嵌入式系統建構：開發運作於STM32的韌體程式

台灣成功大學資訊工程系嵌入式系統開放教材: http://wiki.csie.ncku.edu.tw/embedded/schedule
改編自: http://www.stmcu.org/module/forum/thread-599882-1-1.html
編譯者: 黃敬群 <jserv.tw@gmail.com>

本文探討如何開發一個得以獨立運作於STM32硬體的韌體程式，會談到程式執行的原理、硬體運作機制，以及GNU Toolchain如何編譯和連結我們撰寫的程式，過程中也探討如何使用OpenOCD把程式燒錄寫到STM32晶片內部的Flash並執行，對於沒有實體裝置的開發者，本文也介紹QEMU系統模擬。

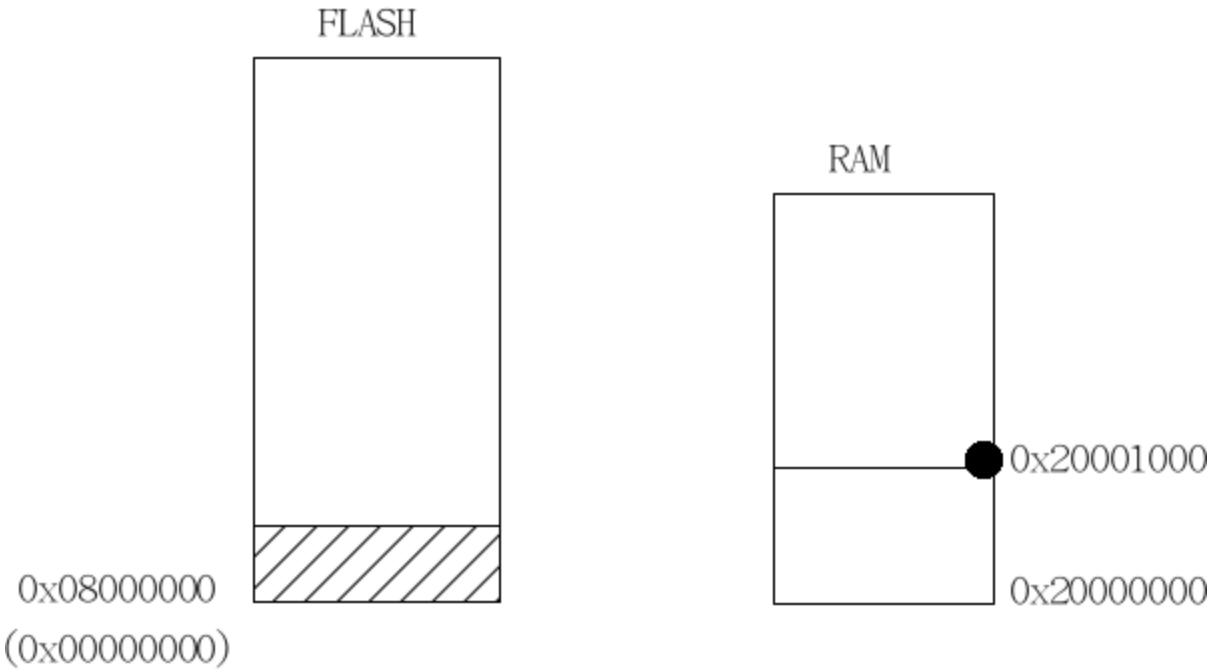
程式的運行方式

在開始進行程式開發前，我們先來探討最簡單的C程式如何運作。

為了使程式足夠簡單，我們可讓CPU直接從Flash上取得指令(fetch instruction)並執行，而且程式中沒用到全域變數，因此編譯出來的目的檔(object file)中是data section長度是0，如此一來，避免了初始化RAM的步驟，因為data section是可讀寫的，如果目的檔中有data section，我們就必須在程式的啟動過程中，將data section複製到RAM中，方可確保程式得以正常工作。

程式的執行環境

我們來定義一下程式執行時的記憶體映射 (memory map)：



Flash的陰影區域表示保存的程式映像(program image)，程式執行過程中的堆疊(stack)當然只能存於RAM 中，示意圖右方黑點標註堆疊頂端指標。

Reset後程式的運行流程

一旦處理器Reset後，就會進行「取得指令—解碼—執行」的循環，也就是3-stage pipeline。因此，PC (program counter; 以下PC均指此暫存器，而非個人電腦)暫存器在Reset後的值就顯得關鍵。在《The Definitive Guide To ARM Cortex M3》中，我們知道，在離開Reset狀態後，ARM Cortex M3所做的第一件事，就是讀取下列兩個 32位元整數的值：

- 從位址0x00000000處取出MSP的初始值

- 從位址0x00000004處取出PC的初始值，這在中斷向量的內含值，其LSB 必須為1 (稍後的實驗會說明這點)。CPU隨即自這個值所對應的位址處取值

以上是ARM對Cortex M3核心定義的行為，那STMicroelectronics作為晶片的製造商，是如何實作的呢？從STMicroelectronics的Reference Manual 得知，STM32系列處理器的引導模式(BOOT)設定，以及不同的模式下處理器的行為。我們在意最簡單的情況：系統從內建的Flash啟動，也就是BOOT0=0的情況。

稍早提過，內建的 Flash起始位址為0x08000000，這是不是說Cortex M3無法自Flash中取得系統Reset後需要的MSP初始值和PC初始值嗎？STM32為此提出的解決方案就是位址別名(alias)，也就是說，內建的Flash有兩套定址空間，除了能從0x08000000存取外，自位址別名(即0x00000000開始)，亦可存取。

基於以上的分析，我們來總結一下我們程式的 image 該存放哪些資訊。

- MSP(堆疊頂端指標)初始值
- PC 初始值(LSB必須為1)
- 程式的text section、data section等

下面的程式碼即可達到我們的期望：

```
asm (".word 0x20001000");
asm (".word main");
main() { … }
```

上述程式碼中，我們指定MSP為0x20001000，即堆疊大小為0x1000 (計算方式: 0x20001000 - 0x20000000 = 0x1000)，對於Hello World等級的程式來說，4K大小的堆疊應該綽綽有餘。需要留意的是，GNU Toolchain能自動幫我們處理好PC初始值LSB必須為1的議題，我們只要確保`main`這個符號是個C函式就行。其實，相較於在GNU/Linux或Microsoft Windows環境中開發程式，有一定程度的區隔，main函式不一定是C程式的第一個被執行的地方，甚至可任意命名，這裡以`main`來命名，其實也是為了方便理解起見。畢竟，我們沒有利用GNU Toolchain提供的啟動程式碼(startup)，也就沒有必要依據toolchain的要求，來命名進入點函式。一般來說，我們稱這樣不需要理會既定作業系統和toolchain規範的執行環境為bare-metal。

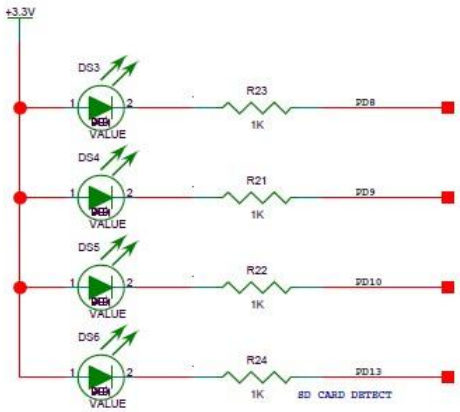
程式都在做什麼？

程式就算再簡單，也不可沒有輸出，不然我們如何檢驗功能呢？在嵌入式系統中，往往缺乏清楚有效的螢幕，讓Hello World程式得以輸出字串，因此，我們只能落入俗套，改玩點燈遊戲。

為了點燈，程式需要操作那些週邊硬體呢？操作這些週邊硬體時，需要讀寫那些暫存器呢？

由於點燈程式相對單純，我們只需要操作STM32的GPIO (General Purpose I/O)，即可達到目的。對於GPIO的操作涉及到的幾個暫存器，可查閱ST提供的參考手冊，實務還得對照開發板的硬體電路，才能確定暫存器值的設定。在本人的開發板上，D號GPIO的9腳連著一個LED 燈，好，就用它了。

下圖即為點燈相關的原理圖，程式運行的效果就讓LED(DS4)不停地閃爍。



接下來，我們要逐一分析相關的暫存器設定，並確定這些暫存器的值。

1. 啟用(enable) D號GPIO port的時鐘
2. 配置D號GPIO port的腳9 (PD9)為通用推拉輸出模式(output push-pull)
3. 間歇地設定PD9的值為0和1，也就是控制LED燈的亮滅

暫存器的設定自然得查詢STMicroelectronics的參考手冊，以下幫讀者列出相關的描述：

- APB2 peripheral clock enable register (**`RCC_APB2ENR`**)
  位址: `0x40021000 + 0x18`
  Reset後的值: `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`**)設為 1，即可啟用GPIOD的時鐘

- Port configuration register high (**`GPIOx_CRH`**) (x=D)
  位址: `0x40011400 + 0x4`
  Reset後的值: `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

依據我們的需求：

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

- Port bit set/reset register (**`GPIOx_BSRR`**) (x=D)
  位址: `0x40011400 + 0x10`
  Reset後的值: `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

