

***Package na***

文本框

**100ns**

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# 问题

1、wifi模式怎么切换

2、

3、

4、

192.168.100.253

# 注意地方

1、可以对比一下

LoRa\_Serial\_IAP、LoRa-Serial工程区别

2、

# nodetools

192.168.100.253

# 超级终端



1、保证 LORA中带有boot程序

2、

# BIN和HEX文件的区别

HEX文件和BIN文件是我们经常碰到的2种文件格式。下面简单介绍一下这2种文件格式的区别：

1、HEX文件是包括地址信息的，***而BIN文件格式只包括了数据本身***。在烧写或下载HEX文件的时候，一般都不需要用户指定地址，因为HEX文件内部的信息已经包括了地址。

***而烧写BIN文件的时候，用户是一定需要指定地址信息的***。

2、BIN文件格式，对二进制文件而言，其实没有”格式”。文件只是包括了纯粹的二进制数据。

3、HEX文件格式，HEX文件都是由记录（RECORD）组成的。在HEX文件里面，每一行代表一个记录。记录的基本格式为：



# IAP 的BootLoader程序

### 参考链接：

<https://www.cnblogs.com/outs/p/4948134.html>

STM32F0系列MCU中断向量表的重映射

<https://blog.csdn.net/weixin_44788542/article/details/114373763>

STM32F0芯片IAP实现之中断向量表重映射（这里两个是因为M0的内核没有SCB->VTOR寄存器，无法使用NVIC\_SetVectorTable(uint32\_t NVIC\_VectTab, uint32\_t Offset)函数的一种解决方法案例）

<https://zhuanlan.zhihu.com/p/530414350>

STM32固件IAP程序实现

### 总结

1、通常情况下 不论是IAR还是KEIL。向量表的地址和code存储到flash中的地址一致。





可以看见这个地址

### 启动配置（F103中文参考手册2.4 启动配置）

在STM32F10xxx里，可以通过BOOT[1:0]引脚选择三种不同启动模式。



在系统复位后， SYSCLK的第4个上升沿， BOOT引脚的值将被锁存。用户可以通过设置BOOT1和BOOT0引脚的状态，来选择在复位后的启动模式。

在从待机模式退出时， BOOT引脚的值将被被重新锁存；因此，在待机模式下BOOT引脚应保持为需要的启动配置。

1、在启动延迟之后， **CPU从地址0x0000 0000获取堆栈顶的地址，并从启动存储器的0x0000 0004指示的地址开始执行代码**。

2、因为固定的存储器映像，**代码区始终从地址0x0000 0000开始**(通过ICode和DCode总线访问)，而数据区(SRAM)始终从地址0x2000 0000开始(通过系统总线访问)。

3、Cortex-M3的CPU始终从ICode总线获取复位向量，即启动仅适合于从代码区开始(典型地从Flash启动)。

4、STM32F10xxx微控制器实现了一个特殊的机制，系统可以不仅仅从Flash存储器或系统存储器启动，还可以从内置SRAM启动。

根据选定的启动模式，主闪存存储器、系统存储器或SRAM可以按照以下方式访问：

● 从主闪存存储器启动：**主闪存存储器（0x0800 0000，这个地址可以偏移一定的量）被映射到启动空间(0x0000 0000)**，但仍然能够在它原有的地址(0x0800 0000)访问它，即闪存存储器的内容可以在两个地址区域访问， 0x00000000或0x0800 0000。

● 从系统存储器启动：系统存储器被映射到启动空间(0x0000 0000)，但仍然能够在它原有的地址(互联型产品原有地址为0x1FFF B000，其它产品原有地址为0x1FFF F000)访问它。

