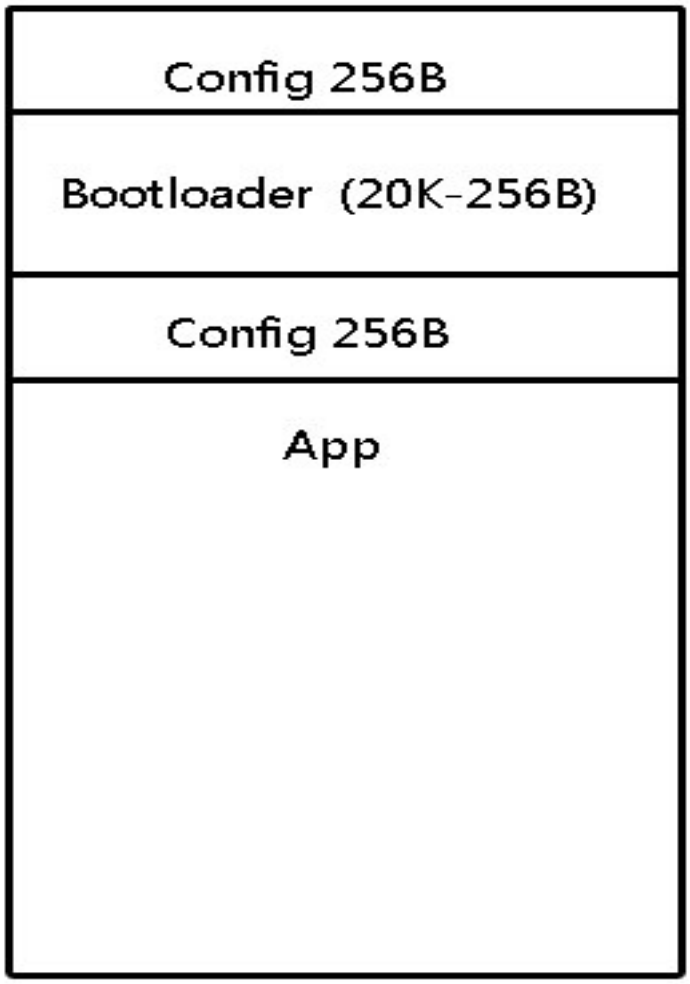
USBS 固件升级

2015.4.9

1. 分区

个人对分区的理解是“划分区域”，也就是定义一块区域用来做什么。
USBS 中的分区如下(最新版)：



其中 256B 的配置数据是通过 Qt 程序写进去的，目的是通过 Qt 软件避免为每一块板子单

独生成一个烧录文件，便于软件管理，同时也便于发布到网上供网友测试。App 的 256B 数据是我在升级 USBS 的时候才需要用到，普通的 App 开发不需要管配置数据。

2. 挂载

在 ST 的库中有一个例程：Mass_Storage，是挂载一个磁盘设备，不过这个例程是将 SD 卡跟 NAND 挂载到电脑上，而我们现在的需求是将片内 Flash 挂载到电脑上作为一个存储设备，所以需要修改对存储器的操作。我们重点需要修改的文件为：**mass_mal.c**。该源文件中主要有四个函数：**MAL_Init**、**MAL_Write**、**MAL_Read**、**MAL_GetStatus**。其中 **MAL_Init** 跟 **MAL_GetStatus** 函数改起来相对简单：

```
uint16_t MAL_Init(uint8_t lun)
{
  uint16_t status = MAL_OK;

  switch (lun)
  {
    case 0:
      SRAM_Init();
      break;
    default:
      return MAL_FAIL;
  }
  return status;
}
```

```
uint16_t MAL_GetStatus (uint8_t lun)
{
    switch(lun)
    {
    case 0:
        return MAL_OK;
    default:
        return MAL_FAIL;
    }
}
```

Write 跟 **Read** 改起来也简单，主要是需要更多的调试。如：

```
    NAND_Write(Memory_Offset, Writebuff, Transfer_Length);
```

改为：

```
STMFLASH_Write(FLASH_SAVE_ADDR+Memory_Offset,(u16*)Writebuff,Transfer_Length/2);
```

```
    NAND_Read(Memory_Offset, Readbuff, Transfer_Length);
```

改为：

```
STMFLASH_Read(FLASH_SAVE_ADDR+Memory_Offset,(u16*)Readbuff,Transfer_Length/2);
```

这样就可以将存储设备改为我们的片内 Flash。

将片内 Flash 作为一个存储设备挂载到电脑上之后你就把它当作一个小容量的 U 盘，我们可以往里边拷贝任意格式的文件，那么我们也可以把程序直接拷贝进去。程序拷贝到了 Flash 中也就完成了程序的下载过程，因为我们下载程序也无非就是将程序通过如 SWD，串口把编译好的程序放到 Flash 当中。而剩下的工作就是怎么让你的程序给运行起来。

3. 引导

在 ST 库中有一个现成的示例程序告诉我们怎么去引导 App，即：Device_Firmware_Upgrade，也就是我们通常说是的 DFU 升级。但是如果你在产品中使用 DFU 方式升级会有个问题：假如你决定使用 ST 的上位机目前你只有在 Windows 系统中有可视化界面，Linux 中是命令行，其他系统暂时没有了解到；假如你决定自己开发上位机那么你就得花时间跟精力去解决各个系统的驱动跟上位机的问题。所以我们可以很容易想到 USBS 可以很好地解决系统兼容性问题。