亮燈：GPIOD9輸出低電位，BR9 = 1
滅燈：GPIOD9輸出高電位，BS9 = 1

至此，程式的實作細節都已經清楚，我們終於可以開始撰寫程式了。

程式碼分析

第一個版本的程式相當簡單，請見以下：

[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");
int 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]

```
CROSS_COMPILE ?= arm-none-eabi-
.PHONY: all
all: blink.bin
blink.o: blink.c
```

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

下面來分析前述3個檔案。

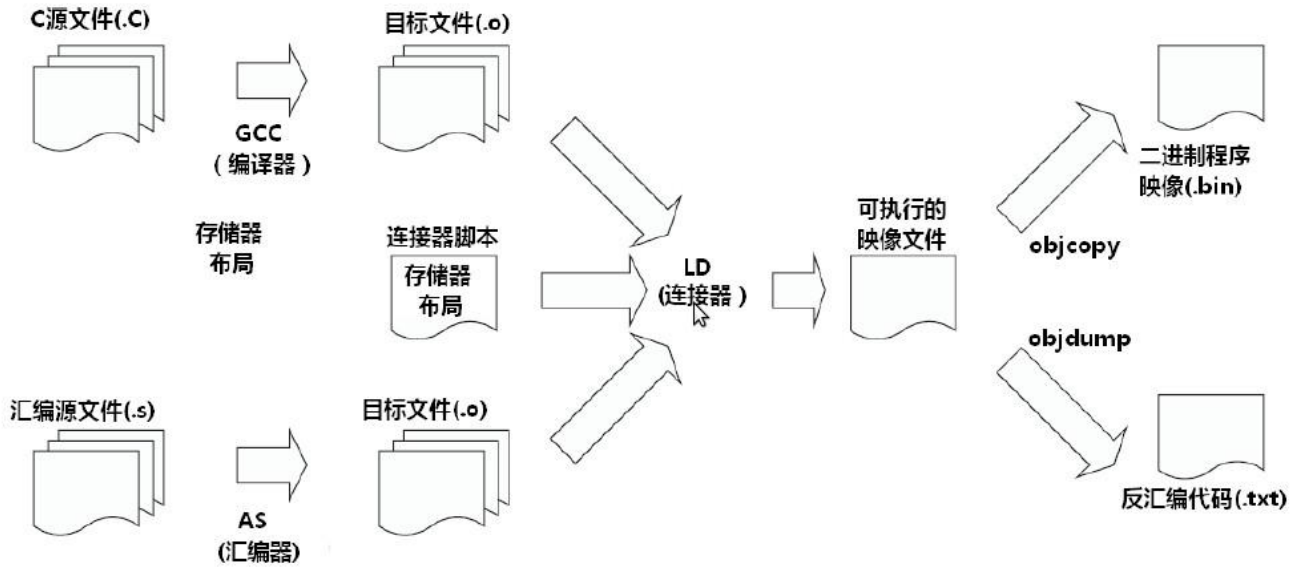
blink.c需要注意的是，程式中有兩個for迴圈用來延時，STM32執行的速度相當快，若不加延時的話，LED燈閃爍的頻率會非常高，讀者不妨自行計算(時鐘頻率在稍後介紹)。閃爍頻率非常高的後果，就是人眼覺察不到閃爍，從而讓人誤認LED燈一直亮著。

simple.ld是個極為簡化的連結腳本(linker script)，其中"`. = 0x0`"指定程式連結的邏輯起始位址位於`0x0`，而程式中的所有符號進行重定位(relocate)時，參考的起始位址為0x0，這部分的背景知識可參考《Linker and Loader》。按照之前的分析，STM32對內建Flash的存取可透過兩種不同的位址，因此我們這裡把起始位址換成0x08000000也行。

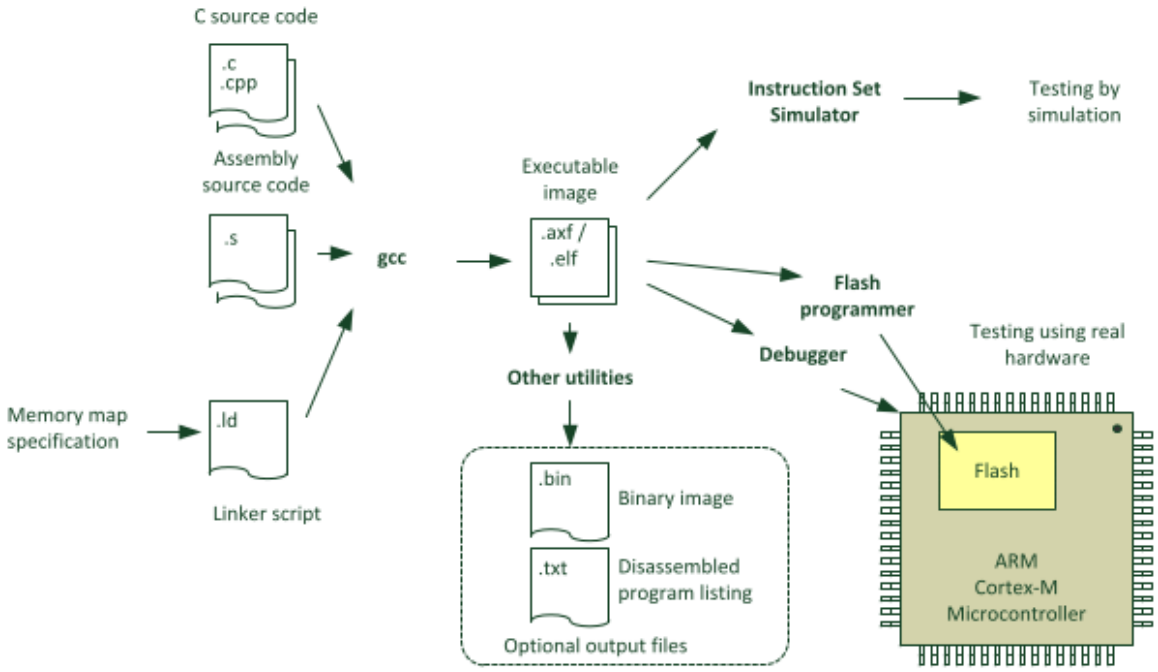
Makefile是非常簡單的GNU make檔案，《Managing Projects with GNU Make》這本書對Makefile的寫法有詳細和深入的探討。我們可察覺到Makefile中指定編譯器的若干參數"`-mcpu=cortex-m3 -mthumb -nostartfiles`"，其中：

- `-mcpu`：要求`gcc`針對`ARM Cortex-M3`產生對應的指令
- `-mthumb`：指定產生`Thumb`指令，而非`ARM`指令，請留意，`ARM Cortex-M3/M4`只支援`Thumb2`指令
- `-nostartfile`：要求連結階段不要使用標準系統起始檔案`(starup file)`，這在沒有作業系統支援的環境是必要的，因為我們自己處理`C`語言程式`main()`函式之前的種種準備動作

Makefile 中描述的編譯流程，可參見下方《The Definitive Guide To ARM Cortex M3》中文版的圖例，清楚得知自原始程式碼、編譯器(gcc)、組合語言程式碼、組譯器(as)、目的檔、連結器腳本、連結器(ld)、以及藉由objcopy工具來產生二進位檔案的種種步驟：



進一進考慮到模擬器和除錯器(debugger)的操作流程，則是以下示意圖：



有了這三個檔案，我們只需在終端機執行make命令，make程式依據指定的編譯和連結規則，自動產生出程式的映像檔blink.bin。既然程式不複雜，相信編譯出來的結果也不該太難，我們來分析吧。

先進行反組譯，使用objdump工具：

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

下面是結果的片段：

```
CROSS_COMPILE ?= arm-none-eabi-

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
...
```

我們可發現，起始位址為`0x00000000`(對應`<main-0x8>`，這是ARM exception table的起始位址，反組譯出來的`andcs`和`andeq`指令沒有意義，我們只在意內含值)，堆疊指標MSP為`0x20001000`，PC初始值是0x00000009，為什麼不是偶數的0x8，而是0x9呢？這是因為GNU Toolchain自動處理ARM對於位址的LSB必須為1的要求。

進一步去分析編譯出來的目的檔(格式為ELF; Executable and Linkable Format)中有哪些section，由objdump的輸出可見到.text、.comment，以及.ARM.attributes等3個section(均以”.”開頭)，其中跟程式的執行相關的只有.text section。在更複雜的程式中，還會有.rodata、.data，和.bss等section。另外開發者也可以自己定義段。對這些不同段的處理，以後會有更詳細的描述。

接著我們來研究即將要燒錄(flash write)到 Flash 裡頭的二進位檔案，使用od工具程式：

```
# od -t x1 blink.bin
0000000 00 10 00 20 09 00 00 00 80 b4 83 b0 00 af 00 23
0000020 7b 60 11 4b 20 22 1a 60 10 4b 11 4a 1a 60 11 4b
0000040 4f f0 00 72 1a 60 00 23 7b 60 02 e0 7b 68 01 33
0000060 7b 60 7a 68 0c 4b 9a 42 f8 d9 0a 4b 4f f4 00 72
0000100 1a 60 00 23 7b 60 02 e0 7b 68 01 33 7b 60 7a 68
0000120 05 4b 9a 42 f8 d9 e2 e7 18 10 02 40 04 14 01 40
0000140 14 44 44 44 10 14 01 40 9f 86 01 00
0000154
```

