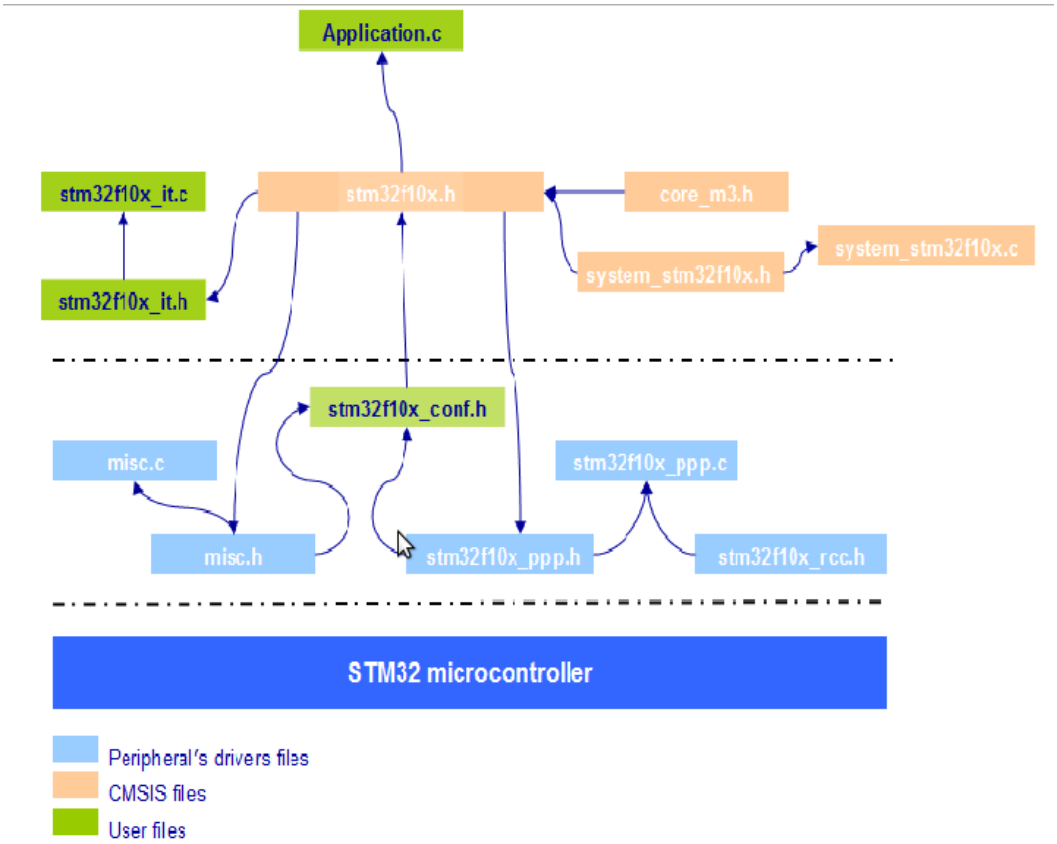
来看看 ST 外设标准库的架构吧，下图摘自 ST 的官方文档



要掌握 ST 标准固件库的使用，关键是掌握“STM32F10x Standard Peripherals Driver”和"CMSIS Layers”。其它部分就轮到我们来发挥自己的创意了。

标准外设库的文件依赖关系

下图同样摘自 ST 的官方文档，它详细的画出了使用 ST 固件库编程时各种文件的依赖关系，我们在这一节实现的例子就遵循了这个依赖关系。



从上图我们可以看出，每个外设都对应有相应的驱动文件，不好分类的就都归为 misc 类了。理解了大致框架之后，大家可以花点时间读一下 ST 关于标准固件库的文档，磨刀不误砍柴功嘛。

外设库函数的使用

下面总结一下基于外设库怎样完成对一个外设的初始化，配置以及操作。假设外设名是 PPP

1. 定义一个 PPP_InitTypeDef 结构体类型的变量，如：
   ```
   PPP_InitTypeDef PPP_InitStructure;
   ```
   该结构体一般定义在数据段，能被用来初始化一个或多个 PPP 实例。
2. 给该机构体的各个成员赋值，有两种方法来完成这个工作
   - 给每个成员单独赋值，如
     ```
     PPP_InitStructure.member1 = val1;
     PPP_InitStructure.member2 = val2;
     PPP_InitStructure.member3 = val3;
     …
     ```
   - 调用函数给各个成员赋默认值，然后修改某些成员的值，如
     ```
     PPP_StructInit(&PPP_InitStructure);
     PPP_InitStructure.member1 = val1;
     ```
3. 调用 PPP_Init()函数初始化 PPP 外设
   ```
   PPP_Init(PPP, &PPP_InitStructure);
   ```
4. 使能 PPP 外设
   ```
   PPP_Cmd(PPP, ENABLE);
   ```
5. 调用其他接口完成需要的工作
   ```
   PPP_xxx(PPP, …);
   ```

在前面的例子中，我们曾经提到过，对任何一个外设操作之前，我们必须先使能该外设对应的时钟，使用标准外设库同样需要记住这一点，如下几个外设库函数可以用来使能各个外设的时钟

- `RCC_AHBPeriphClockCmd(RCC_AHBPeriph_PPPx, ENABLE);`
- `RCC_APB2PeriphClockCmd(RCC_APB2Periph_PPPx, ENABLE);`
- `RCC_APB1PeriphClockCmd(RCC_APB1Periph_PPPx, ENABLE);`