● 从内置SRAM启动：只能在0x2000 0000开始的地址区访问SRAM。

### IAP实现的重要点

* CM3内核在响应中断时，会根据中断号在中断向量表中找到中断服务例程的地址，而在跳转到中断服务程序的时候，必须加上一个偏移量offset,
  + 当中断向量表在FLASH中的时候，就是0x08000000+offset,
  + 当在SRAM中的时候，是在0x20000000+offset,
  + 也就是说，void NVIC\_SetVectorTable(uint32\_t NVIC\_VectTab, uint32\_t Offset)函数的作用就是告诉内核，应用程序的中断向量表是在FLASH中还是在SRAM中，偏移量是多少。
* 也就说，无论是什么应用程序，必须在0x08000000处存放中断向量表，至少要有复位中断向量，这样内核在上电之后，首先从0x08000000处加载栈指针，然后再0x08000004处加载中断服务例程的地址。也就是说，至少有一个应用程序的必须在0x08000000处存放中断向量表，好比说一个bootloader和app，那么bootloader就要放在0X08000000处。具体原因在STM32的中文参考手册中可以找到的（**CPU从地址0x0000 0000获取堆栈顶的地址，并从启动存储器的0x0000 0004指示的地址开始执行代码**。STM32程序执行的硬件入口）
* 在IAP中，一般会有两个中断向量表，其中Bootloader有一个，APP有一个。
  + Bootloader的向量表就存在0X08000000处，这样上电之后首先运行的就是Bootloader,
  + bootloader进行APP代码的搬移，加载APP栈指针，最后执行跳转，跳转到APP的中断向量表的起始位置(如果APP是运行在FLASH中，这个地址是0x08000000+offset)，注意，APP的工程中需要一定的设置，并在系统初始化的时候使用函数void NVIC\_SetVectorTable(uint32\_t NVIC\_VectTab, uint32\_t Offset) 告诉内核APP向量表的区域和位置。
* SCB是系统控制块，主要封装了内核相关的寄存器的设置，具体的内容请参考《CM3权威指南》。

### void NVIC\_SetVectorTable(uint32\_t NVIC\_VectTab, uint32\_t Offset)

|  |
| --- |
| /\*\*  \* @brief Sets the vector table location and Offset.  \* @param NVIC\_VectTab: specifies if the vector table is in RAM or FLASH memory.  \* This parameter can be one of the following values:  \* @arg NVIC\_VectTab\_RAM  \* @arg NVIC\_VectTab\_FLASH  \* @param Offset: Vector Table base offset field. This value must be a multiple of 0x100.  \* @retval None  \*/  void NVIC\_SetVectorTable(uint32\_t NVIC\_VectTab, uint32\_t Offset)  {  /\* Check the parameters \*/  assert\_param(IS\_NVIC\_VECTTAB(NVIC\_VectTab));  assert\_param(IS\_NVIC\_OFFSET(Offset));    SCB->VTOR = NVIC\_VectTab | (Offset & (uint32\_t)0x1FFFFF80);  } |

NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH,(USER\_APP\_BEGIN - NVIC\_VectTab\_FLASH)); //重映射中断向量表

#### 典型代码

##### BootLoader

|  |
| --- |
| #define USER\_APP\_BEGIN (unsigned)(0x08000000)  int main(void)  {  extern unsigned char userdata[];  user\_data\_t \*p = (user\_data\_t\*)userdata;    NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH,(USER\_APP\_BEGIN - NVIC\_VectTab\_FLASH)); //重映射中断向量表  if(IWDG->RLR != 0x0FFF) {  watch\_dog\_flag = 1;  IWDG\_ReloadCounter();  } |

由于BootLoader放在前面，所以这里的地址为flash的起始地址。

##### APP程序

|  |
| --- |
| // APP区地址:0x08010000  #define USER\_APP\_BEGIN ((uint32\_t)0x08010000)  void NVIC\_Configuration(void)  {  #ifdef VECT\_TAB\_RAM  /\* Set the Vector Table base location at 0x20000000 \*/  NVIC\_SetVectorTable(NVIC\_VectTab\_RAM, 0x0);  #else /\* VECT\_TAB\_FLASH \*/  /\* Set the Vector Table base location at 0x08000000 \*/  // NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH, 0x0);  **NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH, (USER\_APP\_BEGIN - NVIC\_VectTab\_FLASH));**  #endif  NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_2);  } |

