**CMSIS厂商库移植规范V2.0**

# 1简介

本文介绍厂商库整理和上传的规范，目的是让开发人员和上传人员有据可遵，避免出现一些因沟通不到位导致的返工等现象。从而提高工作效率。

整理和上传虽然是分开的，可能由不同的人来完成，本规范将两者紧密结合在一起，只要将厂商库按规范整理好，上传会变轻松。

# 2目录组织

目录组织是本规范的关键，下面以东芝（Toshaba）厂商库为例说明目录组织：

命名规范 /$厂商名$\$系列名$, 比如 / Toshaba \ TMPM370.

以下是TMPM370目录下面的目录组织

│ clean.bat

│

├─cmsis\_lib

│ ├─all

│ │ ├─example

│ │ │ └─project

│ │ │ ├─coide

│ │ │ │ template.coproj

│ │ │ │

│ │ │ └─src

│ │ │ main.c

│ │ │

│ │ └─lib

│ │ └─project

│ │ ├─coide

│ │ │ template.coproj

│ │ │

│ │ └─src

│ │ main.c

│ │

│ ├─cmsis\_boot

│ │ └─document

│ │ cmsis\_boot.xml

│ │ description.html

│ │

│ ├─cmsis\_core

│ │ └─document

│ │ cmsis\_core.xml

│ │ description.html

│ │

│ ├─GPIO

│ │ ├─document

│ │ │ description.html

│ │ │ gpio.xml

│ │ │

│ │ └─example

│ │ └─GPIO\_LED

│ │ ├─coide

│ │ │ template.coproj

│ │ │

│ │ ├─doc

│ │ │ example.html

│ │ │ GPIO\_LED.xml

│ │ │

│ │ ├─ewarm

│ │ │ templet.ewd

│ │ │ templet.ewp

│ │ │ templet.eww

│ │ │

│ │ └─src

│ │ Example\_LED.c

│ │ main.c

│ ├─ADC

│ │ ├─…….

│ │ …….

│ │

│ ├─SPI

│ │ ├─…….

│ │ …….

│ │

│ ├─I2C

│ │ ├─…….

│ │ …….

│ │

│ ├─……

│ │ ├─…….

│ │ …….

│ │

│ │

│ └─lib

│ ├─cmsis\_boot

│ │ │ system\_TMPM370.c

│ │ │ system\_TMPM370.h

│ │ │ TMPM370.h

│ │ │

│ │ └─startup

│ │ ├─arm

│ │ │ startup\_TMPM370.s

│ │ │

│ │ ├─coide

│ │ │ startup\_coide.c

│ │ │

│ │ └─iar

│ │ startup\_TMPM370.s

│ │

│ ├─cmsis\_core

│ │ core\_cm3.c

│ │ core\_cm3.h

│ │

│ ├─inc

│ │ tmpm370\_adc.h

│ │ tmpm370\_cg.h

│ │ tmpm370\_fc.h

│ │ tmpm370\_gpio.h

│ │ tmpm370\_ofd.h

│ │ tmpm370\_tmrb.h

│ │ tmpm370\_uart.h

│ │ tmpm370\_vltd.h

│ │ tmpm370\_wdt.h

│ │ tx03\_common.h

│ │

│ └─src

│ tmpm370\_adc.c

│ tmpm370\_cg.c

│ tmpm370\_fc.c

│ tmpm370\_gpio.c

│ tmpm370\_ofd.c

│ tmpm370\_tmrb.c

│ tmpm370\_uart.c

│ tmpm370\_vltd.c

│ tmpm370\_wdt.c

│

├─config

│ ├─debugger

│ │ TMPM37x.xml

│ │

│ └─device

│ TMPM370FDFG.xml

│ TMPM370FWFG.xml

│ TMPM370FYFG.xml

│

└─flash

│ TMPM37x\_128.elf

│ TMPM37x\_256.elf

│ TMPM37x\_512.elf

│ TMPM37x\_64.elf

│

└─source

├─framework

│ FlashAlgorithm.h

│ FlashExecFnc.c

│ target.ld

│

└─TMPM37x

│ FlashAlgoFncs.c

│ FlashAlgoInfo.c

│

└─gccProject

│ makefile

│

├─bin

└─obj

# 3目录说明

以上目录组织是一个厂商库的最小目录组织，下面具体说明每个目录和目录下的文件的作用。

1. clean.bat清除CoIDE、MDK、IAR工程生成的临时文件
2. config里面保存是一些具体芯片信息相关的文件，包括两个方面一个是芯片Device，另一个是调试模型。芯片Device里面包含芯片信息，一个芯片对应一个Device文件。调试模型里面包含调试器可以访问的memory map，一般，一个系列一个文件。
3. flash跟目录下是下载算法，这些算法是自动生产的，source目录下有framework和$series$ （表示系列名，这里是TMPM37x）；framework目录下的东西是固定的，不需要有任何修改，真正需要编写和修改的是$series$下的两个文件，FlashAlgoFncs.c和FlashAlgoInfo.c。完成这两个文件的编写之后，执行gccProject目录下的makefile，自动生产算法。（注意FlashAlgoInfo里面的宏定义要和makefile匹配）
4. cmsis\_lib 包含所有的厂商库源码、例子、文档等，这个目录下有三类目录：

