STM32F09x 不使用 BOOT 脚实现 System Bootloader 升级代码

前言

众所周知，使用 STM32 时，当需要使用 System Memory 中的 Bootloader 进行代码升级的时候，需要将 BOOT0 脚拉高，复位后才能进入 Bootloader 程序，使用 Flash Loader Demonstrator 等工具进行串口烧写升级。这就需要在 BOOT0 这个引脚上留出按键或者是跳线脚。但是，STM32F09x 具有一项新的特性，使不必使用 BOOT 脚而进行串口升级成为可能。我们来共同探讨一下。

问题

某客户在其产品的设计中，使用了 STM32F091RCT6，产品在出厂后将来可能由于功能的升级需要升级代码。由于外观的需要，客户不希望留一个用于升级的按键或是跳线槽在外边。希望能够是通过接收串口命令来实现启动升级，又希望能够直接使用 System Memory 中的 Bootloader 进行代码升级。

调研

1. 认识一下 STMF09x 和 STM32F04x 在 Boot Configuration 中的新特性

打开参考手册 RM0091，翻到 Boot Configuration 那一节，我们可以看到 Table 3 中对 Boot Mode 进行了描述，如下：

Table 3. Boot modes (1)

Boot mode configuration				Mode
nBOOT1 bit	BOOT0 pin	BOOT_SEL bit	nBOOT0 bit	
x	0	1	x	**Main Flash memory** is selected as boot space(2)
1	1	1	x	**System memory** is selected as boot space
0	1	1	x	**Embedded SRAM** is selected as boot space
x	x	0	1	**Main Flash memory** is selected as boot space
1	x	0	0	**System memory** is selected as boot space
0	x	0	0	**Embedded SRAM** is selected as boot space

1. Grey options are available on STM32F04x and STM32F09x devices only.
2. For STM32F04x and STM32F09x devices, see also Empty check description.

从表格中，我们先来看一下第 1 条注释：

“Grey options are available on STM32F04x and STM32F09x devices only.”

意思是说，灰色部分对 STM32F04x 和 STM32F09x 是有效的。也就是说，除了 BOOT1 是在 Option Byte 中的 nBOOT1 配置之外，STM32F04x 和 STM32F09x 甚至可以将 BOOT0 通过 Option Byte 中的 nBOOT0 来配置，只要将 Option Byte 中的 BOOT_SEL 位配置为 0。这使得纯粹使用软件来实现进入 System Memory 中的 Bootloader 来进行串口升级成为可能。