在使用某个外设的过程中，有时我们会需要将一个外设的所有寄存器全部复位到初始状态，函数 PPP_DeInit()可以帮我们完成这个工作。而且在完成设备的初始化之后，如果我们需要修改某个设置，可以修改 PPP_InitStructure 的成员后再次调用 PPP_Init()函数。

外设库编程

接下来就到了介绍如何使用 ST 标准外设库来编程的时候了。程序的基本要素其实跟我们前面不用标准外设库一样，只不过不用我们自己去写代码完成数据段搬移，寄存器操作了。ST 外设标准库中提供了参考的引导程序，只需要稍加修改就可以用 GNU 的 toolchain 来编译链接。ST 外设标准库默认支持的 toolchain 有如下 5 种

- RealView Microcontroller Development Kit **MDK-ARM**
- Embedded Workbench for ARM **EWARM**
- Raisonance Integrated Development Environment **RIDE7**
- Hitex Development Tools **HiTOP**
- Atolic **TrueSTUDIO**

引导程序我们可以直接使用 **RIDE7** 的版本，因为我们的芯片属于 high density 系列，所以引导程序代码为 startup_stm32f10x_hd.s，这个代码跟我们前面的例子中的代码原理基本类似，也是完成数据段的拷贝，bss 段的清零等工作。但这个代码比前面例子中的引导程序要完善多了，在代码中定义了完整的异常矢量表，而且还定义了一个默认异常处理函数。在完成必须的初始化操作之后，启动代码会调用 SystemInit（）函数去实现用户程序定义的初始化任务，然后调用 main（）函数。

链接脚本就需要自己来写了，其实道理跟前面的例子也是一样的，要注意的是对异常矢量的处理，在这里不再赘述，请直接阅读 stm32f103vet6.ld。

来看看程序是如何组织的吧。前面的文件依赖关系图上我们可以看出 stm32f10x.h 是应用程序要包含的头文件，而 stm32f10x_conf.h 包含一些可配置的选项，我们必须确保宏 USE_STDPERIPH_DRIVER 是定义了的，这样才能在 stm32f10x.h 中包含 stm32f10x_conf.h，而 stm32f10x_conf.h 又会包含各个外设驱动的头文件。最后的结果就是应用程序只需要包含 stm32f10x.h 一个文件，就可以使用所有外设驱动定义的函数。如果不定义宏 USE_STDPERIPH_DRIVER，应用程序就只能通过访问寄存器的方法来操作外设了。除此之外，还要记得修改 stm32f10x.h 文件以正确地设置芯片的型号。

下面是例子程序的目录组织结构

```
.
|-- Makefile
|-- cmsis
|   |-- core_cm3.c
|   `-- system_stm32f10x.c
|-- driver
|   |-- misc.c
|   |-- stm32f10x_gpio.c
|   |-- stm32f10x_rcc.c
|   `-- stm32f10x_usart.c
|-- include
|   |-- core_cm3.h
|   |-- misc.h
|   |-- stm32f10x.h
|   |-- stm32f10x_conf.h
|   |-- stm32f10x_gpio.h
|   |-- stm32f10x_rcc.h
|   |-- stm32f10x_usart.h
|   `-- system_stm32f10x.h
|-- startup
|   `-- startup_stm32f10x_hd.s
|-- stm32f103vet6.ld
`-- usr
    |-- main.c
    |-- stm32f10x_it.c
    `-- stm32f10x_it.h
```

读者需要自己花时间去熟悉各个文件中的内容。这里仅介绍两个重要的函数，SystemInit()和 main()

SystemInit()定义在 cmsis/system_stm32f10x.c 中，是 ST 官方提供的实现，里面最复杂的部分就是时钟设置，仔细看看，是不是跟我们前面 clock 例子基本一样？最终系统时钟也设置成 PLL 输出，72MHz。

main()函数是应用程序的核心，ST 提供了很多例子做参考，但 ST 的例子中假设我们使用的是 ST 提供的参考开发板，往往跟我们自己的开发板配置并不一致，因此我们必须自己来实现这个函数。我们当然可以参考 ST 的"STM32 EVAL Driver”，自己封装一些类似的接口，对自己的开发板上的外设进行访问，如 LED，串口，按键

等。在本例中就这样做了，LEDInit(), LEDOn(), LEDOff() 就是这样的三个函数。仔细看看外设库函数的使用，是不是跟前面介绍的一致？

通过使用外设库函数，避免了让我们自己陷入无趣的寄存器设置工作，可以把更多精力放在能发挥创造力的领域--应用设计，何乐而不为呢？如果你仔细的调试过前面的例子程序，就会理解调试寄存器设置是多么烦人的工作，现在，让我们正式逃离吧。

本节的例子程序的功能就是闪烁三个 LED 灯，由于使用了外设库，main()函数的实现清晰了很多，更重要的是，由于避免了寄存器操作，并且使用了符合 CMSIS 规范的外设库，程序还具有了可移植性。

感兴趣的读者可以仔细参考例子，实现其他功能的程序。

插入点题外话，本例子的 Makefile 虽小，但五脏俱全，包含了依赖生成，代码、目标文件分离两大重要功能，是一个很好的参考模板，在以后还会经常用写类似的 Makefile。Makefile 的写法有很多不错的参考资料，比如《Managing Projects with GNU Make》，请感兴趣的读者自己深入学习。

<代续...>