a是lib目录，放所有的源码，lib之下的目录都是固定的，参考模板就行了，其他代码一般厂商都会提供，\cmsis\_lib\lib\cmsis\_boot\startup\coide\目录下的启动代码需要，开发人员自己编写。

b是all目录，这个目录是用来上传的，目录下有两个子目录，一个是lib一个是example，

其中lib里面包含上传组件的工程，example目录包含上传例子的工程（注：因为用于上传，所以只要有CoIDE工程就可以了）

c是组件目录，在这里进行组件拆分，将lib按照外设拆分成一个个组件，每个外设组件一个目录，所以会有多个组件目录：比如 SPI GPIO UART ADC…….等等；一般一个组件包含一个c文件，一个h文件。对于特殊用途的文件，比如：xxx\_common.h，xxx\_config.h，xxx\_lib.h，xxx\_assert.h，xxx\_assert.c等，可以根据实际情况删减（删减的一般是配置文件，他用于配置各个组件的头文件是否要包含，由于CoIDE可以自动添加组件依赖关系，所以不需要这个文件。比如ST的）并合并到cmsis\_boot。

组件拆分好了之后，然后开始想每个组件目录填充文档、例子、工程等。其中cmsis\_boot和cmsis\_core仅仅有上传文档就行了，不需要提供例子。

其他组件以GPIO为例来说明：

GPIO下包含两个目录，document和example。

其中document里面定义了两个文档，一个是帮助文档，简单的描述这个组件功能；另一个是xml文档，用于配合定义上传时的规范，包含了 组件名字，组件描述，上传时的CoIDE目录组织，组件依赖关系和该组件支持的芯片。

Example目录下 包含这个组件的所有例子，例子可以有多个，一个例子一个目录，比如GPIO\_LED、GPIO\_BUTTON等等。具体的例子目录下面包含三类目录，a是例子源码，b是例子文档，c是工程目录。

a 例子源码里面可以包含多个源码文件，文件可以是c文件，也可以是h文件，还有一个main.c文件。

b 例子文档包含两个，一个帮助文档，详细描述这个例子的功能和使用的方法包括如何搭建硬件环境、如何操作，以及最后的现象，还有原理图。可以包含图片，甚至视频链接。

c 工程目录 可以有多个 至少需要CoIDE的工程，工程目录下一般就是一个工程文件（.coproj CoIDE工程）。

# 4移植过程

移植包含以下几个基本步骤：资料和环境准备阶段（半天）、芯片信息整理（半天）、芯片算法文件编写测试（1-3天）、组件整理（1天）、例子整理（1天）、例子测试（1天）、上传（1天）。总共预计时间为6-8天。其中芯片算法文件编写存在时间上的风险，如果没有MDK的参考，1天时间可能完不成。

下面详细介绍这几个步骤：

## 4.1资料和环境准备

需要准备以下资料或者环境：

1. 在官方网站下载相应的厂商库源码、参考手册、数据手册；

2. 在https://github.com，上注册一个帐号。

3. 然后需要安装git。安装方法如下：http://help.github.com/win-set-up-git/

4.从https://github.com/coocox/MPLib上Fork coocox的厂商库项目。从git://github.com/coocox/mplib.git上clone 厂商库源码。

1. 安装CoIDE（ [http://www.coocox.org](http://www.coocox.org/) ），MDK（www.**keil**.com） 和IAR（[www.**iar**.com](http://www.iar.com)）
2. 安装 makesection并将bin加到环境变量
3. 安装 notepad++（也可以是其他的文本编辑器）
4. 安装一个GCC，并将bin目录加到环境变量

注意：外包时，相应文件CooCox提供。

仓库准备：

1. 将\resource\template文件夹复制到/$manufacturer$/目录下。这个$manufacturer$是你说移植的厂商库厂商的名字，常见的有 ST、TI、Atmel、NXP、holtck、Nuvoton、Energy Micro、ARM（这是个特殊的厂商，M0、M3、M4）。然后将template重命名为$series$，这个$series$表示一个系列，比如NUC1xx、M051、LPC17xx、STM32F1xx等。
2. 打开从官网下载或者CooCox提供的厂商库代码，并将代码移到仓库中对应的目录下面，