#### SCB->VTOR

这个寄存器不仅仅在NVIC\_SetVectorTable函数中使用到，其实在stm32启动时候都会使用到。

具体可以在SystemInit函数中查看的到。

由于每一个编译器中。都会使用链接器帮助我们生成对应bin文件，然后通过软件帮助我们下载到flash对应的一个地址，然后此时我们也需要在代码中告知单片机我们用户程序的起始地址，也就是向量表的起始地址。这样单片机发生中断时候就能够在向量表中找到中断函数在code中的具体地址，然后运行相应的代码。



#define NVIC\_VectTab\_FLASH ((uint32\_t)0x08000000) // misc.h

NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH, 0x00010000); //重新配置中断向量表

配合地址的改变，这样程序运行的时候才能找到中断服务函数的入口地址，正常执行。

#### 函数作用、中断向量表

##### 中断向量表和flash程序地址



我们知道keil中，程序刚运行时候就是在就会运行到这个复位中断函数。具体怎么执行的往下看

***1、顾名思义，这个表存放的是中断服务函数的入口地址，当发生中断时，CPU来这个表中查询，以此获取中断函数的入口地址***。

**2、在stm32 的启动文件中，设置完成堆栈，就来设置 中断向量表，**

这个是KEIL的版本

AREA RESET, DATA, READONLY ;

定义只读数据段，实际上是在CODE区（**假设STM32从FLASH启动，则此中断向量表起始地址即为0x8000000**）

如果改变了程序的起始地址，这种就是IAP的情况了，像这个样子，那么这个时候就需要重新设置中断向量表。

|  |
| --- |
| #define NVIC\_VectTab\_FLASH ((uint32\_t)0x08000000) // misc.h  NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH, 0x00010000); //重新配置中断向量表 |

***3、配合地址的改变，这样程序运行的时候才能找到中断服务函数的入口地址，正常执行***。

4、这个向量表的编写是有讲究的，跟硬件一一对应不能乱写的，CPU找入口地址就靠它了，bin文件开头就是他们的地址

##### 中断函数地址在bin文件、hex的体现



使能bin文件生成

向量表地址

复位函数地址

栈顶地址

我们能够看到向量表起始地址也变为了0x8001000

reset\_hander位于0x0800113d这个地址上



使用winhex软件打开这个bin文件后，能够看到

第一个4字节为E8 1E 00 20 20001EE8 正好是map文件中的 \_\_initial\_sp 变量地址

第二个4字节为3D 11 00 08 0800113D 正好是map文件中Reset\_Handler 中断函数地址

（**CPU从地址0x0000 0000获取堆栈顶的地址，并从启动存储器的0x0000 0004指示的地址开始执行代码**。）具体见STM32中文手册。

#### system\_stm32f4xx.c文件

SystemInit函数

但是在RTT的标准版本中，还是stm32裸机程序中，都会调用SystemInit函数，位于system\_stm32fxxx.c这样的文件里面

这个初始化向量表的代码位于SystemInit函数



**SystemInit函数中也会设置SCB->VTOR寄存器**

|  |
| --- |
| void SystemInit(void)  {  /\* 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;  #if defined (DATA\_IN\_ExtSRAM) || defined (DATA\_IN\_ExtSDRAM)  SystemInit\_ExtMemCtl();  #endif /\* DATA\_IN\_ExtSRAM || DATA\_IN\_ExtSDRAM \*/  /\* 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**  } |

/\*!< Uncomment the following line if you need to relocate your vector Table in

Internal SRAM. \*/

/\* #define VECT\_TAB\_SRAM \*/

**#define VECT\_TAB\_OFFSET 0x00 /**\*!< Vector Table base offset field.

This value must be a multiple of 0x200. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#### startup\_stm32f10x.s