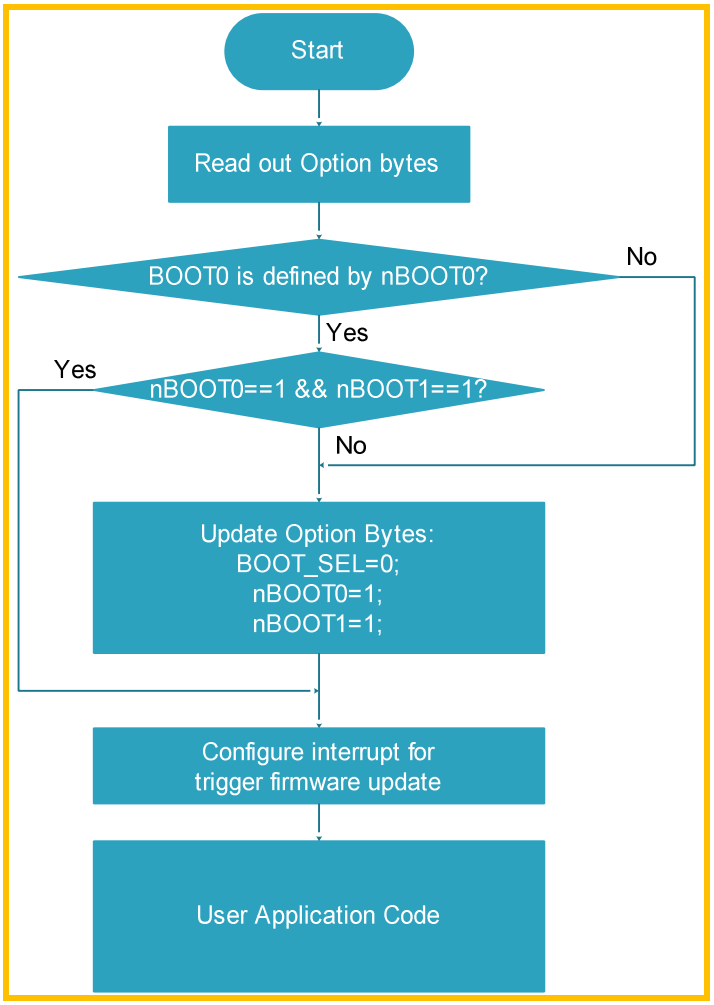
2. 方案设计

既然 STM32F04x 和 STM32F09x 甚至可以将 BOOT0 通过 Option Byte 中的 nBOOT0 来配置，那么基本的思路就是：

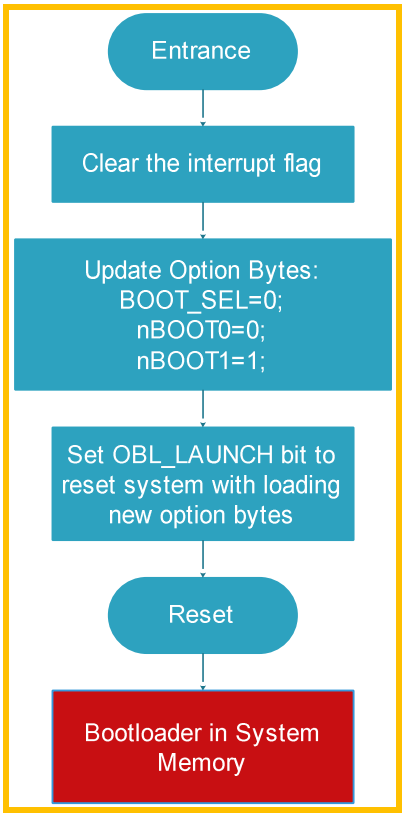
1) 在用户代码中，设置进入串口升级的条件（比如说使用复合按键，或者接收到串口发来的一串命令，等等）；一旦条件成立，就将 Option Byte 中关于 BOOT 的配置设置为：BOOT_SEL=0（BOOT0 信号由 nBOOT0 选项位定义），nBOOT1=1 & nBOOT0=0。然后进行复位，重载 Option Byte，这个时候系统就会进入 System Memory 中去运行 Bootloader。
2) 此时，打开 Flash Loader Demonstrator，可以进行下载，在下载之前我们要设置在烧写完程序后从 Bootloader 跳转到用户代码，因为我们需要在用户代码中去将 BOOT 的配置修改为从 Main Flash memory 启动。如果不这么做，那么每次复位后都将从 System memory 启动。
3) 从第二步可以看到，我们还需要在用户代码的最前面加入判断，如果 BOOT 的配置不是从 Main Flash memory 启动的话，必须对 Option Byte 进行改写，将 BOOT 的配置设置为从 Main Flash memory 启动：BOOT_SEL=0（BOOT0 信号由 nBOOT0 选项位定义），nBOOT1=1 & nBOOT0=1。

从这个思路出发，我们可以得到需要设计的流程图。

主程序的流程图如下：


Start
Read out Option bytes
BOOT0 is defined by nBOOT0?
No
Yes
Yes
nBOOT0==1 && nBOOT1==1?
No
Update Option Bytes:
BOOT_SEL=0;
nBOOT0=1;
nBOOT1=1;
Configure interrupt for
trigger firmware update
User Application Code

当进入串口升级的条件成立时，其流程如下：



实验

1. 设计目标

使用 NUCLEO-F091RC 来实现此实验。空片进行第一次烧写时使用 SWD 进行烧写，之后所有的.hex 文件再次烧写都将使用 Flash Loader Demonstrator 进行下载。不使用 BOOT0 脚。由于只是实验，这里用按下 USER 按键来代表进入串口烧写的条件，用户代码运行 LD2 闪烁的功能。

