一种从用户代码调用系统存储器中 Bootloader 的方法

前言

大家都知道，任何 STM32 都包含有一块系统存储器（System Memory），里边存储着内部的启动代码 Bootloader。不同的 STM32 型号所支持的用于升级代码的通讯口不尽相同，需要参考应用笔记 AN2606。但是，有一个问题避免不了，那就是如何进入 System Memory 去执行 Bootloader？通常的办法都是将 BOOT1 和 BOOT0 进行配置：BOOT0 拉高，BOOT1 拉低（有些型号的 BOOT1 由选项字节 nBOOT1 进行控制）。可是在一些产品中，由于外观的要求，往往不方便在外边开口去放置按键或跳线来改变 BOOT 脚的电平。而且，用户并不想自己写 IAP 代码，觉得麻烦。特别是一些产品，需要使用 USB DFU 来进行代码升级的，而在产品功能中 USB 又没用到，用户就会觉得自己为了一个通过 USB 进行代码升级的功能，去写 IAP 的话，需要去熟悉 USB 的代码，觉得麻烦，而且这些 USB 的代码还占用了用户的程序空间。对于这些用户来讲，他们很希望能在不管 BOOT 脚的情况下能够去调用 STM32 中 System Memory 的 Bootloader，完成代码升级功能。

问题

某客户在其产品的设计中，使用了 STM32F411。由于产品外观的要求，无法在外部对 BOOT 脚进行控制，而且外观上只有 USB 接口是留在外边的，需要使用 USB DFU 进行升级。而且 USB 接口只用于代码升级，没有其他功能，所以客户不想去碰 USB 代码，希望能够直接使用 System Memory 中的 Bootloader 进行代码升级。

调研

1. 判断其可行性

首先，打开应用笔记 AN2606《STM32 microcontroller system memory boot mode》，翻到 3.1 Bootloader activation 一节的最后，可以看到如下信息：

> In addition to patterns described above, user can execute bootloader by performing a jump to system memory from user code. Before jumping to Bootloader user must:
> - Disable all peripheral clocks
> - Disable used PLL
> - Disable interrupts
> - Clear pending interrupts
>
> System memory boot mode can be exited by getting out from bootloader activation condition and generating hardware reset or using Go command to execute user code.

这里的意思就是说，用户可以通过从用户代码跳转到系统存储器去执行 Bootloader。但是，在跳转到 Bootloader 之前，有几个事情必须要做好：

1) 关闭所有外设的时钟
2) 关闭使用的 PLL

3) 禁用所有中断

4) 清除所有挂起的中断标志位

最后，可以通过离开 Bootloader 激活条件且产生一个硬件复位或者直接使用 Go 命令去执行用户代码。

那么，如何从用户代码跳转到 System Memory 中去呢？这个其实并不难，如果写过 IAP，或者看过关于 IAP 的应用笔记中的参考代码的话，比如应用笔记 AN3965“STM32F40x/STM32F41x in-application programming using the USART”及其参考代码 STSW-STM32067，都应该知道，IAP 的启动代码通过重新设置主堆栈指针并跳转到用户代码来执行用户代码的。同样的道理，只要知道 System Memory 的地址，一样可以从用户代码通过重新设置主堆栈指针并跳转到 System Memory 来执行 Bootloader。而 System Memory 地址可以从参考手册来获得。比如，查看 STM32F411 的参考手册 RM0383，可以找到如下的表格：

Table 4. Flash module organization (STM32F411xC/E)

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

可以知道 STM32F411 的 System memory 地址从 0x1FFF0000 开始。

那很多人又会问了，我的代码很复杂，用了很多外设，开了很多中断，可是要跳转到 System Memory 中的 Bootloader，需要关所有外设的时钟，需要关 PLL，需要关闭所有中断，需要禁用所有的中断，清除所有挂起的中断。这可是一项非常庞大的的任务啊！所以，在这里，我们需要一个更简单的事情来完成这项庞大的任务。其实真的就有这么简单的一个方法——复位！通过软件复位来实现这一目的。但是，复位后，又怎么知道还记得我们要去做代码升级呢？这又要用到 STM32 另一个特性了，那就是后备数据寄存器 Backup Data Registers 在软件复位后会保留其值，这样给了我们在复位前后做一个标志的机会。