下载时0x08000000作为flash的下载地址

keil的裸机程序

|  |
| --- |
| ; Vector Table Mapped to Address 0 at Reset  AREA RESET, DATA, READONLY  EXPORT \_\_Vectors  EXPORT \_\_Vectors\_End  EXPORT \_\_Vectors\_Size  **\_\_Vectors DCD \_\_initial\_sp ; Top of Stack**  **DCD Reset\_Handler ; Reset Handler**  DCD NMI\_Handler ; NMI Handler  DCD HardFault\_Handler ; Hard Fault Handler  DCD MemManage\_Handler ; MPU Fault Handler  DCD BusFault\_Handler ; Bus Fault Handler  DCD UsageFault\_Handler ; Usage Fault Handler  DCD 0 ; Reserved  DCD 0 ; Reserved  DCD 0 ; Reserved  DCD 0 ; Reserved  DCD SVC\_Handler ; SVCall Handler  DCD DebugMon\_Handler ; Debug Monitor Handler  DCD 0 ; Reserved  DCD PendSV\_Handler ; PendSV Handler  DCD SysTick\_Handler ; SysTick Handler  ; External Interrupts  DCD WWDG\_IRQHandler ; Window Watchdog  DCD PVD\_IRQHandler ; PVD through EXTI Line detect  DCD TAMPER\_IRQHandler ; Tamper  DCD RTC\_IRQHandler ; RTC  DCD FLASH\_IRQHandler ; Flash  DCD RCC\_IRQHandler ; RCC  DCD EXTI0\_IRQHandler ; EXTI Line 0  DCD EXTI1\_IRQHandler ; EXTI Line 1  DCD EXTI2\_IRQHandler ; EXTI Line 2  DCD EXTI3\_IRQHandler ; EXTI Line 3  DCD EXTI4\_IRQHandler ; EXTI Line 4  DCD DMA1\_Channel1\_IRQHandler ; DMA1 Channel 1  DCD DMA1\_Channel2\_IRQHandler ; DMA1 Channel 2  DCD DMA1\_Channel3\_IRQHandler ; DMA1 Channel 3  DCD DMA1\_Channel4\_IRQHandler ; DMA1 Channel 4  DCD DMA1\_Channel5\_IRQHandler ; DMA1 Channel 5  DCD DMA1\_Channel6\_IRQHandler ; DMA1 Channel 6  DCD DMA1\_Channel7\_IRQHandler ; DMA1 Channel 7  DCD ADC1\_2\_IRQHandler ; ADC1\_2  DCD USB\_HP\_CAN1\_TX\_IRQHandler ; USB High Priority or CAN1 TX  DCD USB\_LP\_CAN1\_RX0\_IRQHandler ; USB Low Priority or CAN1 RX0  DCD CAN1\_RX1\_IRQHandler ; CAN1 RX1  DCD CAN1\_SCE\_IRQHandler ; CAN1 SCE  DCD EXTI9\_5\_IRQHandler ; EXTI Line 9..5  DCD TIM1\_BRK\_IRQHandler ; TIM1 Break  DCD TIM1\_UP\_IRQHandler ; TIM1 Update  DCD TIM1\_TRG\_COM\_IRQHandler ; TIM1 Trigger and Commutation  DCD TIM1\_CC\_IRQHandler ; TIM1 Capture Compare  DCD TIM2\_IRQHandler ; TIM2  DCD TIM3\_IRQHandler ; TIM3  DCD TIM4\_IRQHandler ; TIM4  DCD I2C1\_EV\_IRQHandler ; I2C1 Event  DCD I2C1\_ER\_IRQHandler ; I2C1 Error  DCD I2C2\_EV\_IRQHandler ; I2C2 Event  DCD I2C2\_ER\_IRQHandler ; I2C2 Error  DCD SPI1\_IRQHandler ; SPI1  DCD SPI2\_IRQHandler ; SPI2  DCD USART1\_IRQHandler ; USART1  DCD USART2\_IRQHandler ; USART2  DCD USART3\_IRQHandler ; USART3  DCD EXTI15\_10\_IRQHandler ; EXTI Line 15..10  DCD RTCAlarm\_IRQHandler ; RTC Alarm through EXTI Line  DCD USBWakeUp\_IRQHandler ; USB Wakeup from suspend  \_\_Vectors\_End  \_\_Vectors\_Size EQU \_\_Vectors\_End - \_\_Vectors  AREA |.text|, CODE, READONLY  **; Reset handler**  **Reset\_Handler PROC**  **EXPORT Reset\_Handler [WEAK]**  **IMPORT \_\_main**  **IMPORT SystemInit**  **LDR R0, =SystemInit**  **BLX R0**  **LDR R0, =\_\_main**  **BX R0**  **ENDP** |