2. 程序设计

main()主程序如下：

```
int main(void)
{
  /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f0xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f0xx.c file
     */

  /* 以下两行代码，是将向量表映射为 Main Flash memory，否则从 Bootloader 跳转到用户
```

```
    代码时，将无法正常工作。如有不清楚之处，请参考另外一份文档：
    《STM32F091 从自举程序向应用程序跳转的问题与解决》  */
  RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
  SYSCFG->CFGR1 = (uint32_t)0x00000000;

  /* 此处调用子程序，用来判断 BOOT 的配置是否从 Main Flash memory 启动，如果不是，将
     Option Bytes 中的 BOOT 配置设置为从 Main Flash  memory 启动  */
  BOOTCONF_User();

  /* 配置 LED 灯 LD2 的初始化 */
  STM_NUCLEO_LD2Init();
  /* 配置 USER 按键的初始化，设置为中断口 */
  STM_EVAL_PBInit(BUTTON_USER, BUTTON_MODE_EXTI);

  /* 主循环进行闪灯，随时等待按键中断以启动串口烧写 */
  while (1)
  {
    /* Toggle LD2 */
    STM_NUCLEO_LD2Toggle();
    /* Inset delay */
    Delay(0xAFFFF);
  }
}
```

BOOTCONF_User()就是我们思路中的第 3)步，也是我们所关心的，我们来仔细看代码：

```
void BOOTCONF_User(void)
{
  FLASH_Status status = FLASH_COMPLETE;
  /* 定义一个 8 个半字的数组用来存放 option bytes 的数据 */
  uint16_t ob[8];

  /* 读取 Option Bytes 的数据 */
  ob[0] = *(uint16_t*)(0x1FFFF800);    //RDP
  ob[1] = *(uint16_t*)(0x1FFFF802);    //USER
  ob[2] = *(uint16_t*)(0x1FFFF804);    //DATA0
  ob[3] = *(uint16_t*)(0x1FFFF806);    //DATA1
  ob[4] = *(uint16_t*)(0x1FFFF808);    //WRP0
  ob[5] = *(uint16_t*)(0x1FFFF80A);    //WRP1
  ob[6] = *(uint16_t*)(0x1FFFF80C);    //WRP2
  ob[7] = *(uint16_t*)(0x1FFFF80E);    //WPR3

  /* 判断 BOOT 配置是否正确并更新 */
  /* 当从串口烧写完代码之后，从 Bootloader 跳转到用户代码，我们需要将 BOOT 配置设置回来，
     修改为从 Main Flash  memory 启动；平时已经是此设置的时候，就不需要修改了 */
  if((ob[1] & 0x0098) != 0x0018)//判断是否是“BOOT_SEL=0 且 nBOOT1=1 且 nBOOT0=1”,  否则更新
  {
    /* 先解锁 */
```

```
    FLASH_Unlock();
    FLASH_OB_Unlock();

    /* 再擦除，擦除完成后对 option bytes 进行修改更新*/
    /* Wait for last operation to be completed */
    status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
    if(status == FLASH_COMPLETE)
    {
      /* 擦除 */
       /* If the previous operation is completed, proceed to erase the option bytes */
      FLASH->CR |= FLASH_CR_OPTER;
      FLASH->CR |= FLASH_CR_STRT;
      /* Wait for last operation to be completed */
      status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
      /* If the erase operation is completed, disable the OPTER Bit */
      FLASH->CR &= ~FLASH_CR_OPTER;

      /* 更新 option Bytes */
       /* ob[1]对应的是 USER，里边包含了 BOOT_SEL, nBOOT1, nBOOT0，所以我们对 ob[1]中的这三个位
进行修改 */
      ob[1] = (ob[1] & 0x6767) | 0x8018;

        /* 写回 option bytes */
      /* Enable the Option Bytes Programming operation */
      FLASH->CR |= FLASH_CR_OPTPG;
      /* Restore the last read protection Option Byte value */
      OB->RDP = ob[0];
      status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
      if(status == FLASH_TIMEOUT)
        while(1)   ;      //ERROR
      FLASH->CR &= ~FLASH_CR_OPTPG;
      FLASH->CR |= FLASH_CR_OPTPG;
      OB->USER = ob[1];
      status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