位元順序為little-endian。

程式的燒錄和執行

當程式撰寫並順利編譯連結後，就可將產生的二進位檔案燒錄到晶片內建的 Flash 中，並試著執行測試。燒錄程式碼有不少可用的機制，本文以OpenOCD搭配OpenJTAG作為解說。

燒錄步驟如下：

1. 執行OpenOCD

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

2. 在終端機中執行以下命令，進行Flash的擦拭消除(erase)和燒錄

```
# 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. 開始測試
發現燈亮了嗎？如果沒亮的話，請仔細檢查步驟是否正確、暫存器的值是否充分設定。

加強版的程式碼

看完簡單版LED閃爍的程式碼，接著我們挑戰較進階的議題，程式碼還是讓LED閃爍，不同之處在於，本程式的目的檔中有data section。稍早提過，data section必須在開機階段，從(一開始)唯讀的Flash複製到RAM 中，才可讀寫並發揮作用，這個複製的動作當然得由自己撰寫程式來完成。對比來說，在個人電腦上面開發程式，程式設計師不必親自完成這個步驟，因為作業系統會提供載入器(loader)來處理這些工作。

由於涉及記憶體間資料的搬移以及 PC在不同記憶體上的跳躍，而經由編譯和連結所產生的仍然是一個二進位檔案，這意味著，勢必得將不同記憶體上的資料合併為單一的二進位檔案，當然，連結器腳本(*.ld)不會再像前一個程式那般簡單了。

是此，我們應當對ELF裡頭的眾多section，以及連結器對各個section的處理，以及如何對裡面的符號進行重定位(relocation)的種種原理。

每個目的檔中都包含一系列的section，每個section都有一個符號名稱和佔用空間的資訊，大多數section還包含額外資訊。若一個section包含的資料，在程式執行時期必須載入到記憶體中，則該段是可載入的(loadable)；若一個section沒有包含資料，但是必須在記憶體中預留一塊區間，則該段是可配置的(allocatable)。那些既不可載入，又不可載入的，一般都是跟程式碼除錯有關的資訊。

ARM交叉編譯器產生的ELF目的檔裡頭，一般會有`.text`、`.data`、`.rodata`、`.bss`、`.comment`、`.ARM.attributes`等不同的section。`.comment` 和`.ARM.attributes`這兩個section，不用急著看，，都是提供便於使用除錯器的資訊。當然，我們也可依據程式開發需求，自行定義一些新的section。

先建立ELF重要section的概念：

- `.text` (text section)：此section保存的內容並非是英文”text”字面意思上的「文字」，而是ARM指令的機械碼序列，一般是唯讀
- `.rodata` (唯讀data section)：此段中的資料是只讀的變數。如 C 語言中用 const 關鍵字定義的全域變數即是此類
- `.data` (有初始值的data section)：在C語言程式中，定義一個全域變數並給予初始值，則該變數會被保存於在此section中
- `.bss` (無初始值的data section)：沒有初始值的全域變數，會存放在此section

特別要注意到，上述出現三種data section，都是只有全域變數才會出現在這三種section中，而區域變數在執行時期是堆疊變數，也就是執行時期依據需要而產生，沒被使用時(比方說`{ … }`標注的範圍以外)就被銷毀，因此不需要有對應的section去特別保存這類變數。

繼續想下去，不難得出連結器對各個section有不同的處理方法。真正需要保存在二進位目的檔中的section，其實只有`.text`、`.rodata`，以及`.data`三個，而對`.bss` section，我們只需要適度紀錄該section的佔用的空間大小，並在程式啟動過程中，對其空間內容清零(也就是用`memset()`一類的函式將記憶體內容填為0)即可。按連結器手冊中的定義，text section，只讀data section，有初始值的data

section為可裝載段(loadable section)，".bss"那樣無初始值的data section叫可配置段(allocatable section)。

對連結器來說，輸入是由編譯器產生的若干個目的檔(*.o)，輸出則是一個大的映像檔案(前述的例子來說，就是*.out)。連結器將輸入的目的檔中各個section進行合併，對其中的一些符號進行重定位，然後要考慮到以下兩個議題：

1. 如何合併
以前例來說，我們需要合併各目的檔的`.text`、`.rodata`、`.data`，以及`.bss`等section，甚至還有開發者自行定義的section，又因為`.text`和`.rodata`屬性一致，可把它們合在一個section內，最終產生的各個section首位相連，另外要注意，有些section需要從4 bytes對齊(aligned)的位址開始，因此個別section之間可能會存在用來填補(padding)的位元組。

簡單來說，連結器對給定目的檔裡頭的section進行處理，輸出也是ELF section，但是已合併的section結果，而"input section"和"output section"則是GNU linker ld參考手冊裡頭使用的術語。跟合併有關的資訊如下：

- 需要保留哪些輸入section，以及其屬性
- 輸出section的屬性

2. 如何重定位
把原始目的檔合併成一個大目的檔並非連結器的全部，因為還得考慮到一個事實：程式編譯過程中，有些符號的位址並不能確定，比如某些外部變數(也就是C語言程式中，用`extern`宣告的變數)，或者我們慣用的`printf()`函式也是，都是無法在編譯過程中得知最終在執行時期，它們確切的位址，於是乎，連結器還要負責解析這些變數或符號的位址，因為這些位址只有在連結階段才能最終確定。由於變數在輸入section中的偏移(offset)事先已知，因此要確定重定位後的位址，連結器只要知道輸出section在執行時期的起始位址即可。

由上述分析，我們可看出連結腳本要處理的關鍵議題就是以下兩者：

- ELF中各section的配置
- 執行時期個別section在記憶體中的位址

對本程式而言，連結腳本必須要能告訴程式程式碼：

- data section在ELF中的偏移是多少
- data section佔用的有多大
- 程式應該把data section複製到主記憶體的什麼位址

不難看出，上面反覆出現「偏移」和「位址」兩個詞彙，前者是指在存放在儲存裝置上的位置，後者是指程式執行時期的位址(具體來說，就是虛擬記憶體; virtual memory)。GNU linker ld參考手冊
定義了以下兩種位址：

- LMA(Load Memory Address): 表示段被保存在記憶體上的位址
- VMA(Virtual Memory Address): 表示程式碼執行時期的位址

大多數情況下（不需要段搬移的情況），這類位址是相同的。比如目的檔由作業系統載入器載入的情況，或者像上面那個例子，沒有作業系統載入器，但沒有data section需要搬移，指令直接從 FLASH 上讀取並執行。但

在本例中，由於data section需要從Flash中搬移到 SRAM 中，所以data section的 LMA 和 VMA 並不一樣。text section存儲在Flash上的位址為LMA，搬移到 SRAM 中的位址為 VMA。在後面的描述中，經常會出現這兩類位址。連結器腳本的一個重要作用就是管理各個段的 LMA 和 VMA，並在必要的情況下，把有關資訊傳給程式程式碼使用。下面會看到實例：

```
//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 = . ;
}
```

連結腳本的精確語法定義可以參考手冊，這裡只是對該腳本的功能做具體分析。

首先，連結腳本定義了兩個記憶體區間（MEMORY），下面輸出段定義中會包含對它們的引用，表明某一輸出段位於某一記憶體區間。其實我們不用記憶體區間定義這個機制也可實作同樣的功能，但是使用它會使描述簡單而清晰。

連結器腳本的關鍵部分是 SECTIONS 部分，從手冊中，我們可以查到 SECTIONS 命令的標準格式規範：

```
SECTIONS
{

    section-comman
    d
```

```
    section-comman
    d
    ...
}
```

sections-command 是如下四個之一

- ENTRY 命令
- 符號賦值
- 輸出段定義
- 重疊定義

其中，"ENTRY 命令“和”符號賦值“可以出現在”輸出段定義“內部。上面的腳本中用到了“符號賦值”和“輸出段定義”兩種元素，“重疊定義”一般較少使用，“ENTRY 命令”以後有機
會再具體分析。

先來分析簡單的--”符號賦值“。前面提到過，連結器腳本必須能夠提供一些位址相關的資訊給程式程式碼，其實就是通過這個機制來實作的。程式程式碼中引用一些外部符號，在連結器腳本中給這些外部符號賦值。在連結階段，連結器負責處理這些外部符號的對應關係。”符號賦值“提供了一種連結器給程式程式碼提供資訊的途徑。上面我們看到一個特殊的符號："."，這個特殊符號叫“位置計數符”，出現在不同的位置會有不同的含義。位置計數符表示的是在包含它的元素的定址空間內到當前偏移的位址，如果出現在 SECTIONS 中，它表示相對於把各段合併後的位址(LMA)；如果在“輸出段定義”內部，則表示在該段內部的位址（VMA)。

來看核心部分吧—輸出段定義的精確格式如下：

```
section [address] [(type)] :

  [AT(lma)] [ALIGN(section_align)] [SUBALIGN(subsection_align)]