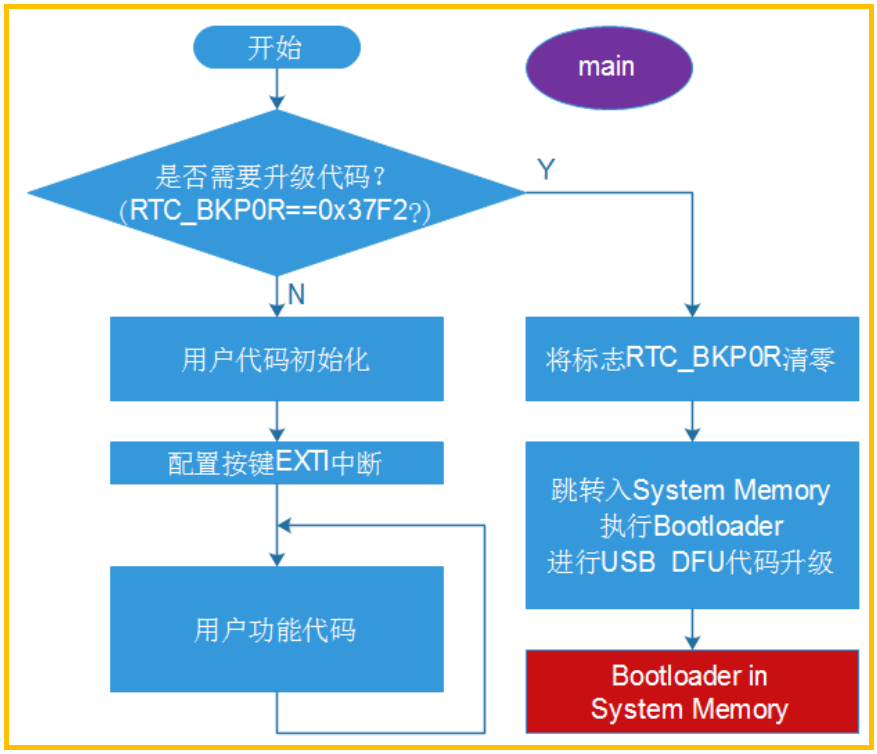
这样，考证下来，客户的需求是具备可行性的。接下来需要做的是理清思路。

2. 软件流程

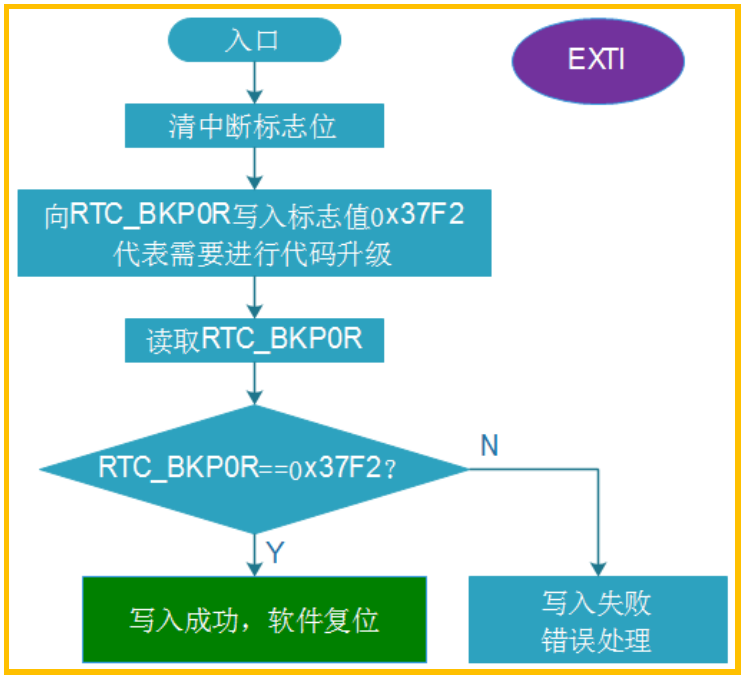
这里使用 32F411EDISCOVERY 板来设计一个参考例程：设计一个用户程序，让 LED3 进行闪烁；当用户按键被按下，产生 EXTI 中断，在中断中选择后备数据寄存器 RTC_BKP0R，写入值 0x32F2，然后产生软件复位；软件复位后，在运行代码的