具体如下：

a.先删除\$manufacturer$\$series$\cmsis\_lib\lib\cmsis\_core下面的内容，然后将CMSIS\CMx\CoreSupport 移到 \$manufacturer$\$series$\cmsis\_lib\lib\cmsis\_core目录下面。

b.先删除将\$manufacturer$\$series$\cmsis\_lib\lib\cmsis\_boot 下面的内容（保留startup\coide\startup\_coide.c），然后将

CMSIS\CMx\DeviceSupport\$manufacturer$\$series$\ 目录中的内容移到

\$manufacturer$\$series$\cmsis\_lib\lib\cmsis\_boot 下。

c.将源码中的头文件移到 \$manufacturer$\$series$\cmsis\_lib\lib\inc下面

d. 将源码中的源码文件文件移到 \$manufacturer$\$series$\cmsis\_lib\lib\src下面

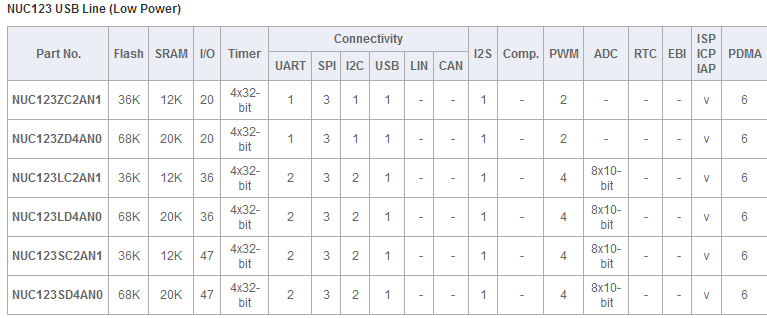
## 4.2芯片信息整理

芯片文件的路径为\$manufacturer$\$series$\config\devices

一般一个系列包含多个芯片，每个芯片对应一个xml文件，下面以一个NUVOTON的芯片为例说明。以下红色部分是整理芯片文件的时候需要修改的，修改为对应芯片的信息。模版中也有对应的模版，“XXX”表示需要填写的信息。

这些信息可以从两个地方获取：芯片页面和芯片数据手册。

比如：



芯片信息是一个XML文件，包含以下信息：

芯片名：<Name>NUC123LC2AN1</Name>

芯片厂商：<Vendor>NUVOTON</Vendor>

芯片系列或者家族：<Family>NUC123</Family>

芯片内核：<Arch>Cortex-M0</Arch>

芯片描述：<Description>NUVOTON's NUC123LC2AN1 ….</Description>

芯片参数，参数可以有多条，如下嵌套：

<Parameters>

<Parameter>32-bit timer (TMRB): 4 channels</Parameter>

<Parameter>(I2C): 2 channels</Parameter>

….

</Parameters>

相关的链接，一般包含两个链接，一个是芯片页面，一个是芯片数据手册：

<ReferenceLinks>

<Link name="Chips Page">芯片页面链接</Link>

<Link name="Chips Datasheet">数据手册链接</Link>

</ReferenceLinks>

调试模型的选择，路径是固定的，需要选择一个文件，这个文件在下一个步骤中编写，这里先命名就行了，一般就是系列名：

<DebugSetting>$TOOLKIT\_CONFIG\_DIR$\debugger\NUC123.xml</DebugSetting>

FLASH的地址和大小定义，如果有多个flash就定义多个Block，一般情况就一个：

<Flash>

<Block addr="0x00000000" size="0x00008000" read="true" write="false" execute="true" name="Embedded Flash" />

</Flash>

RAM的地址和大小定义，如果有多个RAM就定义多个Block，一般情况就一个：

<Ram>

<Block addr="0x20000000" size="0x00003000" read="true" write="true" execute="true" name="Embedded RAM" />

</Ram>

下载算法，如果有多个算法就定义多个loader，一般情况就一个，这个文件也是在falsh算法章节生成的，这里先命名就行了命名方式是系列名+FLASH大小（注意一定要和后面的命名一致）：