{

    output-section-comman

    d output-

    section-command

    …

} [>region] [AT>lma_region] [:phdr :phdr …] [=fillexp]
```

大家可以看到好多可選元素，一般情況下輸出段定義不會用到這嗎多。需要注意的一個細節是，所有 LMA 都是出現在 AT 關鍵字之後的。

Output-section-command 可是以下四個中的一個：

- 符號賦值
- 輸入段描述
- 數值（用於在輸出段中插入一些資料）
- 特殊關鍵字

最常用的當然就是“符號賦值”和“輸入段描述”了。

輸入Section描述

「輸入section描述」的作用就是告訴連結器如何把輸入檔案映射到主記憶體配置中。一個輸入段描述包含一個檔案名，和一系列括號包著的輸入段名，檔案名 和輸入段名都可以包含通配符：

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

最常用的輸入段描述是把所有檔案中的某個段全部輸出到一個輸出段中，例如，把所有檔案中的.text 段輸出到輸出檔案中對應的命令如下

*.(.text)

這種描述應付我們例子中的需求已經夠用了。使用統配符的副作用是各個檔案中的各個段最終在輸出檔案中的排序就由連結器來決定了，這在多數情況下是無所謂的。如果需要顯示指定某個檔案的某個段出現在最靠前的位置，就必須顯示指定檔案名 了，如下所示：

data.o(.text)

剛提到了輸入段在輸出檔案中的排序問題，事實上，在使用通配符的情況下，最終排序的結果是由連結器處理的順序決定的，如果想顯示指定順序，除了顯示指定檔案名 之外，可以指定連結器處理的順序，如按字母順序。

總結一下，從邏輯的角度上來看，上例中連結器腳本的總體結構如下所示

```
存儲區間定義

 {
    區間 1
    區間 2

    ...

 }
輸出段定義

 {
    段 1
    {
        輸入段描述 1
        輸入段描述 2

        . ..

    } > 區間 1
    符號定義

    ...
    段 2

    {

        ...
    } > 區間 n

    ...

}
```

基於以上的背景知識，先以.text section為例，分析對輸入section的處理。

.text 段會依次包含所有輸入檔案中的.isr_vector 段，.text 段，.rodata 段，在.isr_vector 上加KEEP是為了防止連結器的垃圾回收(garbage collection)功能啟用時，忽略.isr_vector section，實際上，isr_vector 段的內容相當關鍵。先分析一下吧：

從《The Definitive Guide To ARM Cortex M3》得知：「在 0 位址處提供 MSP 的初始值，然後緊跟著就是向量表（向量表在以後還可以被移至其它位置），向量表中的數值是 32 位的位址，而不是跳躍指令。向量表的第一個條目指向Reset後應執行的第一條指令」。前面提到過，STM32 中的 FLASH 可以從 0x00000000，或0x08000000 起始的位址存取。isr_vector 中的內容就對應 MSP 初始值，還有向量表的內容。我們這裡的程式這嗎簡單，就沒必要實作向量表移動了。

現在，我們先不去管 CortexM3 為實作中斷和例外機製而引入的一系列機制，先來看看中斷向量表的定義。編號為 1-15 的對應系統例外，大於 15 的則都是外部例外，具體的例外定義可以參考 CortexM3 的手冊。實作中斷控制其實還是很複雜的，但對簡單的程式來說，中斷優先級可以先不用考慮，我們只需要搞清楚記憶體上需要保存哪些值就足夠了。來看看上電後的向量表吧：

位址	例外編號	值（32 位整數）
0x0000,0000	-	MSP 的初始值
0x0000,0004	1	Reset向量（PC 初始值）
0x0000,0008	2	NMI 服務例程的入口位址
0x0000,000c	3	Hard Fault 服務例程的入口位址
…	…	其它例外服務例程的入口位址

這個表中只列出了向量表的前三項，正好對應我們的程式程式碼。

```
__attribute_ ((section(".isr_vector")))
pfnISR VectorTable[] =
{

  (pfnISR)(0x20010000), // The initial stack pointer is the top of
  SRAM ResetISR, // The reset handler
  NMIException,
  HardFaultExceptio
  n
};
```

可看出，硬體上電後，PC 會自動跳躍到 ResetISR()函式。

符號賦值

腳本中有 5 個關鍵的符號定義：_etext, _data, _edata, _bss, _ebss，這幾個符號本身沒有意義，而且也不佔用輸出段的空間，它們用做標誌資料，真正起作用的是它們的位址，能用來完成向程式程式碼提供連結器有關資訊的功能。注意這些符號表示的位址都是輸入段內相應偏移在執行時期的位址。

• _etext
該變數存在輸出檔案中.text 段的運行位址結尾，由於.text 段的連結位址和運行位址是一致的，而且.data段的起始正是.text 段的結尾，因此它的位址能代表data section在輸出目的檔上的連結起始位址。

• _data, _edata

這兩個變數分別保存在.data 段的開頭和末尾。它們位址的差值能提供段的大小資訊，正是我們所需要的。

• _bss, _ebss
這兩個變數保存在.bss 段的開頭和結尾，程式根據這兩個變數的位址來定位需要清零的區域。剛才提到了 ResetISR()函式是系統上電後執行的第一段程式碼，因此完成data section內容的搬移非它莫屬了。