      if(status == FLASH_TIMEOUT)
        while(1)   ;      //ERROR
      FLASH->CR &= ~FLASH_CR_OPTPG;
      FLASH->CR |= FLASH_CR_OPTPG;
      OB->DATA0 = ob[2];
      status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
      if(status == FLASH_TIMEOUT)
         while(1)  ;      //ERROR
      FLASH->CR &= ~FLASH_CR_OPTPG;
      FLASH->CR |= FLASH_CR_OPTPG;
      OB->DATA1 = ob[3];
      status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
      if(status == FLASH_TIMEOUT)
        while(1)   ;      //ERROR
```

```
    FLASH->CR &= ~FLASH_CR_OPTPG;
    FLASH->CR |= FLASH_CR_OPTPG;
    OB->WRP0 = ob[4];
    status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
    if(status == FLASH_TIMEOUT)
      while(1)    ;      //ERROR
    FLASH->CR &= ~FLASH_CR_OPTPG;
    FLASH->CR |= FLASH_CR_OPTPG;
    OB->WRP1 = ob[5];
    status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
    if(status == FLASH_TIMEOUT)
      while(1)    ;      //ERROR
    FLASH->CR &= ~FLASH_CR_OPTPG;
    FLASH->CR |= FLASH_CR_OPTPG;
    OB->WRP2 = ob[6];
    status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
    if(status == FLASH_TIMEOUT)
      while(1)    ;      //ERROR
    FLASH->CR &= ~FLASH_CR_OPTPG;
    FLASH->CR |= FLASH_CR_OPTPG;
    OB->WRP3 = ob[7];
    status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
    if(status == FLASH_TIMEOUT)
      while(1)    ;      //ERROR
    FLASH->CR &= ~FLASH_CR_OPTPG;
  }
  else
  {
    if (status != FLASH_TIMEOUT)
    {
      /* Disable the OPTPG Bit */
      FLASH->CR &= ~FLASH_CR_OPTPG;
    }
  }

  /* 写完，重新锁止 */
  FLASH_OB_Lock();
  FLASH_Lock();

  /* 验证 option bytes 数据被正确写入 */
  if((*(uint16_t*)(0x1FFFF800)) != ob[0])
    while(1)      ;      //ERROR
  if((*(uint16_t*)(0x1FFFF802)) != ob[1])
    while(1)      ;      //ERROR
  if((*(uint16_t*)(0x1FFFF804)) != ob[2])
    while(1)      ;      //ERROR
  if((*(uint16_t*)(0x1FFFF806)) != ob[3])
    while(1)      ;      //ERROR
  if((*(uint16_t*)(0x1FFFF808)) != ob[4])
```

```
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF80A)) != ob[5])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF80C)) != ob[6])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF80E)) != ob[7])
      while(1)      ;      //ERROR

      /* 将 FLASH_CR 的 OBL_LAUNCH 位置 1，产生一个可以重新载入 option bytes 的复位
         其目的在于避免此复位和 Power On 的复位以外的其他复位产生时不能正常工作。
         例如，使用串口烧写完代码，从 Bootloader 跳入用户代码，此时如果按下 RESET 按键
         产生复位，则又重新进入 System  Memory 中。所以加此复位，以避免此现象产生。 */
    FLASH_OB_Launch();
  }
}
```

我们再来看按键中断的程序：

```
void EXTI4_15_IRQHandler(void)
{
  if(EXTI_GetITStatus(EXTI_Line13) != RESET)
  {
    EXTI_ClearITPendingBit(USER_BUTTON_EXTI_LINE);
    BOOTCONF_System();
    FLASH_OB_Launch();
  }
}
```

这个代码很简单，就是清除一下中断标志位，然后将 Option Bytes 中的 BOOT 配置设置为从 System memory 启动，然后再调用 FLASH_OB_Launch()来产生一个可以重新载入 option bytes 的复位，这个复位将导致复位后进入的是 System memory，而不是 Main Flash memory 了。