<FlashLoader>

<loader>$TOOLKIT\_FLASH\_DIR$\NUC123\_32.elf</loader>

</FlashLoader>

## 4.3芯片算法文件和调试模型编写测试

### 4.3.1调试模型

调试 模型的路径是：\$manufacturer$\$series$\config\debugger

这里面有两个模版，一个M0的模版，一个M3的模版。其中M0的模版适用于M0和M0+的芯片；M3的模版适用于M3和M4的芯片，这里需要根据实际的内核来选择模版。

一般一个系列的芯片只需一个调试模型文件。选择好模版之后按照模版的提示修改填写一些信息就可以了。

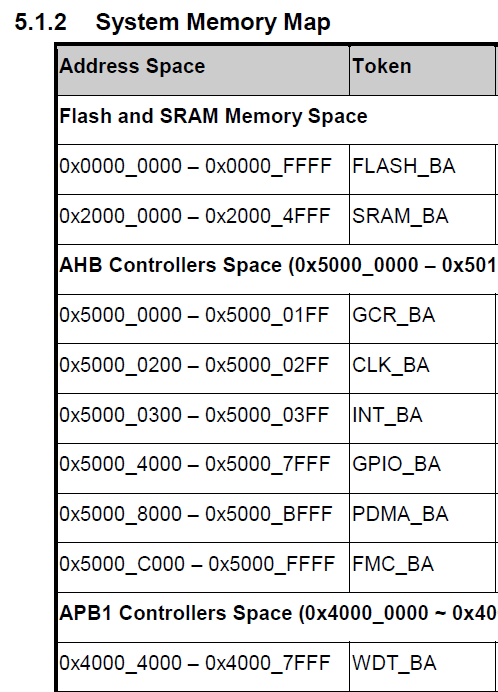
具体内容如下：

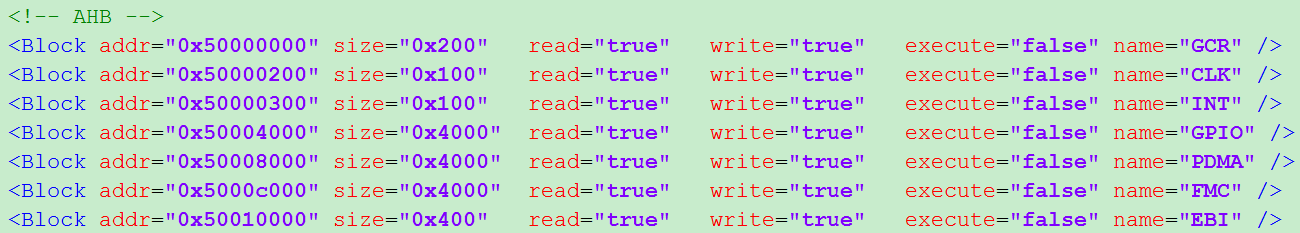
复位时钟大小： <SystemResetClkFreq>12000000</SystemResetClkFreq> 根据实际芯片的复位时钟来填，一般就是内部时钟的值。

Memory Map：Memory Map包括Private peripheral bus 和FiRM外设总线，其中Private peripheral bus中的内容是固定的，只需要填写FiRM外设总线就行了。

FiRM外设总线分为AHB 和 APB ，APB可能又分为 APB1和APB2，具体可以按照参考手册中的memory map 来填写。

如下图所以，一一对应。





### 4.3.2Flash算法

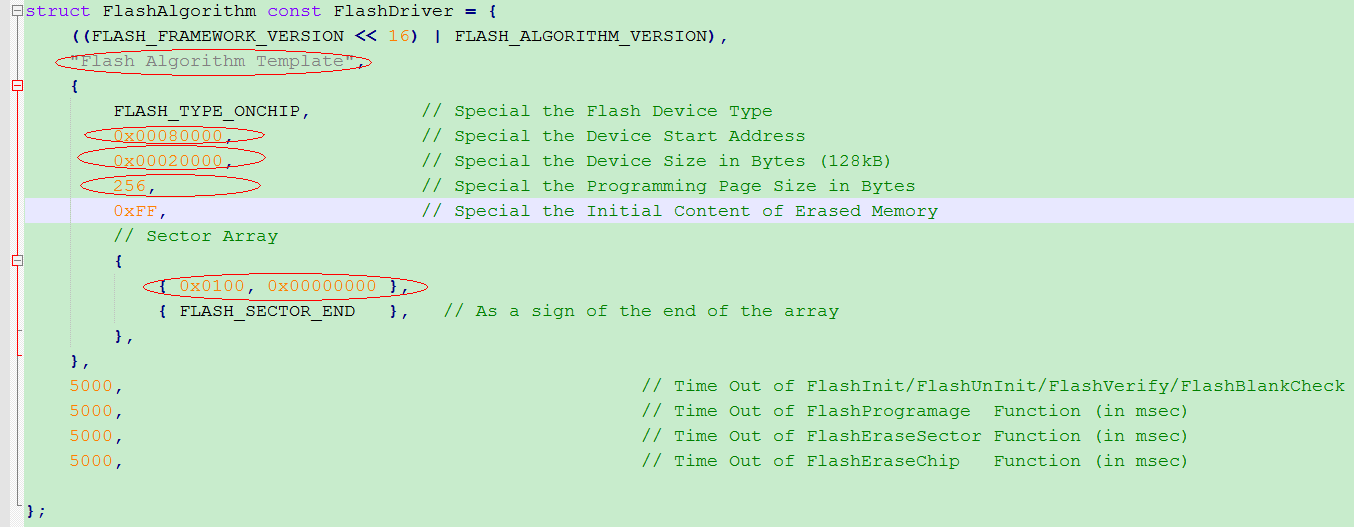
算法的的路径是：\$manufacturer$\$series$\flash

算法可以说是整个厂商库移植中最难的部分了，不像芯片文件、调试模型、甚至后面会提到的启动代码、外设库例子一样照着模型生搬硬套就可以了。这里需要写代码、测试、调试。虽然稍微难一些，但依然有据可循，有模本可依。