```
// 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();
}
```

程式碼說明了一切。ResetISR（）函式完成了資料搬移及.bss 段的清零之後，就呼叫 main()函式，開始做正事了。

好了，做正事的程式碼大夥應該很容易看懂了吧，這裡就不再贅述了。把程式編譯出來，燒到開發板上，讓三個燈閃起來吧。

更進階的C程式

在這個程式中，我們會先執行Flash上的一小段程式，把指令以及資料都從Flash上複製到 RAM 中，然後再跳躍到 RAM 上執行。對於這個複製的步驟，有人就要問了，為啥要把指令讀入 RAM 之後再執行呢？在 RAM 上執行有啥特別的好處嗎？

就這個問題，有兩個方面的考量:

1. 在 RAM 上取得指令然後執行，速度會比較快。
2. 指令全放在 RAM 中，程式的設計會更靈活，限制更少

STM32 是哈佛結構，也就是資料匯流排(data bus)和指令匯流排(instruction)分離，從硬體角度上來講，把指令全放在Flash中，資料全放在 RAM 中沒有任何問題。但這樣做還是會給軟體設計帶來一些不便。程式員必須負責合理安排不同的存儲區域，而且在程式設計階段就必須對不同的存儲區域有合理的規劃。

大家如果開發過ARM9上的 bootloader 的話，應該都有過類似的經驗。這類 bootloader 的實作中，開始運行的第一步就是把text section，data section全部移到 RAM 中，尤其是對有些從 NAND FLASH 啟動的開發板，這一步是無法避免的。

這個例子比前一個例子只是多了一個text section的複製，看起來並不算複雜，關鍵是對中斷向量表及中斷服務例程的特殊處理，讀者可以做為練習自己解說一下。給出的參考例程中，中斷向量表和中斷服務例程還是運行在 FLASH 上的，感興趣的話可以，把除了`ResetISR()`的中斷服務例程也放在 SRAM 中。自己試試吧！

總結：通過三個點燈程式，大家基本能夠熟悉連結器腳本的使用，中斷向量表的組織，真正幹活的程式如何存取暫存器。接下來就可來點複 雜的了。

序列通訊程式

寫了三個點燈程式了，換個輸出方式吧。拿到一個開發板後，通常要做的第一件事就是把序列通訊(serial communication)調通。畢竟和LED 燈比起來，序列通訊能提供的資訊要豐富的多了。

STM32 提供了三個 USART(Universal synchronous asynchronous receiver transmitter)port USART1, USART2, USART3，以及兩個 UART(Universal asynchronous receiver and transmitter)port UART4, UART5。該port的功能還是很豐富的。除了 能用於非同步通信之外，還可提供諸如同步通信，智能卡介面，紅外信號接收等。在通信的實作上，可以支援中斷接受，DMA 資料傳送等高級功能。在這裡，我們沒必要使用這些高級功能，最簡單的非同步發送和接受就夠了。USART1 能達到的最高速率是 4.5Mbits/s，其他port的最大速度是 2.25Mbit/s。

硬體準備

實作雙向非同步通信最少需要兩個管腳，接收腳(Receive Data In, RX)和接收腳(Transmit Data Out, TX)。

- •RX：接收腳是串行資料的輸入口，資料恢復過程中，為了區分有效資料和噪聲，採用了過採樣技術。
- TX：當傳送器沒有enable時，該輸出口可以作為通用輸入輸出port(GPIO)使用。當傳送器enable但沒有資料需要傳送時，該port為高電位。

我們先來指定一下我們期望實作的port參數吧

- baud rate為 9600bps
- 資料位 8，停止位 1，無硬體握手，無流量控制在開發板上，可以查到 USART1 是引出的

軟體出馬

來看看實作序列通訊資料發送需要幹些啥吧。

1. 設定 USART_CR1 的 UE 位為 1 來enable USART
2. 設定 USART_CR1 的 M 位來定義資料位長度
3. 設定 USART_CR2 中的相關位來定義停止位長度
4. 設定 USART_CR3 中的相關位來設定enable DMA 模式
5. 設定 USART_BRR 來選擇期望的baud rate
6. 設定 USART_CR1 中的 TE 位來發送一個 idle 幀作為發送的開始
7. 寫資料到 USART_DR 暫存器，STM32 會把資料發送出去。
8. 重複步驟 7，直到資料全部發送完畢。把最後一個資料發送之後，等待 TC 位為 1。TC 位為 1 顯示最後一幀的發送已結束，這個步驟是為了保證資料完整的被發送。

可看出，上面的步驟中，都是操作暫存器，暫存器的值怎確定？當然是要查手冊了。其中最複雜的是baud rate暫存器值的計算。

查閱手冊，可以發送器和接收器的baud rate是相同的，baud rate的計算公式如下：

$$\text{Tx /Rx baud} = \frac{f_{ck}}{16 * \text{USARTD}}$$

IV

fck 是週邊硬體的輸入時鐘（USART2，3，4，5 是 PCLK1，USART1 為 PCLK2，這就是為什麼 USART1 的最高

速度跟其它port不一樣的原因）。USARTDIV 是一個無符號的定點小數，下面就來看看這個定點小數是怎嗎由
USART_BRR 暫存器來定義的：

USART_BRR 暫存器中定義了 DIV_Mantissa 和 DIV_Fraction，分別用來定義 USARTDIV 的整數部分和小數部分
USARTDIV =DIVMantissa DIVFraction /16
在實際的計算中，難免會遇到取整，這個時候就必須要注意誤差不能太大，在 STMicroelectronics的手冊中有對不同baud rate和
時鐘頻率下誤差的分析。

下面來計算一下對我們現在的需求，USART_BRR 暫存器的值應該是多少吧。值得注意的是，STM32 重啟後預設的 fck 是 8MHz，在這裡先不管時鐘設定的細節，因為時鐘的設定還是相對比較麻煩的，下一個例子會專門予
以討論。

在開發板上，fck=8MHz, 我們期望的 baud = 9600, 則 USARTDIV=52.083，進而 DIV_Mantissa=0d52=0x34，DIV_Fraction=0.083*16=0x1，則 USART_BRR=0x341

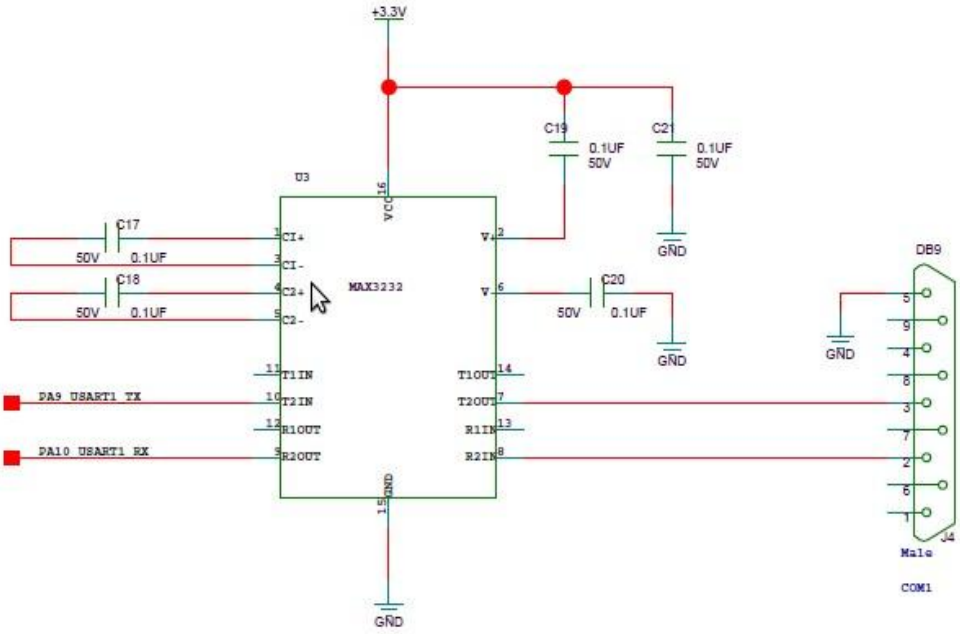
好了，暫存器的值確定下來沒有問題了，查一下暫存器的位址，寫程式碼實作吧，這次再來一個版本的 helloworld：在序列通訊上輸出一行 helloworld。

USART1 暫存器起始位址是 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，
往裡面填值吧。

另外還需要注意的是，要像前面的例子設 IO 口輸入輸出模式那樣，設定序列通訊線對應管腳的工作模式。如下面的原理圖所示，Tx 腳復用 GPIOA 的 9 腳，Rx 腳復用 GPIOA 的 10 腳。我們必須像設定 GPIO 口輸入輸出模式那樣，設定Tx和Rx腳的工作模式。需要注意的是，Rx 腳為輸入模式，與 GPIO 口設定的可選模式相同，而對Tx腳這樣的輸出管腳，需要設定專門的工作模式(Alternate function output Push-pull 或 Alternate function output Open- drain)，而不能設定為 General purpose output 模式。對序列通訊輸出而言，需要選擇 Alternate function output Push-pull，據此可確定 GPIOA_CRH 暫存器的值。