其中关键的代码是 BOOTCONF_System()，我们来看一下：

```
void BOOTCONF_System(void)
{
  FLASH_Status status = FLASH_COMPLETE;
  uint16_t ob[8];

  /* Get opiton bytes data */
  ob[0] = *(uint16_t*)(0x1FFFF800);    //RDP
  ob[1] = *(uint16_t*)(0x1FFFF802);    //USER
  ob[2] = *(uint16_t*)(0x1FFFF804);    //DATA0
  ob[3] = *(uint16_t*)(0x1FFFF806);    //DATA1
  ob[4] = *(uint16_t*)(0x1FFFF808);    //WRP0
  ob[5] = *(uint16_t*)(0x1FFFF80A);    //WRP1
  ob[6] = *(uint16_t*)(0x1FFFF80C);    //WRP2
  ob[7] = *(uint16_t*)(0x1FFFF80E);    //WPR3

  /* Unlock */
```

```
FLASH_Unlock();
FLASH_OB_Unlock();
/* Erase and update Option bytes*/
/* Wait for last operation to be completed */
status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
if(status == FLASH_COMPLETE)
{
  /* If the previous operation is completed, proceed to erase the option bytes */
  FLASH->CR |= FLASH_CR_OPTER;
  FLASH->CR |= FLASH_CR_STRT;
  /* Wait for last operation to be completed */
  status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
  /* If the erase operation is completed, disable the OPTER Bit */
  FLASH->CR &= ~FLASH_CR_OPTER;

  /* Update option bytes */
  ob[1] = (ob[1] & 0xFFF7) | 0x0800;
  /* Enable the Option Bytes Programming operation */
  FLASH->CR |= FLASH_CR_OPTPG;
  /* Restore the last read protection Option Byte value */
  OB->RDP = ob[0];
  status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
  if(status == FLASH_TIMEOUT)
    while(1)      ;      //ERROR
  FLASH->CR &= ~FLASH_CR_OPTPG;
  FLASH->CR |= FLASH_CR_OPTPG;
  OB->USER = ob[1];
  status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
  if(status == FLASH_TIMEOUT)
    while(1)      ;      //ERROR
  FLASH->CR &= ~FLASH_CR_OPTPG;
  FLASH->CR |= FLASH_CR_OPTPG;
  OB->DATA0 = ob[2];
  status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
  if(status == FLASH_TIMEOUT)
    while(1)      ;      //ERROR
  FLASH->CR &= ~FLASH_CR_OPTPG;
  FLASH->CR |= FLASH_CR_OPTPG;
  OB->DATA1 = ob[3];
  status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
  if(status == FLASH_TIMEOUT)
    while(1)      ;      //ERROR
  FLASH->CR &= ~FLASH_CR_OPTPG;
  FLASH->CR |= FLASH_CR_OPTPG;
  OB->WRP0 = ob[4];
  status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
  if(status == FLASH_TIMEOUT)
    while(1)      ;      //ERROR
  FLASH->CR &= ~FLASH_CR_OPTPG;
```

```
    FLASH->CR |= FLASH_CR_OPTPG;
    OB->WRP1 = ob[5];
    status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
    if(status == FLASH_TIMEOUT)
      while(1)      ;      //ERROR
    FLASH->CR &= ~FLASH_CR_OPTPG;
    FLASH->CR |= FLASH_CR_OPTPG;
    OB->WRP2 = ob[6];
    status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
    if(status == FLASH_TIMEOUT)
      while(1)      ;      //ERROR
    FLASH->CR &= ~FLASH_CR_OPTPG;
    FLASH->CR |= FLASH_CR_OPTPG;
    OB->WRP3 = ob[7];
    status = FLASH_WaitForLastOperation(FLASH_ER_PRG_TIMEOUT);
    if(status == FLASH_TIMEOUT)
      while(1)      ;      //ERROR
    FLASH->CR &= ~FLASH_CR_OPTPG;
    }
    else
    {
      if (status != FLASH_TIMEOUT)
      {
        /* Disable the OPTPG Bit */
        FLASH->CR &= ~FLASH_CR_OPTPG;
      }
    }

    /* Lock */
    FLASH_OB_Lock();
    FLASH_Lock();

    /* Verify */
    if((*(uint16_t*)(0x1FFFF800)) != ob[0])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF802)) != ob[1])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF804)) != ob[2])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF806)) != ob[3])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF808)) != ob[4])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF80A)) != ob[5])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF80C)) != ob[6])
      while(1)      ;      //ERROR
    if((*(uint16_t*)(0x1FFFF80E)) != ob[7])
      while(1)      ;      //ERROR
```

```
}
```