最前面对 RTC_BKP0R 进行判断，如果其值不是 0x32F2 则直接去运行用户代码，如果其值为 0x32F2 则是需要跳转到 Bootloader 去进行代码升级，并在跳转前将 RTC_BKP0R 清零。这样，在进入 Bootloader 后，客户进行 USB DFU 升级后，将来不会因为非需要升级代码的复位而误入 Bootloader。

来看软件流程图，先来看主程序的流程图：



再来看 EXTI 中断的流程图：



3. 主要代码

使用 STM32F4Cube 库来开发这个例程。先来看位于 main.c 中的 main 函数：

```
int main(void)
{
  /* STM32F4xx HAL library initialization:
       - Configure the Flash prefetch, Flash preread and Buffer caches
       - Systick timer is configured by default as source of time base, but user
             can eventually implement his proper time base source (a general purpose
             timer for example or other time source), keeping in mind that Time base
             duration should be kept 1ms since PPP_TIMEOUT_VALUEs are defined and
             handled in milliseconds basis.
       - Low Level Initialization
     */
  HAL_Init();

  /* Configure the System clock to have a frequency of 100 MHz */
  SystemClock_Config();

  /* Configure LED3, LED4, LED5 and LED6 */
  BSP_LED_Init(LED3);
  BSP_LED_Init(LED4);
  BSP_LED_Init(LED5);
  BSP_LED_Init(LED6);

  /* Configure EXTI Line0 (connected to PA0 pin) in interrupt mode */
  EXTILine0_Config();

  /* Infinite loop */
  while (1)
  {
  }
}
```

Main 函数很简单，配置系统时钟，对使用的 LED 进行初始化，然后配置了用户按键的 EXTI 中断，然后就进入主循环了。前面说到，要实现用户的功能程序为 LED3 闪烁，在主循环我们没看到，是因为在 Cube 库中，会使用 SysTick，所以把 LED3 的闪烁放到 SysTick 的中断代码中了，查看 stm32f4xx_it.c，如下：

```
void SysTick_Handler(void)
{
  HAL_IncTick();

  // LED3 Toggle
  led_toggle_counter++;
  if (led_toggle_counter >= 500)
  {
    led_toggle_counter = 0;
    BSP_LED_Toggle(LED3);
  }
}
```

从 main 函数最开始的那段注释中知道，跳入 main 函数前，在 startup_stm32f411xe.s 中早已经先调用执行了位于 system_stm32f4xx.c 中的 SystemInit 函数。SystemInit 函数主要执行初始化 FPU、复位 RCC 时钟寄存器、配置向量表等功能。由于我们希望在最原始的状态下进入 System Memory，所以我们将跳转到 System Memory 放在这个函数的最前头，如下：

```
void SystemInit(void)
{
  /* Check if need to go into bootloader before configure clock*/
  RtcHandle.Instance = RTC;
  if(HAL_RTCEx_BKUPRead(&RtcHandle, RTC_BKP_DR0) == 0x32F2)
  {
    __HAL_RCC_PWR_CLK_ENABLE();
    HAL_PWR_EnableBkUpAccess();
    __HAL_RCC_RTC_CONFIG(RCC_RTCCLKSOURCE_HSE_DIV2);
    __HAL_RCC_RTC_ENABLE();
    HAL_RTCEx_BKUPWrite(&RtcHandle, RTC_BKP_DR0, 0x0);
    __HAL_RCC_RTC_DISABLE();
    HAL_PWR_DisableBkUpAccess();
    __HAL_RCC_PWR_CLK_DISABLE();

    __set_MSP(*(__IO uint32_t*) 0x1FFF0000);
    SysMemBootJump = (void (*)(void)) (*((uint32_t *) 0x1FFF0004));
    SysMemBootJump();
    while (1);
  }

  /* FPU settings ------------------------------------------------------------*/
  #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
    SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));  /* set CP10 and CP11 Full Access */
  #endif
  /* Reset the RCC clock configuration to the default reset state ------------*/
  /* Set HSION bit */
  RCC->CR |= (uint32_t)0x00000001;

  /* Reset CFGR register */
  RCC->CFGR = 0x00000000;

  /* Reset HSEON, CSSON and PLLON bits */
  RCC->CR &= (uint32_t)0xFEF6FFFF;

  /* Reset PLLCFGR register */
  RCC->PLLCFGR = 0x24003010;

  /* Reset HSEBYP bit */
  RCC->CR &= (uint32_t)0xFFFBFFFF;

  /* Disable all interrupts */
  RCC->CIR = 0x00000000;
```

```
  /* Configure the Vector Table location add offset address ------------------*/
#ifdef VECT_TAB_SRAM
  SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal SRAM */
#else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal FLASH
*/
#endif
}
```