最後，千萬不要忘了啟用USART1，GPIOA的時鐘，就像前面幾個例子，需要啟用GPIO port的時鐘那樣。

在這裡還跟讀者分享一個除錯序列通訊的經驗，如果遇到序列通訊無輸出的情況，可以把 PC 端的序列通訊軟體的baud rate設一個比較低的值，如果是因為baud rate的原因導致無輸出的話，PC 端較低的 baud rate就會輸出亂碼，這樣就可以確定問題的根源，為進一步分析提供線索了。



RS232

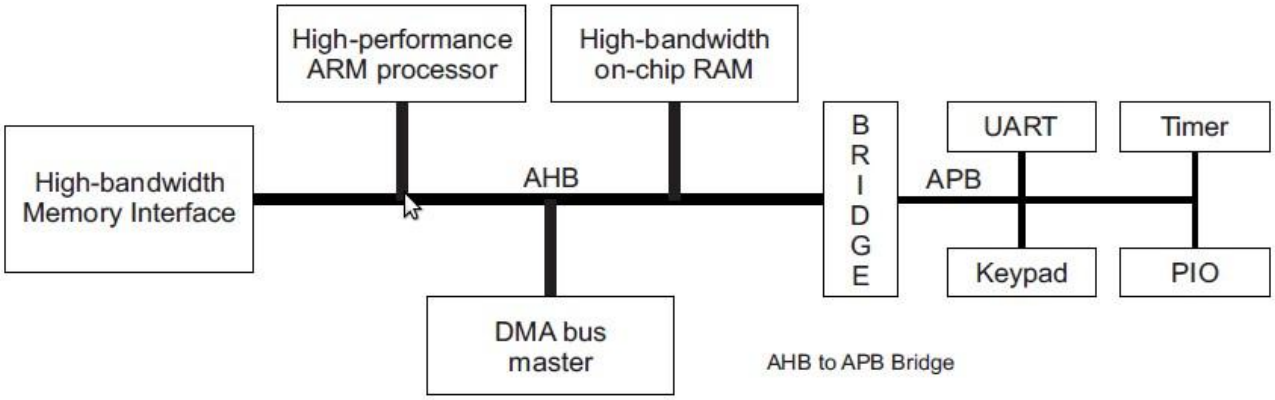
STM32 匯流排介紹

AMBA 匯流排規範

AMBA(Advanced Microcontroller Bus Architecture) 是 ARM 公司為高性能微控制器定義的片上通信標準，在標準中，定義了三種匯流排：
1. Advanced High-performance Bus (AHB)
2. Advanced System Bus (ASB)
3. Advanced Peripheral Bus (APB)

從名字就可看出各個匯流排的特性，其中 ASB 與 AHB 的作用類似，都可用於連接系統中的高速模組，但 AHB比 ASB 更適合高性能的場合，因此，很多 SoC 中只使用了 AHB。而 APB 則被用於低功耗的週邊硬體模組。

符合 AMBA 規範的微控制器一般包含一個高性能的系統骨幹匯流排，用於處理器上記憶體和其它 DMA 設備的互聯，而且還連接一個橋接器，將 APB 上個低速外部設備連接到 AHB 上。下圖是一個典型的符合 AMBA 規範
的系統：



APB 本身主要用於低帶寬周邊週邊硬體之間的連接，AHB-to-APB 橋是 APB 上唯一的主模組，也是 AHB 上的從模塊，其主要功能是鎖存來自 AHB 的位址、資料和控制信號，並提供二級譯碼以產生 APB 外圍設備的選擇信號，從而實作 AHB 協議到 APB 協議的轉換。

對學習 STM32 來說，我們沒有必要深究 AMBA 的規範細節，只需要知道 STM32 中那些模組連接在 AHB上，哪些模組連接在 APB 上，如何enable它們的時鐘就行了。對好奇的讀者，可以參考 ARM 公司的 AMBA 規範。

STM32中的匯流排

STM32 中的 AHB 與其它 ARM 核心的 SoC 比起來沒啥特別之處，我們需要特別關心的是 APB 匯流排上都連接了那些設備。

APB 在 STM32 中分為兩種：APB1 和 APB2，分別用來連接低速週邊硬體和高速週邊硬體。
• 連接在 APB1 上的設備有：
CAN, USB, I2C1, I2C2, UART2, UART3, UART4, UART5, SPI2, SPI3, Watchdog, Timer2, Timer3, Timer4, Timer5
• 連接在 APB2 上的設備有：
GPIO_A-E , USART1, ADC1, ADC2, ADC3, Timer1, Timer8
這些資訊來源於 STM32 的 Datesheet。搞清楚各個模組連接那個匯流排，對實作精確的時鐘控制非常重要。前面例子中已經提到過，要操作一個模組，務必先enable該模組連接匯流排的時鐘。

時鐘例程

STM32 中的時鐘

其實，時鐘模組是應該最先仔細學習的內容，它是很多模組工作的基礎，但前面的例子一直避免接觸它們，主要是因為其複雜性和不直觀性。經過前面幾個例子的實驗，現在可以通過設計一個時鐘相關的程式來熟悉它。本例的目的是使 STM32 全速（72MHz）工作，並通過序列通訊輸出消息，同時控制 3 個 LED 交替亮滅。注意本
程式將會使用和以前程式一樣的延時函式，這樣就可以很直觀的通過比較燈兩滅的速度來比較 CPU 運行的速度了。

前面序列通訊的例子中已經提到過，系統重啟後預設的時鐘是 8MHz，實際上，STM32 的系統時鐘可以由以下三
個時鐘源驅動：

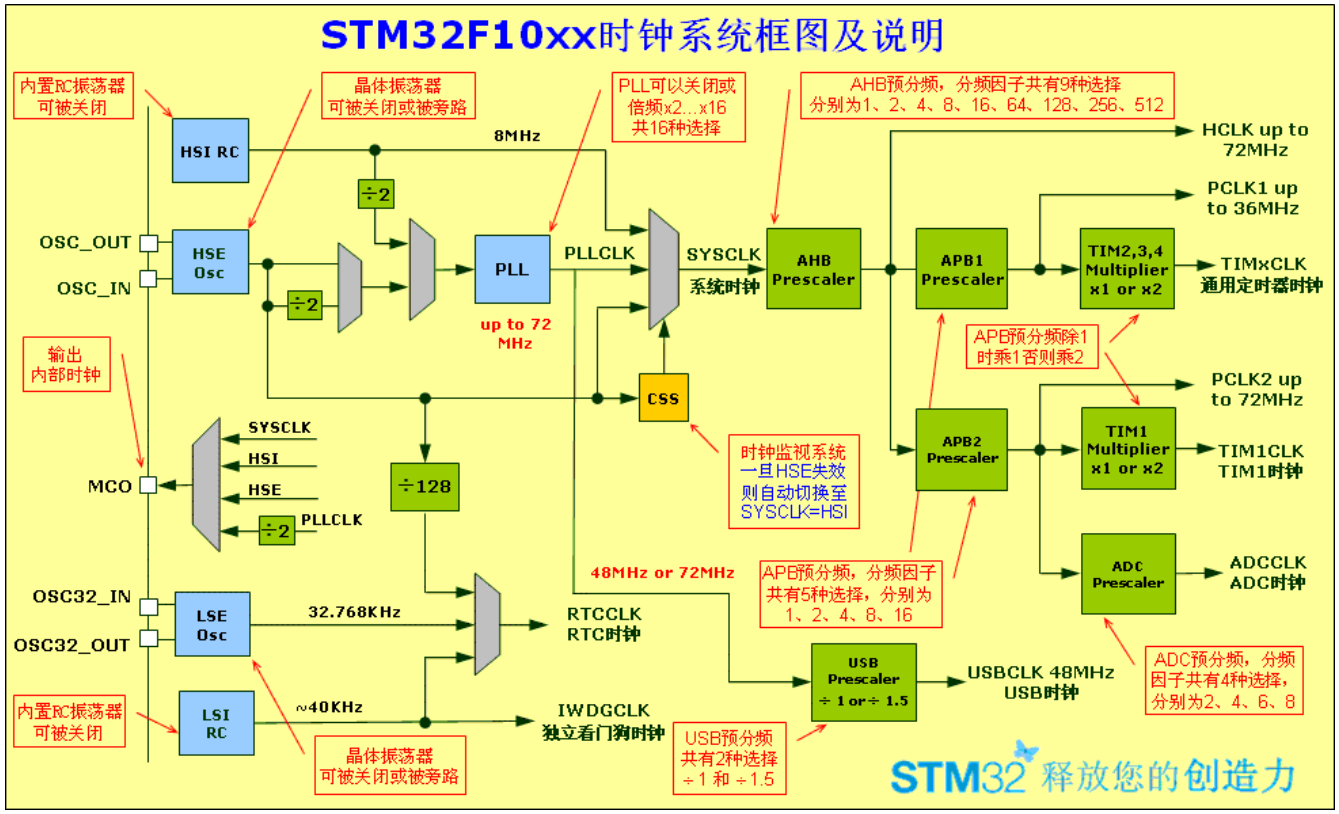
1. 高速內部 RC 振盪器(HSI oscillator clock)，頻率為 8MHz
2. 高速外部振盪器(HSE oscillator clock)
3. 鎖相環時鐘(PLL clock)

除此之外，還有兩個二級時鐘源

1. 40kHz 的內部 RC 振盪器(LSI)，用來驅動看門狗。而且，該時鐘也可用做實時時鐘，實時時鐘用來將處理器從停止/休眠狀態喚醒。
2. 32.768kHz 的低速外部晶振（LSE)，也可以用來驅動實時時鐘。

各個時鐘都可以獨立的打開或關閉，進而降低系統的功耗。系統重啟後，預設的系統時鐘就是高速內部 RC 振盪器。

下圖的各個時鐘的頻率限制，配置都有很清晰的說明。



下面先對時鐘模組的輸出，即產生的各個時鐘的用途做簡要分析。

- SYSCLK

系統時鐘，為 PLL 的輸出，是 HCLK，PCLK1，PCLK2，TIMxCLK，TIM1CLK，ADCCLK 的基礎

- HCLK

AHB Clock，AHB 匯流排的時鐘

- PCLK1

APB1 low speed clock，週邊硬體匯流排的低速時鐘

- PCLK2

APB2 high speed clock，週邊硬體匯流排的高速時鐘

- TIMxCLK

Timer2, Timer3, Timer4 的時鐘

- TIM1CLK Timer1 的時鐘
- ADCCLK

ADC 模組的時鐘

- USBCLK

USB 時鐘，必須是 48MHz

- RTCCLK RTC 時鐘
- IWDGCLK

獨立看門狗時鐘

- MCO

輸出時鐘

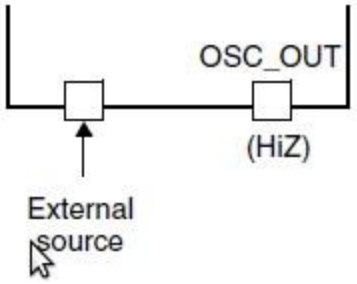
在進行時鐘配置的時候，務必注意各個時鐘的頻率限制，圖中已經註明了一些，還需要注意的是在使用HSI作為 PLL 的輸入時，SYSCLK 最大值為 36MHz。

時鐘源

接下來再看看時鐘源有關的管腳。前面提到的三個系統時鐘驅動信號，只有 HSE 有對應的外部管腳。而兩個二級時鐘源中，32.768KHz 的外部晶振需要兩個管腳。

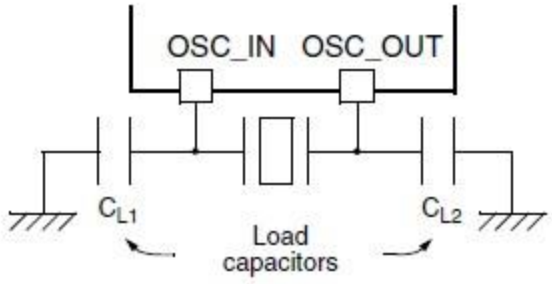
HSE 信號有兩種產生方式

- 直接輸入時鐘信號



在這種模式下，OSC_OUT 管腳必須是高阻態。

- 外接晶振/陶瓷諧振器產生時鐘信號



這種模式是最常見的用法，在進行 HSE 相關的時鐘配置之前，一定要保證晶振起振並進入穩定狀態。LSE 的管腳解法與 HSE 的類似，這裡不再贅述。

時鐘的穩定性

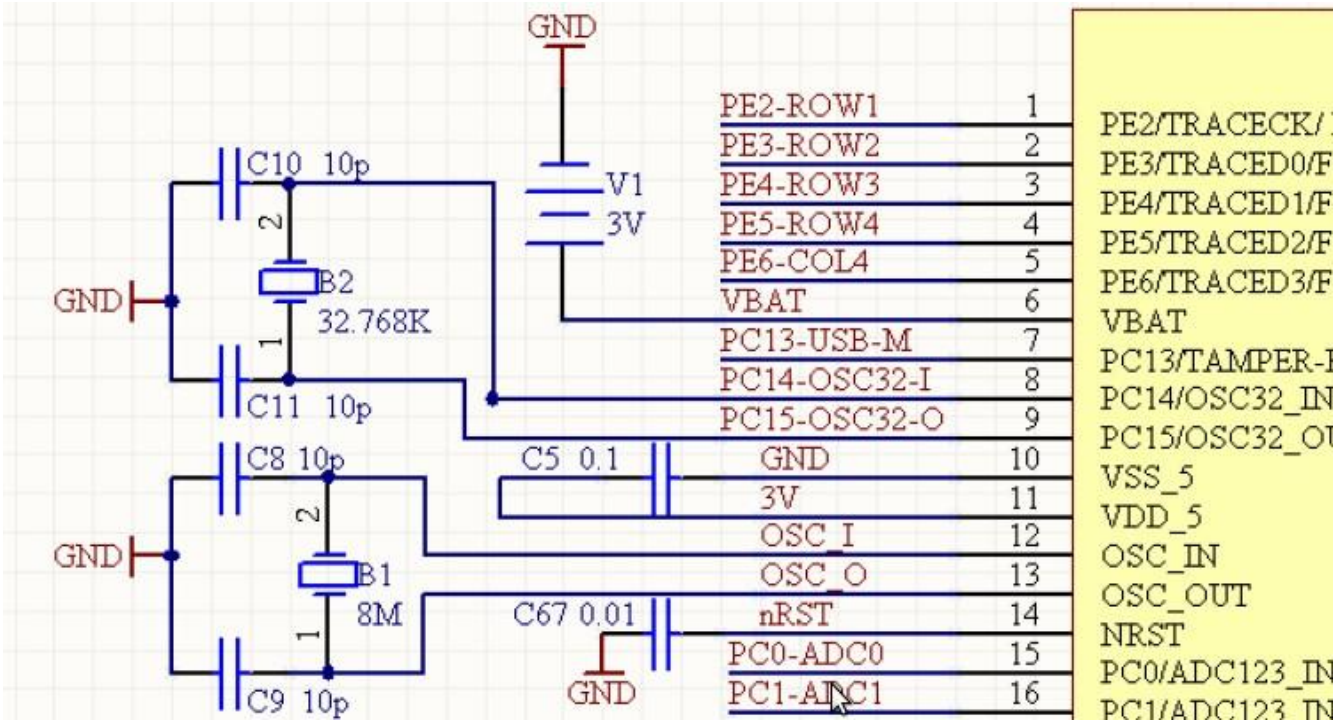
STM32 內部的時鐘源（HSI, LSI)雖然使用方便，但精度沒有外部時鐘高，在實際使用中，如果使用到了內部時鐘源，需要進行校準。這方面的詳細資訊可以參考 STMicroelectronics的手冊。

STM32 時鐘的配置步驟

搞清楚了 STM32 中的各個時鐘後，接下來了解一下如何配置這些個時鐘的頻率以及如何enable它們。在前面的例子中，我們只是簡單的提到了要enable某某時鐘，在這一節會更精確的闡述各個時鐘的配置和enable。

先來看看開發板上時鐘相關的電路。

OSC_IN，OSC_OUT 外接了晶振/陶瓷諧振器，晶振的頻率為 8MHz，OSC32_IN 和 OSC32_OUT 同樣外接了晶振/陶瓷諧振器，晶振頻率為 32.768kHz。可以看出，為了得到 72MHz 的系統時鐘，PLL 的倍頻係數必須設為 9。



來看時鐘初始化的詳細步驟吧：

1. 打開外部高速時鐘晶振 HSE
2. 等待外部高速時鐘晶振穩定
3. 配置時鐘相關的參數
   - PLL 的輸入時鐘源設為 HSE
   - PLL 的倍頻係數
   - PCLK1，PCLK2 相對於系統時鐘的降頻係數，即 APB1 和 APB2 的頻率
   - HCLK 相對系統時鐘的降頻係數，即 AHB 的頻率