关于算法的工作原理、结构，以及API等，你可以参考这里： <http://www.coocox.org/CN/CoFlashGuide/CoFlash_UserGuide_Algorithm.htm> 先学习下。

下面我具体讲步骤：

1. 打开flash下的source会看到两个文件夹：framework和template。其中framework里面的内容不需要修改，我们的工作主要集中在template目录下的两个文件。在开始改写这两个文件之前，先把template重命名为对应的系列$series$。
2. FlashAlgoInfo.c主要定义算法关心的一些内容，比如算法描述，flash的起始地址，flash的大小，page size，sector信息等等，具体内容可以参考结构体的定义。下图是模版的内容，我标注了需要修改的地方：



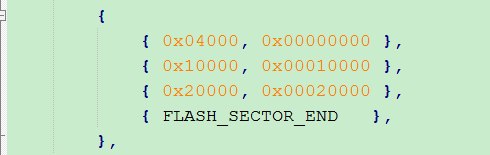
描述：应该包括系列名和flash大小，规范的描述举例“NUC1xx 128KB Flash”

Flash起始地址：从参考手册或者数据手册中获取，一般是0x00000000

Flash 大小：从参考手册或者数据手册中获取，没有参考值，一般一个系列的芯片会对应多种不同大小的flash。下面会提到这种情况如何处理。

Page Size：flash的Page大小从参考手册或者数据手册中获取，常见的是128，256，512，一般一个系列的芯片page size都是一样的，不过仍有特例，需注意。

扇区信息：包含两个信息，第一个是扇区大小，第二个是扇区的起始地址。从参考手册或者数据手册中获取具体信息。如果flash设备中的所有扇区的大小都是一样的，那么您只需要指定一次扇区的大小和起始地址，接着设置FLASH\_SECTOR\_END就行，参见模板里的描述。否则，您需要一一列举出所有的扇区的大小和起始地址。如下图所示：



上面提到，Flash 大小可能存在多个，这种情况下就需要用If defined endif语句来区分。

具体就是一个“Flash 大小”对应一个If defined endif，XXX一般用系列名+法拉盛大小组合，比如NUC1xx\_128 NUC1xx\_64等。

#if defined ( XXX )

#if defined ( \_\_GNUC\_\_ )

\_\_attribute\_\_ ((section(".driver\_info")))

#endif

struct FlashAlgorithm const FlashDriver = {

……

}

#endif

这时，一一对应就可以了，NUC1xx\_128对应的flash大小为0x00020000，NUC1xx\_64对应的flash大小为0x00010000。

到了这里，FlashAlgoInfo.c文件就算完成了。

1. FlashAlgoFncs.c 实现CooCox定义的算法API函数，API列表如下：

// Initialize Flash Before Various Flash Operation

extern int FlashInit (unsigned long baseAddr,

unsigned long clk,

unsigned long operateFuc);

// UnInit Flash After Various Flash Operation

extern int FlashUnInit (unsigned long operateFuc);

extern int FlashEraseChip (void); // Erase Complete Device

extern int FlashEraseSector (unsigned long sectorAddr); // Erase Sector Function

extern int FlashProgramPage (unsigned long pageAddr, // Program Page Function

unsigned long size,

unsigned char \*buf);

extern int FlashVerify (unsigned long verifyAddr, // Page Verify Function

unsigned long size,

unsigned char \*buf);

// Page Blank Check Function

extern int FlashBlankCheck (unsigned long checkAddr,

unsigned long size);

**FlashInit** 虽然定义的参数比较多，看起来繁杂，实际中，在FlashInit里面只需完成几点即可，如果flash有锁或者保护措施，解锁或者解除保护；清除初始的标志位；关闭看门狗。

**FlashUnInit** 一般不需要很特殊的操作，直接返回即可。

**FlashEraseChip** 擦除整个芯片

**FlashEraseSector** 擦除一个扇区

**FlashProgramPage** 对一个page编程

**FlashVerify** 检验，可选的功能。当没有实现该功能时，colink将会直接读取flash存储器里的数据进行校验。

**FlashBlankCheck** 空检查，可选的功能。当没有实现该功能时，colink将会直接读取flash存储器里的数据进行空白检测。

这个文件说起来内容就这些，实际写的时候还有很多需要注意的地方：

1. 需要定义一些外设的地址和寄存器含义，比如flash外设的，WDT外设的，甚至clock相关外设的地址和寄存器含义。还有flash相关的命令
2. 定义数据类型和寄存器访问的宏，如下所示：

typedef volatile unsigned long vu32;

typedef unsigned long u32;