在原版的 DFU 中引导是像下面这样去实现的：

```
/* Check if the Key push-button on STM3210x-EVAL Board is pressed */
if (DFU_Button_Read() != 0x00)
{ /* Test if user code is programmed starting from address 0x8003000 */
   if (((*(__IO uint32_t*)ApplicationAddress) & 0x2FFE0000 ) == 0x20000000)
   { /* Jump to user application */

      JumpAddress = *(__IO uint32_t*) (ApplicationAddress + 4);
      Jump_To_Application = (pFunction) JumpAddress;
      /* Initialize user application's Stack Pointer */
      __set_MSP(*(__IO uint32_t*) ApplicationAddress);
      Jump_To_Application();
   }
} /* Otherwise enters DFU mode to allow user to program his application */
```

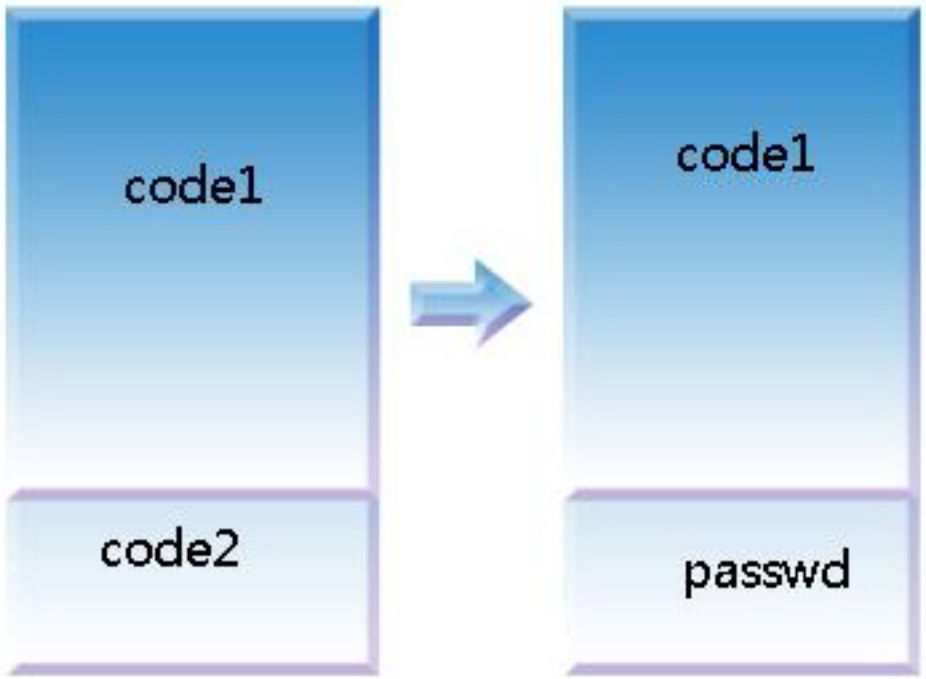
在 USBS 中引导过程是一样的，只不过 App 地址不再是 0x8003000，而是 0x800A000、0x8005000 这样一些地址。

当我们实现了程序的下载跟 App 引导，实际上这个时候我们已经实现了一个 bootloader，即已经具备了 USBS 的基本功能。但是这个时候你把 USBS 从烧好的板子上读取出来，然后烧写到另外一块板子上程序是可以正常运行的，也就是不具备安全性！因此我们要在 USBS 中加入另外一个功能，即加密。

4. 加密

目前的加密算法是需要 MCU 拥有唯一的 ID 号的，也就是说对于没有 ID 的 MCU 加密将失效。

加密算法是之前研究出来的，通过将加密代码擦除的方式确保加密信息只生成一次，说白了就是破坏程序的完整性。这样带来的一个意外的好处就是即使你去反汇编你也不可能得到生成加密信息的那段代码。加密过程中 Flash 内容变化如下：



而最新版 USBS 中在 256B 的配置数据中专门为 App 分配了 32B 的空间用于存放加密数据。所以最新版的 USBS 中使用了两套加密算法，一套是给自己用的，并且不公开，确保 USBS 被读出来之后在另外一块 IC 上不能正常运行；另外一套加密算法结合 ID 跟 Licence 生成串加密信息由 App 进行校验，算法公开，主要通过 Licence 确保安全。另外在 App 中还对 USBS 进行 CRC32 匹配，通过双校验确保安全。

加密的原始设计在《Flash 加密技术》一文中有详细描述，当时的加密算法相对简单，加密信息也比较少，只有 12B：

```
_mtext->id[0] = uid[1]^uid[2]^UID0;
_mtext->id[1] = uid[1]^UID0^UID1;
_mtext->id[2] = uid[0]^uid[2]^UID2;
```

到目前的最新版，加密信息扩展到了 32B，其设计原理跟《Flash 加密技术》中所描述的是一致的，依然是擦除加密代码。

5. 配置

为了解决版本控制问题，并且便于发布到网上供网友测试，在 USBS 中做了一个抽象：

```
typedef struct { //  输出设备
    //void (*init)(QiFreeIO *io); //  初始化
    void (*led_init)(QiFreeIO *io); // led 初始化
    void (*key_init)(QiFreeIO *io); // key 初始化
    void (*led_on)(QiFreeIO *io, int on);     //  点灯,输出
    void (*reversal)(QiFreeIO *io); //  电平翻转,输出
    uint8_t (*key_down)(QiFreeIO *io); //  读取电平,输入
}QiFree_ioObj;
```

QiFree_ioObj 的存在使得我可以通过一串配置信息(即 256B)改变检测及 LED 指示所使用的 IO 端口，从而在调试程序的时候再不必同一个版本为每一个板子专门管理，硬件换了只需用 Qt 程序配置一下 IO 即可以使用。

小结

总的来说，我们只需解决**分区、挂载、引导**，一个 U 盘方式的升级就一句做好了。而**加密**是为了解决安全问题才引入的，**配置**则是为了解决额外的版本控制而引入的。