##### 对应生成的map文件

**向量表的起始地址就是bin文件的起始地址。**

\_\_Vectors 0x08000000 Data 4 startup\_stm32f10x\_md.o(RESET) ***可以看到向量表起始地址就是flash的起始地址***

\_\_Vectors\_End 0x080000ec Data 0 startup\_stm32f10x\_md.o(RESET)

|  |
| --- |
| Reset\_Handler 0x0800013d Thumb Code 8 startup\_stm32f10x\_md.o(.text) 复位函数的地址  ADC1\_2\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  CAN1\_RX1\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  CAN1\_SCE\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  DMA1\_Channel1\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  DMA1\_Channel2\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  DMA1\_Channel3\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  DMA1\_Channel4\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  DMA1\_Channel5\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  DMA1\_Channel6\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  DMA1\_Channel7\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  EXTI0\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  EXTI15\_10\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  EXTI1\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  EXTI2\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  EXTI3\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  EXTI4\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  EXTI9\_5\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  FLASH\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  I2C1\_ER\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  I2C1\_EV\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  I2C2\_ER\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  I2C2\_EV\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  PVD\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  RCC\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  RTCAlarm\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  RTC\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  SPI1\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  SPI2\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  TAMPER\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  TIM1\_BRK\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  TIM1\_CC\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  TIM1\_TRG\_COM\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  TIM1\_UP\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  TIM2\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  TIM3\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  TIM4\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  USART1\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  USART3\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  USBWakeUp\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  USB\_HP\_CAN1\_TX\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  USB\_LP\_CAN1\_RX0\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  WWDG\_IRQHandler 0x08000157 Thumb Code 0 startup\_stm32f10x\_md.o(.text)  \_\_initial\_sp 0x20001ee8 Data 0 startup\_stm32f10x\_md.o(STACK) |

### 一些技术上的验证

#### 1、编译器中设置的rom起始地址，hex中包含了程序下载位置





在jlink的下载软件中，就可能看到程序下载的地址是0x8001000

|  |
| --- |
| Programming and verifying target (11100 bytes, 1 range) ...  - Checking if selected data fits into selected flash sectors.  - Start of preparing flash programming  - End of preparing flash programming  - Start of determining dirty areas in flash cache  - End of determining dirty areas  - CPU speed could not be measured.  - Start of flash programming  **- Programming range 0x08001000 - 0x08003BFF ( 11 Sectors, 11 KB) 可以看见下载到STM32中是这些地址**  - End of flash programming  - Flash programming performed for 1 range (11264 bytes)  - 0x8001000 - 0x8003BFF ( 11 Sectors, 11 KB)  - Start of verifying flash  - End of verifying flash  - Start of restoring  - End of restoring  - Target programmed and verified successfully (CRC = 0x99C983CB) - Completed after 0.439 sec |

**说明：编译器中设置的起始FLASH地址，我们下载时候就是将bin文件下载到这个地址为起点的位置（0x8001000）。其他编译器不管了。**

#### 2、如果我们将用户程序设置下载到起点为0x8001000，没有BootLoader，那么单片机能够运行起来么

##### 无法运行



|  |
| --- |
| **#define USER\_APP\_BEGIN (unsigned)(0x8001000)**  int main(void)  {  extern unsigned char userdata[];  user\_data\_t \*p = (user\_data\_t\*)userdata;    **NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH,(USER\_APP\_BEGIN - NVIC\_VectTab\_FLASH));//重映射中断向量表** |