4. 啟用PLL
5. 等待PLL穩定
6. 選擇PLL作為系統時鐘
7. 等待硬體確認系統時鐘已經是 PLL 時鐘

通過以上步驟，我們就可以成功的把預設 8MHz 的系統時鐘換成 72MHz，由 PLL 倍頻產生的時鐘了。接下來配置序列通訊輸出資訊，配置 GPIO 點亮 LED 的步驟跟前面的例子就沒啥區別了。唯一需要注意的是 PCLK2 和 PCLK1 變了，即序列通訊的時鐘源發生了變化，跟baud rate相關的暫存器值需要重新計算。而且，系統時鐘是以前的 8 倍，延時函式的執行時間更短，LED 亮滅的頻率更高了。

使用 STMicroelectronics的韌體函式庫編程

通過前面幾個例子的解說，可以發現直接操作暫存器來編程還是相當枯燥的，而且很容易出錯。如果能把對暫存器操作達到一些目的的程式碼封裝成比較通用的函式，無疑會大大方便我們的編程工作。誰能提供最可靠的封裝函式呢？當然是 ST 了。實際上，ST 確實提供這樣的函式，官方的名字叫通用週邊硬體函式庫（Standard Peripheral Libarary)。很多人覺得 ST 提供的週邊硬體函式庫效率不夠高，其實大可不必這嗎擔心。使用 ST 提供的函式庫，可以讓我們把更多的精力集中在需要我們創新的地方，而不是去枯燥的調暫存器。這些機械、無聊的事情就交給 ST 來完成吧。

標準週邊硬體函式庫簡介

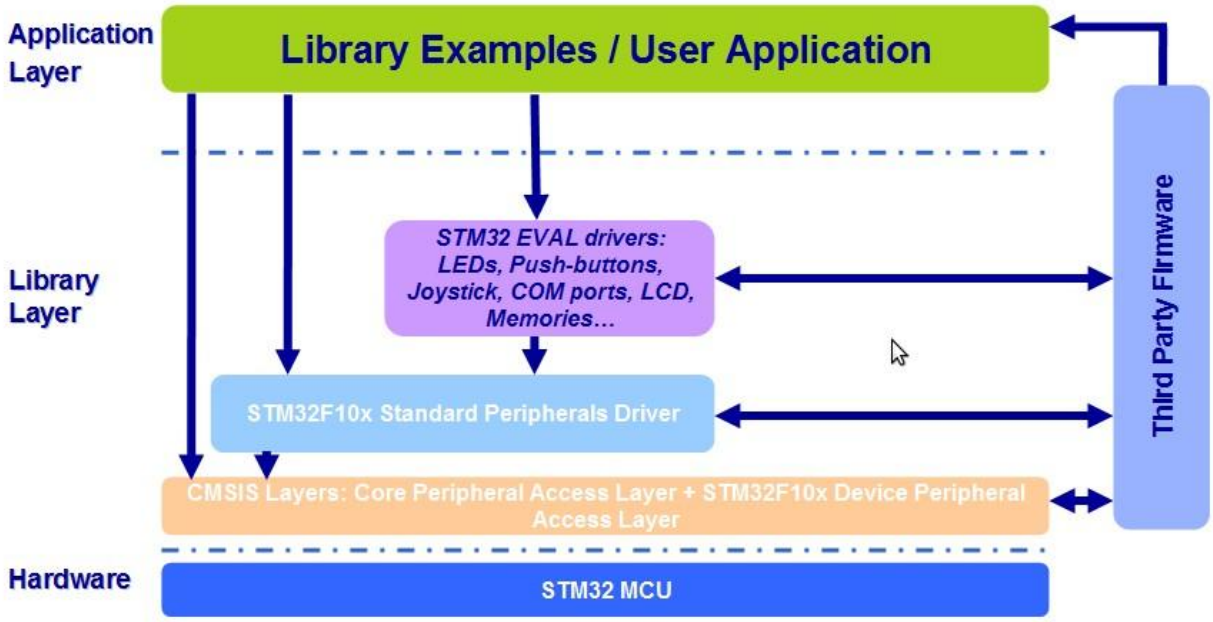
通用週邊硬體函式庫（STM32F10x Standard Peripherals Libray)的詳細介紹可以參考 ST 提供的文件，這裡只是簡要

介紹一下對理解本例子很重要的資訊。

首先要強調的一點是，ST 提供的標準週邊硬體函式庫是完全符合 CMSIS（Cotex Microcontroller Software Interface Standard）的。CMSIS 是 ARM 公司針對 CortexM 系列處理器定義的一個與晶片製造商無關的，獨立的硬體抽象

層，基於符合CMSIS 的韌體函式庫開發程式，程式碼復用的好處是顯而易見的。只要其它晶片供應商的 CortexM 處理器也提供符合 CMSIS 的韌體函式庫，我們基於 STMicroelectronics的韌體函式庫開發的程式就很容易移植過去。

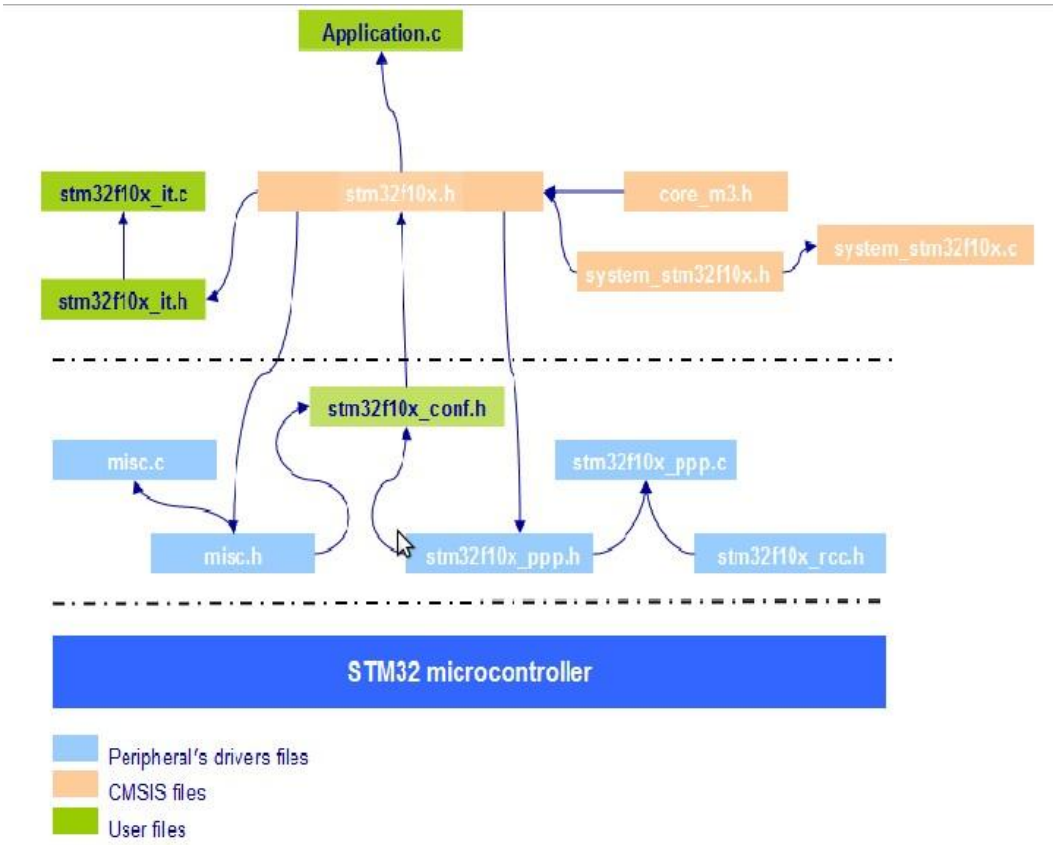
來看看 ST 週邊硬體標準函式庫的架構吧，下圖摘自 STMicroelectronics的官方文件



要掌握 ST 標準韌體函式庫的使用，關鍵是掌握“STM32F10x Standard Peripherals Driver”和"CMSIS Layers”。其它部分就輪到我們來發揮自己的創意了。

標準週邊硬體函式庫的檔案依賴關係

下圖同樣摘自 STMicroelectronics的官方文件，它詳細的畫出了使用 ST 韌體函式庫編程時各種檔案的依賴關係，我們在這一節實作的例子就遵循了這個依賴關係。



從上圖我們可以看出，每個週邊硬體都對應有相應的驅動檔案，不好分類的就都歸為 misc 類了。理解了大致框架之後，大家可以花點時間讀一下 ST 關於標準韌體函式庫的文件，磨刀不誤砍柴功嘛。

週邊硬體函式庫函式的使用

下面總結一下基於週邊硬體函式庫怎樣完成對一個週邊硬體的初始化，配置以及操作。假設週邊硬體名是 PPP

1. 定義一個 PPP_InitTypeDef 結構體類型的變，如：

   ```
   PPP_InitTypeDef PPP_InitStructure;
   ```

   該結構體一般定義在data section，能被用來初始化一個或多個 PPP 實例。
2. 給該機構體的各個成員賦值，有兩種方法來完成這個工作
   - 給每個成員單獨賦值，如

     ```
     PPP_InitStructure.member1 = val1;

     PPP_InitStructure.member2 = val2;

     PPP_InitStructure.member3 = val3;
     …
     ```

   - 呼叫函式給各個成員給予預設值，然後修改某些成員的值，如

     ```
     PPP_StructInit(&PPP_InitStructure);

     PPP_InitStructure.member1 = val1;
     ```

3. 呼叫 PPP_Init()函式初始化 PPP 週邊硬體

   ```
   PPP_Init(PPP, &PPP_InitStructure);
   ```

4. enable PPP 週邊硬體

   ```
   PPP_Cmd(PPP, ENABLE);
   ```

5. 呼叫其他介面完成需要的工作

   ```
   PPP_xxx(PPP, …);
   ```

在前面的例子中，我們曾經提到過，對任何一個週邊硬體操作之前，我們必須先enable該週邊硬體對應的時鐘，使用標

準週邊硬體函式庫同樣需要記住這一點，如下幾個週邊硬體函式庫函式可以用來enable各個週邊硬體的時鐘

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

在使用某個週邊硬體的過程中，有時我們會需要將一個週邊硬體的所有暫存器全部Reset到初始狀態，函式 PPP_DeInit()可以幫我們完成這個工作。而且在完成設備的初始化之後，如果我們需要修改某個設定，可以修改 PPP_InitStructure 的成員後再次呼叫 PPP_Init()函式。

週邊硬體函式庫編程

接下來就到了介紹如何使用 ST 標準週邊硬體函式庫來編程的時候了。程式的基本要素其實跟我們前面不用標準週邊硬體函式庫一樣，只不過不用我們自己去寫程式碼完成data section搬移，暫存器操作了。ST 週邊硬體標準函式庫中提供了參考的引導程式，只需要稍加修改就可以用 GNU 的 toolchain 來編譯連結。ST 週邊硬體標準函式庫預設支援的開發工具有以下5 種：

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

引導程式 (bootstrap) 我們可以直接使用 RIDE7 的版本，因為我們的晶片屬於 high density 系列，所以引導程式程式碼為 startup_stm32f10x_hd.s，這個程式碼跟我們前面的例子中的程式碼原理基本類似，也是完成data section的複製，bss 段的清零等工作。但這個程式碼比前面例子中的引導程式要完善多了，在程式碼中定義了完整的例外向量表，而且還定義了一個預設例外處理函式。在完成必須的初始化操作之後，啟動程式碼會呼叫 SystemInit（）函式去實作開發者自定的初始化任務，然後呼叫 main（）函式。

連結腳本就需要自己來寫了，其實道理跟前面的例子也是一樣的，要注意的是對例外向量的處理，在這裡不再贅述，請直接閱讀 stm32f103vet6.ld。

來看看程式是如何組織的吧。前面的檔案依賴關係圖上我們可以看出 stm32f10x.h 是應用程式要包含的頭文件，而 stm32f10x_conf.h 包含一些可配置的選項，我們必須確保宏 USE_STDPERIPH_DRIVER 是定義了的，這樣才能在 stm32f10x .h 中包含 stm32f10x_conf.h，而 stm32f10x_conf.h 又會包含各個週邊硬體驅動的頭檔案。最後的
結果就是應用程式只需要包含 stm32f10x.h 一個檔案，就可以使用所有週邊硬體驅動定義的函式。如果不定義宏 USE_STDPERIPH_DRIVER，應用程式就只能通過存取暫存器的方法來操作週邊硬體了。除此之外，還要記得修改
stm32f10x.h 檔案以正確地設定晶片的型號。

下面是例子程式的目錄組織結構