我们可以看到，其实其整个过程与 BOOTCONF_User()非常地像，唯独有两个区别：

1) 没有判断当前的 BOOT 配置，并且在更新 USER 时只是将 nBOOT0 清零而已。因为在用户代码的最开始，我们已经保证了，BOOT 的配置一定是从 Main Flash memory 启动，所以要切换到从 System memory 启动，仅需要将 nBOOT0 清零。
2) Option Bytes 更新并验证后，并没有调用 FLASH_OB_Launch()。因为我们把它放在中断程序里头了。

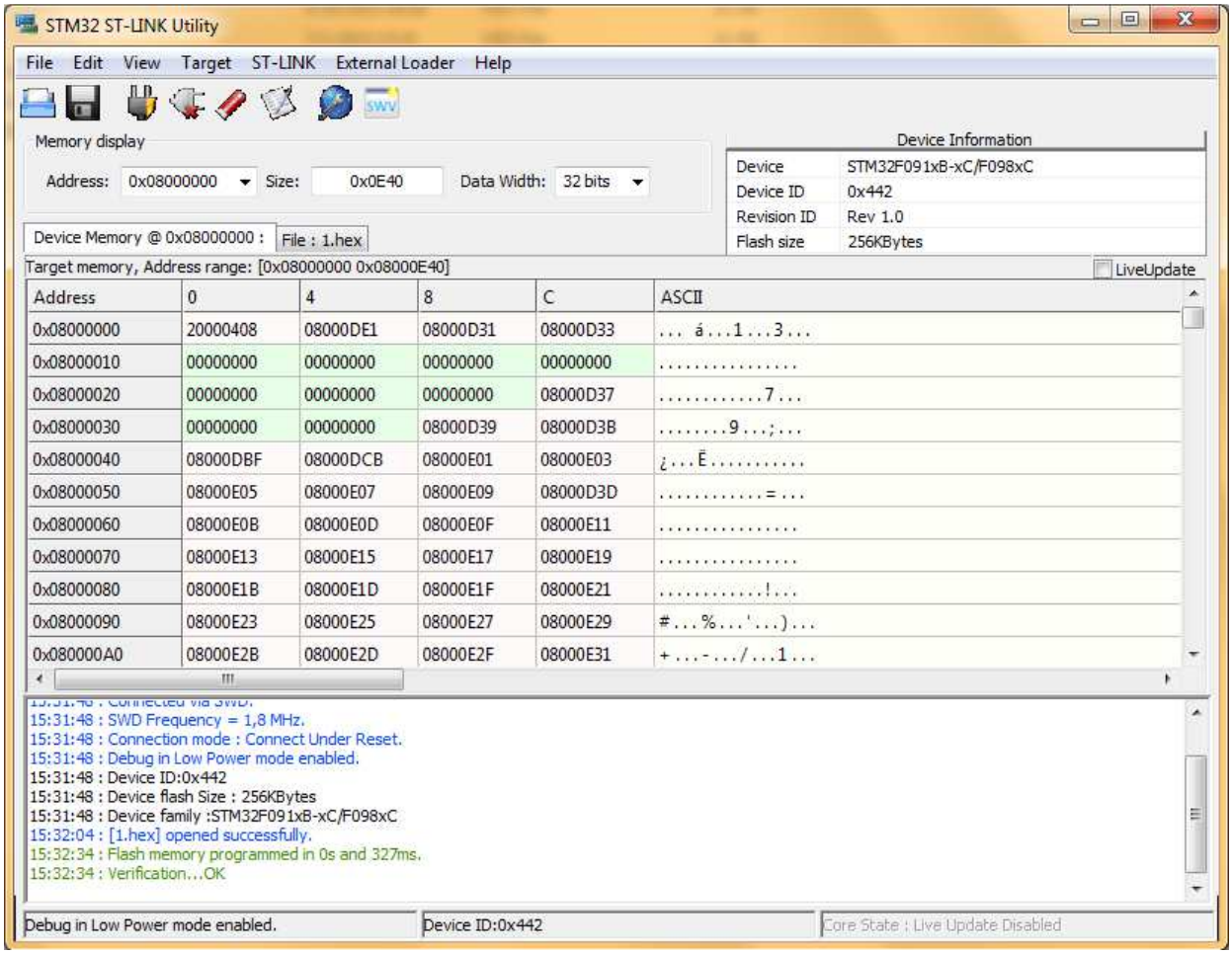
程序要点到这里就已经很清楚了。此程序仅为实验用，所以对于出错处理和代码优化，可根据实际应用自行处理。