这里是BootLoader 的程序，如果我们不下载到0x8000000这个地址上，而是下载到0x8001000，程序能够运行么？

首先使用jlink擦除全片，



下载成功。程序运行后发现根本没法运行，我们可以调试看看PC指针。



调试时候发现这里的PC指针根本没有指向一个函数。因为此时0x0000 0004 （flash启动的话就是0x80000000）位置的flash被擦除，就是0xFFFFFFFF，程序无法运行。

##### 正常运行



|  |
| --- |
| #define USER\_APP\_BEGIN (unsigned)(0x8000000) 因为BootLoader的向量表就是程序的起始地址 |





**为什么这里的PC为0x0800013c，而不是MAP文件中的0x0800013d，注意PC指针指向的地址为函数在map中地址减一。**

此时可以看到程序正常运行。

# STM32F407工程

# 常见flash字节数字

stm32f407zgt6

ROM ： 起始地址0x08000000 0x080FFFFF

**b：字节**

**0x100b 256B 16^2**

**0x200b 512B 16^2\*2**

**0x400b 1KB 16^2\*4**

**0x1000b 4KB 16^3**

**000-3FF 400-7FF 800-BFF C00-FFF 4KB的空间分为4个1KB区域划分**

**000-1FF 200-3FF 400-5FF 600-7FF 800-9FF A00-BFF C00-DFF E00-FFF 4KB的空间分为8个512B区域划分**

**0x10000b 64KB 16^4**

**0x20000b 128KB 16^4\*2**

**0xA0000b 640KB**

**0x100000b 1024KB**

FF+1=0x100个4K=256\*4=1024KB

/\* Base address of the Flash sectors \*/

#define ADDR\_FLASH\_SECTOR\_0 ((uint32\_t)0x08000000) /\* Base @ of Sector 0, 16 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_1 ((uint32\_t)0x08004000) /\* Base @ of Sector 1, 16 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_2 ((uint32\_t)0x08008000) /\* Base @ of Sector 2, 16 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_3 ((uint32\_t)0x0800C000) /\* Base @ of Sector 3, 16 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_4 ((uint32\_t)0x08010000) /\* Base @ of Sector 4, 64 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_5 ((uint32\_t)0x08020000) /\* Base @ of Sector 5, 128 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_6 ((uint32\_t)0x08040000) /\* Base @ of Sector 6, 128 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_7 ((uint32\_t)0x08060000) /\* Base @ of Sector 7, 128 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_8 ((uint32\_t)0x08080000) /\* Base @ of Sector 8, 128 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_9 ((uint32\_t)0x080A0000) /\* Base @ of Sector 9, 128 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_10 ((uint32\_t)0x080C0000) /\* Base @ of Sector 10, 128 Kbyte \*/

#define ADDR\_FLASH\_SECTOR\_11 ((uint32\_t)0x080E0000) /\* Base @ of Sector 11, 128 Kbyte \*/