```
.
|-- 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
```

讀者需要自己花時間去熟悉各個檔案中的內容。這裡僅介紹兩個重要的函式，SystemInit()和 main()
SystemInit()定義在 cmsis/system_stm32f10x.c 中，是 ST 官方提供的實作，裡面最複雜的部分就是時鐘設
置，仔細看看，是不是跟我們前面 clock 例子基本一樣？最終系統時鐘也設定成 PLL 輸出，72MHz。
main()函式是應用程式的核心心，ST 提供了很多例子做參考，但 STMicroelectronics的例子中假設我們使用的是 ST 提供的參考
開發板，往往跟我們自己的開發板配置並不一致，因此我們必須自己來實作這個函式。我們當然可以參考 ST
的"STM32 EVAL Driver”，自己封裝一些類似的介面，對自己的開發板上的週邊硬體進行存取，如 LED，序列通訊，按鍵

等。在本例中就這樣做了，LEDInit(), LEDOn(), LEDOff() 就是這樣的三個函式。仔細看看週邊硬體函式庫函式的使用，是不是跟前面介紹的一致？

通過使用週邊硬體函式庫函式，避免了讓我們自己陷入無趣的暫存器設定工作，可以把更多精力放在能發揮創造力的領域--應用設計，何樂而不為呢？如果你仔細的除錯過前面的例子程式，就會理解除錯暫存器設定是多煩人的工作，現在，讓我們正式逃離吧。

本節的例子程式的功能就是閃爍三個 LED 燈，由於使用了週邊硬體函式庫，main()函式的實作清晰了很多，更重要的是，由於避免了暫存器操作，並且使用了符合 CMSIS 規範的週邊硬體函式庫，程式還具有了可移植性。

感興趣的讀者可以仔細參考例子，實作其他功能的程式。