#define M32(adr) (\*((vu32 \*) (adr)))

1. 可能需要定义扇区的数目

#if defined NUC1xx\_128

#define SECTOR\_NUM 256

#endif

1. 还可能需要定义page大小，比如：

#if defined ( TMPM37x\_512 )

#define PAGE\_SIZE 512 // Page Size

#else

#define PAGE\_SIZE 256 // Page Size

#endif

这里根据经验列举了一些，实际中可能更多，由于不同芯片，功能上有很多的差异，这里不可能一一列举。

不过，实际操作过程中也是有捷径可以走的，比如参考相近系列的实现；比如参考MDK下算法的实现。

如果MDK下有相应算法，参考MDK下的，尤其有效果，几乎只需要移植就行了。

1. 代码已经写完，这一步就是编译生成代码了，gccProject目录下有一个makefile文件，运行这个makefile就可以生成算法了，不同的flash大小对应的算法是不同的（也就是一个flash大小对应一个算法），那么如何通过makefile来生成不同的算法呢？在运行前点击编辑makefile，会看到makefile中有个FLASH\_CHIP选项。Makefile就是通过这个选项来生成不同算法的，那个这个FLASH\_CHIP等于什么呢？答案是等于前面FlashAlgoInfo.c中Ifdefined endif结构中定义的那些值，比如NUC1xx\_128 NUC1xx\_64等。当指定FLASH\_CHIP= NUC1xx\_128就会生成 NUC1xx 128K算法。依次类推。

如何运行makefile？

启动CMD，并cd到makefile对应的目录，然后运行 make（前提是你完成了 资料和环境准备中的 第七步和第八步）

特别注意：

1. 在运行make前还需要在当前目录新建两个文件夹，一个名为bin，一个名为obj
2. 在make完一个算法之后，修改FLASH\_CHIP的值再次make之前，要先执行 make clean。为了保险起见，每次make之前都先执行 make clean。

顺利的话，在make完成之后，返回flash目录，你会看到对应的算法已经生成。

1. 算法测试，分两种方法：黑盒和白盒测试
2. 黑盒测试就是用你生成的算法来下载程序，首先看能不能下载，其次看下载之后的功能是否正确（比如程序功能是闪灯，看实际开发板有没有闪灯）。
3. 白盒测试就是，新建CoIDE一个工程，把FlashAlgoInfo.c FlashAlgoFncs.c FlashAlgorithm.h 以及对应的启动代码加到这个工程。然后再在main.c中编写测试代码。RAM调试，看相应的代码运行之后是否正确？比如擦除一个扇区之后从Memory窗口看，这个扇区是否是全0xFF？对一个page编程之后，查看这个page的值是否和编写的符合？

## 4.4组件整理

组件整理包括几点：启动代码编写（一般的cmsis库都没有GCC的启动代码），组件拆分、去耦（去耦的意思就是避免一个组件必须依赖其他组件才能编译，除非它们真的存在依赖关系），组件文档编写。

### 4.4.1启动代码

启动代码的目录是 cmsis\_lib\lib\cmsis\_boot\startup\coide

在这个目录下给出了一个模版启动代码，这个启动代码可以使用，但是功能不全，确少外部中断的向量表，所以，移植的工作很简单，就是补齐这些向量表就行了。补一个向量需要修改三处：

一是需要一个弱声明 如 void WEAK SysTick\_Handler(void);

其次就是在 g\_pfnVectors数组中添加对应向量，这里要特别注意向量的顺序不能错。

最后就是给这个向量一个默认地址，#pragma weak SysTick\_Handler = Default\_Handler。

启动代码虽然要编写代码，但过程不难，最主要就是要细致。

### 4.4.2组件拆分

组件一共分为三类：

Cmsis core 组件一般固定不变，在cmsis\_lib\lib\cmsis\_core\下面

Cmsis boot 在 cmsis\_lib\lib\cmsis\_boot\下面，一般会依赖Cmsis core

Peripheral组件一般可以按照以下方式分类：

SYSCLK—系统相关的，时钟相关统一划到这个组件下面，这个组件会依赖Cmsis boot

GPIO 所有 IO相关的组件，包括GPIO功能和AFIO功能以及EXTI功能，这个组件依赖SYSCLK

UART

SPI

I2C

ADC

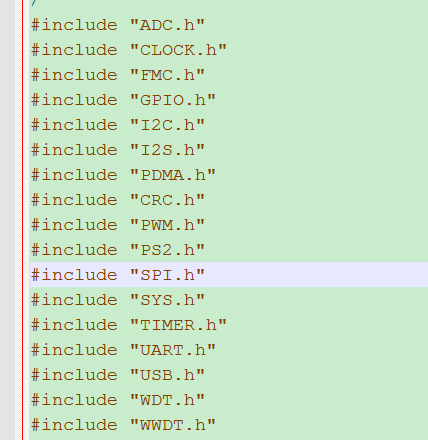
PWM

TIMER

…..