# wifi 8266透传工程

## wifi 8266透传工程串口

#### AT端口定义串口号

这里表示定义哪一个串口，那个串口就可以使用AT命令，

这个工程支持AT指令集，也支持RTT的控制台命令集。

|  |
| --- |
| #ifdef WIFI\_SERIAL /\*透传节点--串口2\*/ 连接到通讯串口，当做透传用  //串口IO  #define AT\_TX\_CLK RCC\_APB2Periph\_GPIOA  #define AT\_TX\_PORT GPIOA  #define AT\_TX\_PIN GPIO\_Pin\_2  #define AT\_RX\_CLK RCC\_APB2Periph\_GPIOA  #define AT\_RX\_PORT GPIOA  #define AT\_RX\_PIN GPIO\_Pin\_3  //串口设备  #define AT\_UART USART2  #define AT\_UART\_CLK RCC\_APB1Periph\_USART2  #define AT\_UART\_CLKInit RCC\_APB1PeriphClockCmd  #define AT\_UART\_IRQn USART2\_IRQn  #define AT\_UART\_ISR USART2\_IRQHandler  //串口DMA配置  #define AT\_DMA\_CLK RCC\_AHBPeriph\_DMA1  #define AT\_DMA\_CHANNEL DMA1\_Channel6  #else /\*终端节点--串口3\*/  //串口IO  #define AT\_TX\_CLK RCC\_APB2Periph\_GPIOB  #define AT\_TX\_PORT GPIOB  #define AT\_TX\_PIN GPIO\_Pin\_10  #define AT\_RX\_CLK RCC\_APB2Periph\_GPIOB  #define AT\_RX\_PORT GPIOB  #define AT\_RX\_PIN GPIO\_Pin\_11  //串口设备  #define AT\_UART USART3  #define AT\_UART\_CLK RCC\_APB1Periph\_USART3  #define AT\_UART\_CLKInit RCC\_APB1PeriphClockCmd  #define AT\_UART\_IRQn USART3\_IRQn  #define AT\_UART\_ISR USART3\_IRQHandler  //串口DMA配置  #define AT\_DMA\_CLK RCC\_AHBPeriph\_DMA1  #define AT\_DMA\_CHANNEL DMA1\_Channel3  #endif |

#### 如果配置了WIFI\_SERIAL和RT\_USING\_UART3

**串口2使用的AT命令集 就是通讯串口**

**串口3使用RTT框架，能够使用finish控制台命令集**

|  |
| --- |
|  |

|  |
| --- |
|  |

## wifi 8266透传工程的配置过程

boot程序 command.h

typedef struct user\_data {

uint16\_t boot\_mode;

uint16\_t app\_en;

uint32\_t flag;

char mac[40]; //Ä£¿émac

char wifi\_ssid[40]; //wifi ssid

char wifi\_key[20]; //wifi key

char zhiyun\_id[40]; //id

char zhiyun\_key[120]; //ÖÇÔÆkey

char ip[38]; //ÖÇÔÆIP

uint16\_t port; //Á¬½ÓPort

uint16\_t sensor\_type; // sensor\_nodeÀàÐÍ

}user\_data\_t;

和app保持一致

APP程序 rf\_if\_app.h

typedef struct {

uint16\_t boot\_mode;

uint16\_t app\_en;

uint32\_t flag;

char mac[40]; //模块mac

char wifi\_ssid[40]; //wifi ssid

char wifi\_key[20]; //wifi key

char zhiyun\_id[40]; //智云id

char zhiyun\_key[120]; //智云key

char ip[38]; //连接IP

uint16\_t port; //连接Port

uint16\_t sensor\_type; // sensor\_node类型

} t\_rf\_info;

### flash\_if.h

如果设置了大容量的code，注意这里的参数地址不要覆盖了代码区域。

#define PARAMETER\_ADDRESS (uint32\_t)0x0801FC00 参数地址要和APP工程一致，

#define APPLICATION\_ADDRESS (uint32\_t)0x08004000

比如boot

## wifi 8266透传工程参数设置

RTT命令集 串口3

rf aid 9298378704

rf akey 7IqsPPQwpNGssgRNnbaOzvLsVAR14MGN

rf ssid zonesion

rf key 018164011650

rf save

AT命令集 串口2

AT+SSID="zonesion"

AT+KEY="018164011650"

AT+AID=9298378704

AT+AKEY="7IqsPPQwpNGssgRNnbaOzvLsVAR14MGN"

# 常见问题

### 1、参数覆盖了APP区导致硬件错误



由于原来的芯片这里使用的64K的芯片，0000-FFFF区域。0x10000=16\*16\*16\*16= 65536B=64KB

现在使用了更大的芯片，范围为0000-FFFF，128KB，而且程序就有50KB，很有可能APP区域被这个地址覆盖掉。

#define PARAMETER\_ADDRESS (uint32\_t)0x0801FE00 修改这里就OK

#define APPLICATION\_ADDRESS (uint32\_t)0x08004000