3. 实验步骤

下面我们来进行实验：

1) 我们将程序编译生成的的.hex（我们将其命名为 1.hex）通过 SWD 接口，使用 STM32 ST-LINK Utility 烧写到 NUCLEO_F091RC 后，可以看到 STM32F091RC 正在欢快地运行 1.hex，闪烁着 LD2。关闭 STM32 ST-LINK Utility。

   注：1.hex 是慢闪烁现象

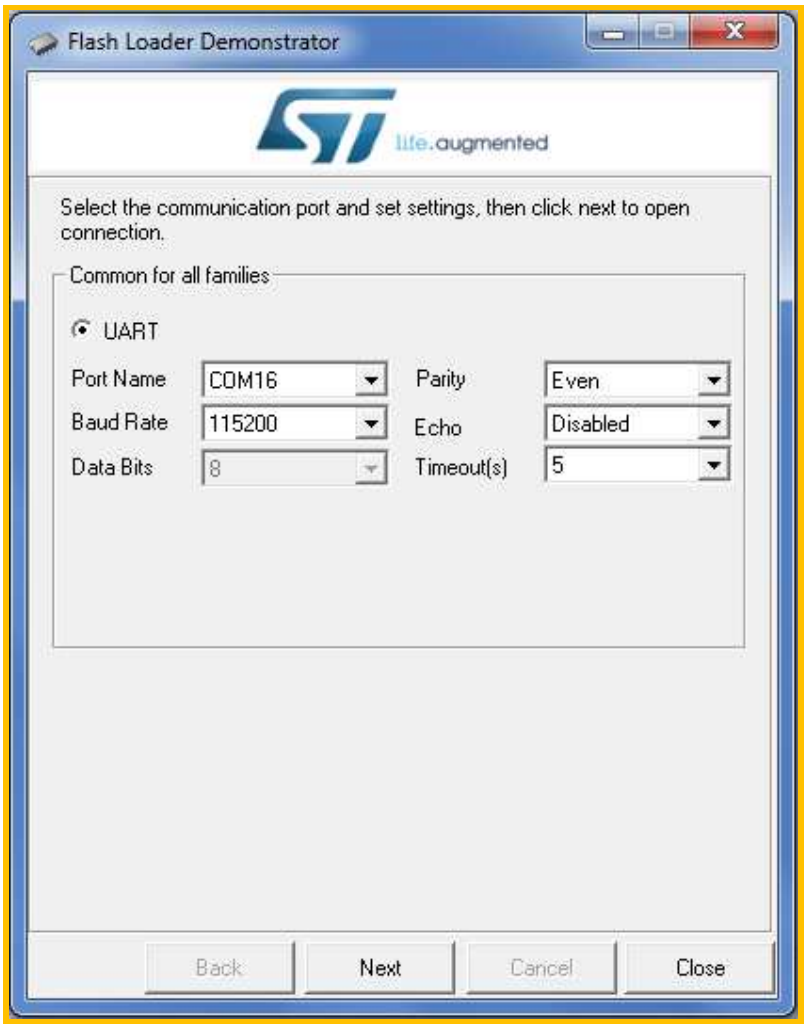


2) 回到程序中，将 main()函数中主循环的“Delay(0xAFFFF);”修改为“Delay(0xFFFF);”，重新编译生成.hex 文件，我们将其命名为 2.hex。