可以看到，在函数的最前面对 RTC_BKP_DR0 进行了判断，如果其值为 0x32F2 的话，则先启动备份域的访问时序，如 RM0383 中 5.1.2 Battery backup domain 所描述的：

Backup domain access

After reset, the backup domain (RTC registers and RTC backup register) is protected against possible unwanted write accesses. To enable access to the backup domain, proceed as follows:

- Access to the RTC and RTC backup registers

1. Enable the power interface clock by setting the PWREN bits in the RCC_APB1ENR register (see *Section 6.3.11: RCC APB1 peripheral clock enable register (RCC_APB1ENR)*)
2. Set the DBP bit in the *Section 5.4.1* to enable access to the backup domain
3. Select the RTC clock source: see *Section 6.2.8: RTC/AWU clock*
4. Enable the RTC clock by programming the RTCEN [15] bit in the *Section 6.3.17: RCC Backup domain control register (RCC_BDCR)*

然后将 RTC_BKP_DR0 清零，再关闭执行这次操作所打开的时钟。

主堆栈指针 MSP 的初始值位于向量表偏移量为 0x00 的位置，复位 Reset 的值则位于向量表偏移量为 0x04 的位置。对于 STM32F411 来说，当执行 System Memeory 中的 Bootloader 时，MSP 的初始值位于 0x1FFF0000，而 Reset 则位于 0x1FFF0004。所以在程序中，使用__set_MSP(*(__IO uint32_t*) 0x1FFF0000);来重新设置主堆栈指针，而后再跳转到 0x1FFF0004 去执行 Bootloader。

再来看位于 stm32f4xx_it.c 中的 EXTI 中断程序：

```
void EXTI0_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(KEY_BUTTON_PIN);
}
```

及其位于 main.c 中的 Callback 函数：

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if(GPIO_Pin == KEY_BUTTON_PIN)
  {
    __HAL_RCC_PWR_CLK_ENABLE();
    HAL_PWR_EnableBkUpAccess();
    __HAL_RCC_RTC_CONFIG(RCC_RTCCLKSOURCE_HSE_DIV2);
    __HAL_RCC_RTC_ENABLE();
    RtcHandle.Instance = RTC;
```