这些组件都会依赖GPIO，但一般这些组件之间不会相互依赖。依赖都是通过头文件的包含产生的，然而下列几种情况会增加这些组件的相互依赖：

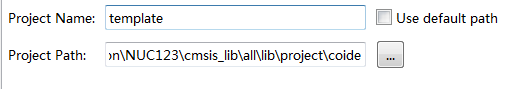
1. 很多cmsis库都定义了一个名为config或者lib之类的头文件，然后在这个头文件中包含了所有的外设头文件，下图所示：



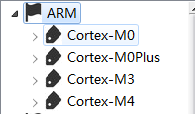
这种情况下，一般每个C文件都会包含这个头文件，这样就变相的增加了依赖关系，我们要做的就是把这些依赖关系去掉，去掉的方法就是用必须要的头文件代替这个带有冗余信息的头文件，比如一般SPI.c只需包含GPIO.h SYS.h和 寄存器定义头文件即可。

1. CMSIS boot组件下的那个定义寄存器的头文件，里面包含了所有的头文件。
2. 组件之间的依赖，比如，USB的原文件中包含了PWM的头文件（这里是随便举例，外设组件有时确实会存在相互依赖，主要靠编译是否正确来判断）。

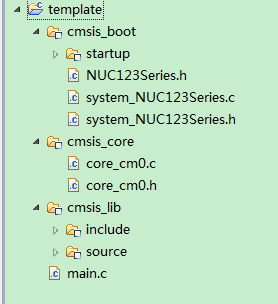
首先，利用coide新建一个工程，命名为template，去掉use default path勾选框，并选择cmsis\_lib\all\lib\project\coide目录，如图：



下一步（如果出现A project name already exist in this directory，请先删除模版中的template.coproj），新建工程时选择 ARM厂商，根据芯片的内核选选择对应的芯片M0 M0+ M3 M4 ，如图：



新建完成之后，在工程视图添加分组和添加文件，最后得到如下工程视图：



然后编译，一般情况下，编译能够正常通过。

接下来就是要把那些增加了依赖关系的文件（config或者lib之类的头文件）去掉，当然，去掉这些头文件，相当于去掉了所有的依赖关系（必须的和非必须的），所以，肯定会出现编译错误的（大部分应该是函数 变量未定义之类的）， 接下来要做的就是修复这些错误，过程很简单，就是把必须的依赖关系加上，即在相应的源文件中添加对于的头文件。这里主要就是微调一些代码，不作详细介绍。

### 4.4.3 组件文档

组件文档的目录是 cmsis\_lib\$ Peripheral$\ document下，其中$ Peripheral$就是上节中拆分的组件名，比如 GPIO ADC PWM 等。每个组件包括两个文档：组件帮助文档，简单介绍这个组件是干什么的，有什么功能。一个是组件上传文档，这个文档记录了上节中组件拆分的细节，包含该组件有哪些文件，依赖哪些其他组件等。

帮助文档，很简单，一般copy参考手册中对应章节的overview就行了（注意：去掉PDF里面可能存在的特殊字符，比如上标®或者°等等）。

组件上传文档，包括 组件名称，组件简短描述，版本，标签，文件路径，coide工程视图中的位置，依赖的组件，和支持和的芯片。

Name（组件名称）一般是 GPIO ADC PWM等外设简写

Description（组件简短描述） 一般是组件名称的全写

Version（版本），组件的版本号

Tags（标签），为对应的组件加标签，关键字，搜索用。

Files（文件路径），组件文件的物理路径

CoIDEFiles（）文件在coide视图中的位置。

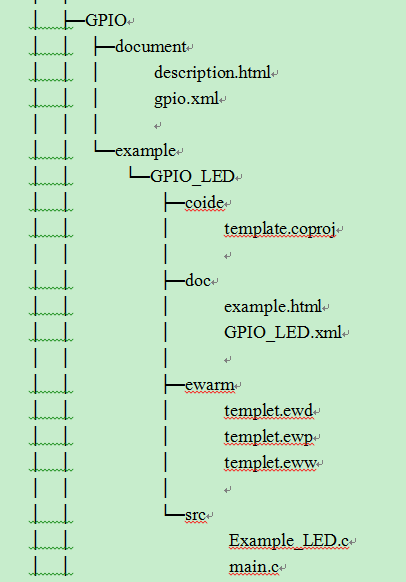
Dependencys（依赖关系），这个组件所依赖的其他组件，这个由这个组件所包含的头文件决定。