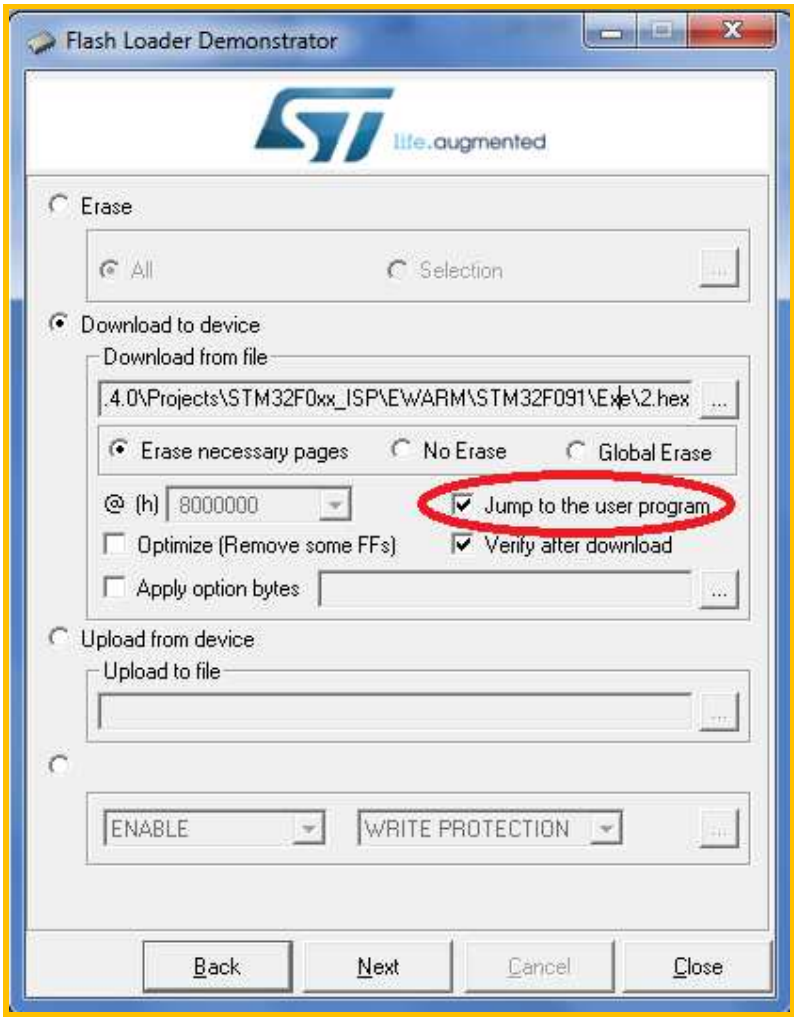
注：2.hex 是快闪烁现象

3) 现在我们要开始进行串口升级了。按下 USER 按键，发现 LD2 不再闪烁了，程序已经进入 System memory 运行 Bootloader 了。

4) 打开 Flash Loader Demonstrator，做以下的配置：



5) 点“Next”进入下一个界面，再点“Next ”再进入下一个界面，再点“Next ”进入下一个界面，我们在这个页面中选择下载代码“Download to device”，导入刚才编译生成的 2.hex，选中“Jump to the user program ”，意思就是通过串口烧写完毕后，直接跳到用户代码去运行。如图：



6) 点击“Next”进行代码烧写。烧写成功后，我们就可以看到 LD2 更加欢快地闪烁了，证明我们的烧写是成功的。按下 RESET 按键，也能正常运行代码；断电后再上电，也正常运行。

到这里，我们的实验成功了，我们可以重复 3)~6)步骤来反复烧写 1.hex 和 2.hex，体验这个串口升级的方便性，再也不用去拔跳线插跳线了。

结论

由于 STM32F04x 和 STM32F09x 可以将 BOOT0 通过 Option Byte 中的 nBOOT0 来配置，这个新特性决定了我们可以很方便地实现不使用 BOOT 脚就直接从 System memory 中的 Bootloader 进行代码升级。非常地方便。

处理

此实验证明了“STM32F04x 和 STM32F09x 不使用 BOOT 脚实现 System Bootloader 升级代码”的可行性。但只是实验，客户可自行处理错误情况和代码优化。此外，此实验用按下 USER 按键来激活串口烧写，客户可自行根据应用来修改激活条件，比如从串口接到到一串固定的数据作为激活条件。

后续

其实我们在看参考手册时，还看到这么一句话：

“For STM32F04x and STM32F09x devices, see also Empty check description.”

也就是说，空片也是可以通过串口进行烧写的，我们后续继续讨论。