```
    HAL_RTCEx_BKUPWrite(&RtcHandle, RTC_BKP_DR0, 0x32F2);
    if(HAL_RTCEx_BKUPRead(&RtcHandle, RTC_BKP_DR0) != 0x32F2)
    {
      // Write backup data memory failed
      BSP_LED_On(LED5);
      while (1) ;               // Error
    }
    else
    {
      // Write backup data memory succeeded.
      BSP_LED_On(LED6);
      HAL_NVIC_SystemReset();   // Software reset for going into bootloader
      while (1)   ;
    }
  }
}
```

当判断到用户按键按下，需要进行用户代码升级时，先启动备份域的访问时序，将 RTC_BKP_DR0 的值写为 0x32F2。再读回来判断是否写入成功，以方便调试。如果写入成功后，则就调用 HAL_NVIC_SystemReset()进行软件复位。重新复位后，就可以进入 System Memory 了。

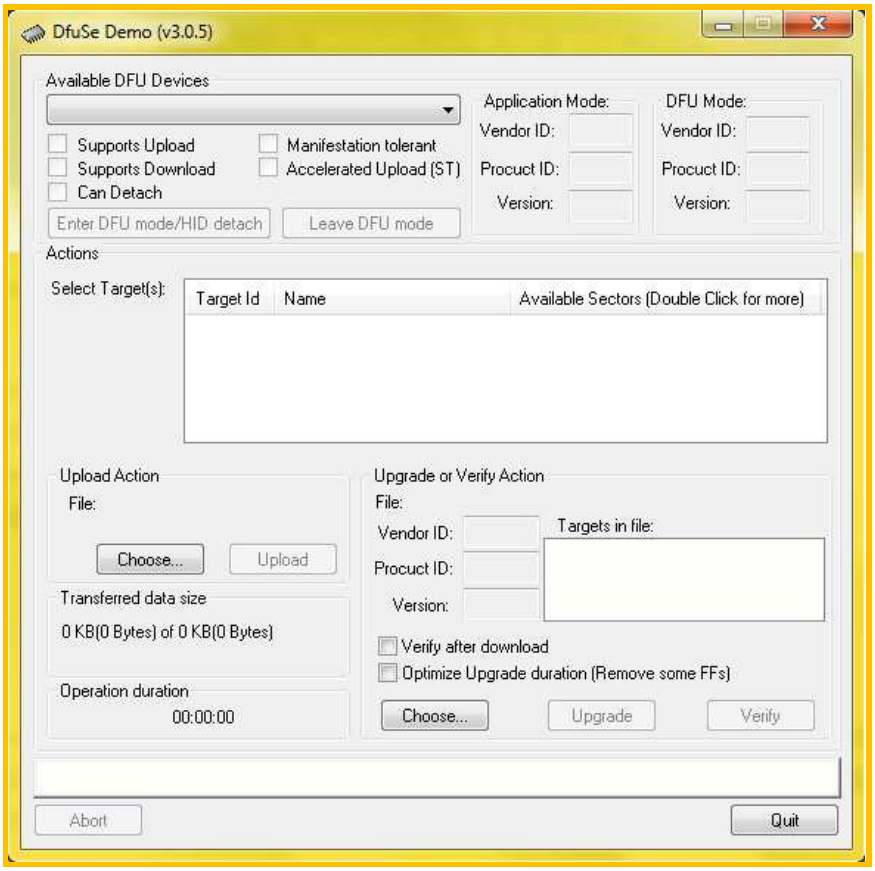
4. 实验

使用 32F411EDISCOVERY 来做实验。

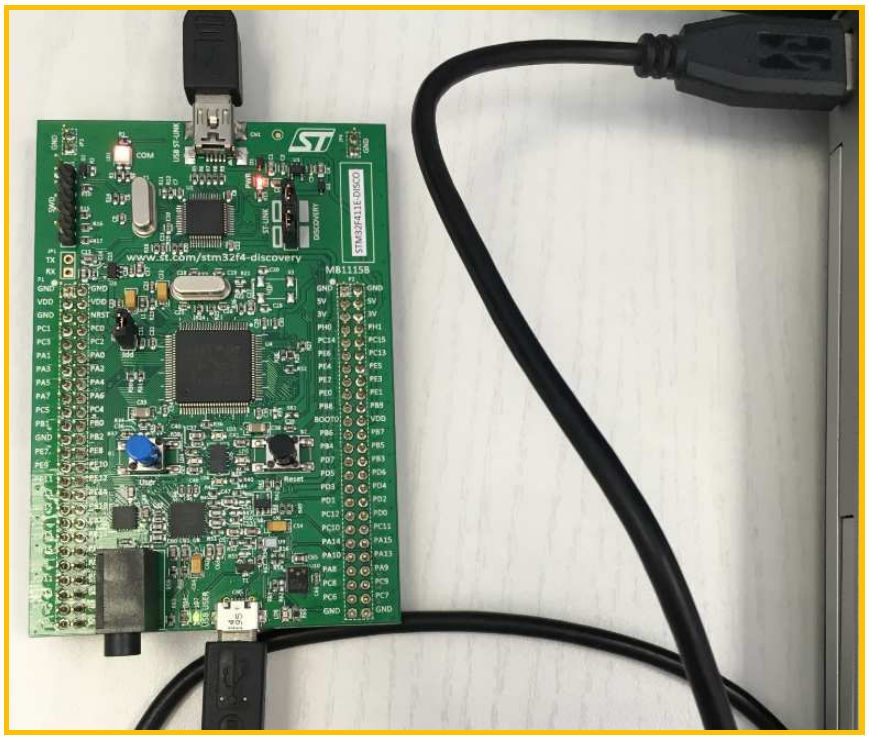
1) 先将程序编译，下载到 32F411EDISCOVERY 板，可以看到 LED3 在进行闪烁

2) 按下 User 按键，LED3 熄灭，已经进入 System Memory 中的 Bootloader

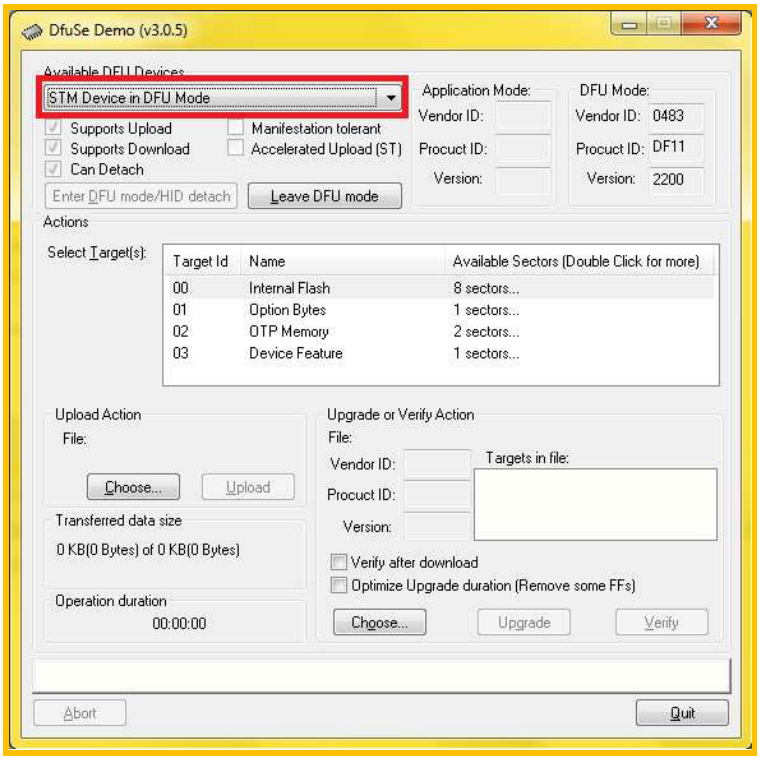