SupportDevices（支持的芯片），一般是这个系列的所有芯片，需要注意的是，有些外设组件不一定支持这个系列的所有芯片。以芯片手册为准。

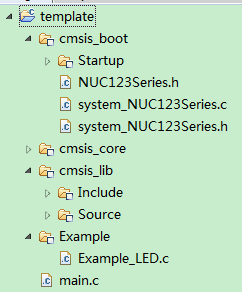
## 4.5例子整理

一个组件至少整理一个典型的例子，例子包括几点：新建例子工程，按照coide例子的规则修改例子代码，编写例子文档。

新建例子工程之前，先前文的目录组织，组织好目录，以GPIO为例，如下图所示：



每个例子至少需要一个coide工程，IAR工程可选，可以配合测试例子的功能用。目录准备好了，就要向目录中填充文件，例子的源文件和头文件都放到src下，一个例子可以包含一个或者若干个文件。这些例子文件从哪儿来？当然还是厂商提供的CMSIS包中，复制过来之后。就可以新建工程了，步骤详见组件拆分 章节，注意下不同的地方就行了：1.工程文件路径保存在GPIO /example/GPIO\_LED/coide下面。2. 新增一个example分组如图：



接下来就是改写代码了，工作量很小，不过要特别注意几点：

1. 一直C库相关的代码，主要包括主要集中在printf的移植，getchar的移植和重定向等。涉及到重定向的才需要修改，具体过程详见：[在CoIDE里使用Retarget printf来打印调试信息](http://note.youdao.com/share/?id=170b576fcd491c9f2b56bc3cdcce2313&type=note)。

将1移植好了之后，就要注意CoIDE下例子的一些特殊表现形式了，CoIDE可以自动添加例子到工程和main.c函数里面，所以CoIDE的例子需要注意一下几点：

a. 例子支持多个文件，但必须有一个C文件作为这个例子的主文件；例子能自动添加到main.c，所以例子的主体函数，要放在例子主文件的第一个位置，其他函数的声明可以放在前面。举个例子，一个例子文件里面包含了三个函数，在文件最前面 声明这三个函数（顺序没有关系），然后，将主体函数放在三个函数的最前面，主体函数的意思就是，这个函数会被CoIDE自动添加到main.c中。

**void** **functionA**(**void**);

**void** **functionB**(**void**);

**void** **functionC**(**void**);

**void** **functionC**(**void**)

{

functionB();

functionA();

....

}

**void** **functionA**(**void**)

{

....

}

**void** **functionB**(**void**)

{

....

}

那么这个例子文件中的functionC会在coide添加例子时自动添加到main.c的main函数中，如下：

**extern** **void** functionC (**void**);

**int** **main**(**void**)

{

functionC ();

**while**(1);

**return** (0);

}

b. 例子函数命中不要有特殊符号

2. 由于厂商提供的CMSIS中的例子都是包含main函数的，所以移植的时候要去掉这个main函数，一般重命名为例子主体函数名（比如GPIO\_LED\_Example）并将这个函数移到当前文件的最前面（意思就是这个函数是当前文件的第一个函数）。这样做可能导致一些当前函数（在主体函数里面调用的）没有声明的错误。所以还需要把当前文件中的其他函数在文件开头声明。

举例说明,下图就是修改前后的结构对比。



1. 修改main.c

**extern** **void** GPIO\_LED\_Example (**void**);

**int** **main**(**void**)

{

GPIO\_LED\_Example ();

**while**(1);

**return** (0);

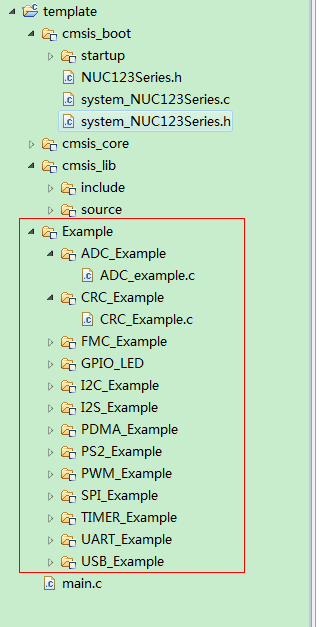
}

1. 编译这个例子，保证编译没有错误。

以上是一个例子的移植，其他例子的移植类似，而且可以省去新建工程的步骤，直接复制第一个例子工程，删除无效文件并重新添加例子文件就可以了。

所有外设的例子整理完成之后，还需要额外建立一个工程，路径位于cmsis\_lib\all\example\project\coide下，新建过程不重复（也可以直接将cmsis\_lib\all\lib\project\coide下的工程复制过来），然后增加example分组，并在这个分组下为每个例子新建一个分组，每个分组下添加对应的例子文件。

如图所示：



## 4.6例子测试

## 4.7上传