3) 打开 DfuSeDemo 软件，此时 Available DFU Devices 中没有任何显示



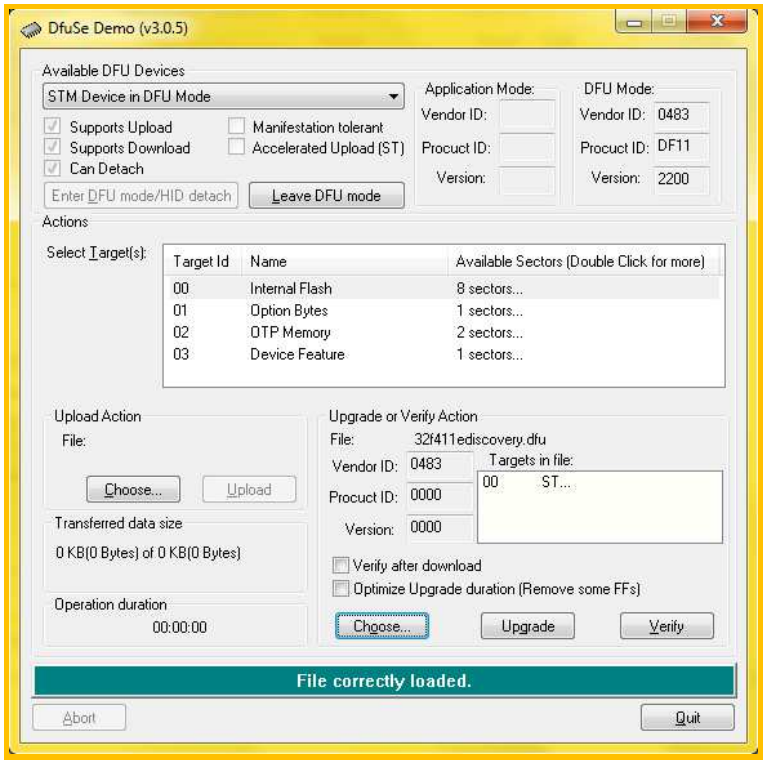
4) 将一根 USB Micro 连接线插入 32F411EDISCOVERY 板的 CN5，可见 LED7 亮起，USB 已连接



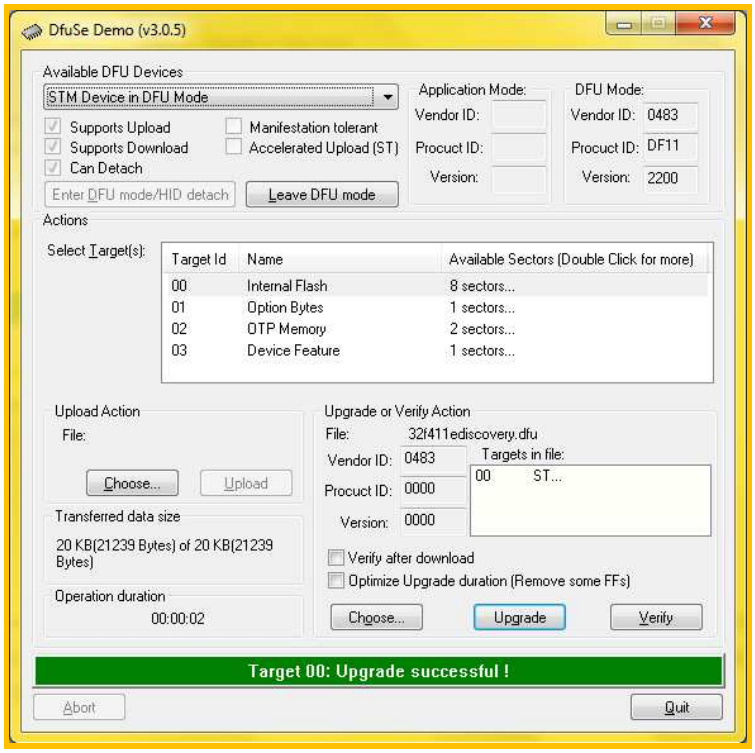
5) 驱动完成后，可以再查看一下 DfuSeDemo，Available DFU Devices 已经显示为“STM Device in DFU Mode”，代表已经成功驱动并正常工作了



6) 之后就是正常的升级代码的流程了，点“Choose”按钮选择要更新的代码，这里准备好了一个 32F411EDISCOVERY 板的 Demo 程序经过 Dfu file manager 软件生成的 32f411ediscovery.dfu 的文件，导入



7) 点“Upgrade”按钮进行升级，弹出的对话框选 Yes 就可以了，之后就升级成功了



8) 再点一下“Leave DFU mode”，进度条显示“Successfully left DFU mode!”，就可以进入更新后的用户代码了，可以看到 4 个 LED 灯正常欢快的滚动和闪烁着……

注意

此例程仅为验证其可行性，在实际应用中，有不尽完善的地方请用户自行完善。另外，有几个需要注意的地方：

1) 此 Demo 代码基于 STM32Cube_FW_F4_V1.11.0 撰写，解压缩后，可将其放入 \STM32Cube_FW_F4_V1.11.0\Projects\STM32F411E-Discovery\Templates 替换掉原来的源代码文件，即可编译运行。
2) 此程序使用按键按下作为条件来触发软件代码升级，用户可以根据自己的情况修改触发条件，如多个按键同时按下，等等。
3) 当用户应用中使用了 RTC 的话，RTC 时钟源一旦被选择后是无法修改的，除非备份域被复位。在 RM0383 关于 RCC_BDCR 的描述中有提及：

> Bits 9:8 **RTCSEL[1:0]:** RTC clock source selection
>
> Set by software to select the clock source for the RTC. Once the RTC clock source has been selected, it cannot be changed anymore unless the Backup domain is reset. The BDRST bit can be used to reset them.

4) 关于如何使用 Dfu file manager 生成.dfu 文件，请参考 UM0412“Getting started with DfuSe USB device firmware upgrade”或者实战经验“利用 USB DFU 实现 IAP 功能”。
5) 关于 Bootloader 中所使用的 USB DFU 协议，请参考 AN3156“USB DFU protocol used in the STM32 bootloader”。

重要通知 – 请仔细阅读

意法半导体公司及其子公司（“ST”）保留随时对ST 产品和/ 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于ST 产品的最新信息。ST 产品的销售依照订单确认时的相关ST 销售条款。

买方自行负责对ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的ST 产品如有不同于此处提供的信息的规定，将导致ST 针对该产品授予的任何保证失效。

ST 和ST 徽标是ST 的商标。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。