

***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、

# Ymodem

文件传输协议

YModem协议是由XModem协议演变而来的，每包数据可以达到1024字节，是一个非常高效的文件传输协议。

(Ymodem) Ymodem 协议。Ymodem是一种错误纠正协议。使用较大数据块的调制解调采用这种协议，以获得更高的工作效率。采用Ymodem协议的调制解调器以1024字节数的块发送数据。成功接收的不会被确认。有错误的块被确认（NAK），并重发。Ymodem类似于Xmodem-1K，不同之处是提供批处理模式（batch mode）。在批处理模式下，可以使用一个命令发送一些文件。Ymodem使用循环冗余码校验作为错误校验方式。 [1]

等同于Xmodem-1K 加批文件传送的一种文件传送协议 ( 也称 Ymodem Batch) 。它比标准 Xmodem 要快，而且在发送数据之前发送文件名。 Ymodem-G 无需无错误通道的认可或当调制解调器自纠错时就可传送，但是一旦出现错误传送就会取消。

## YMODEM协议的基本操作流程

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据包开始信号 | 发送序号 | 发送序号反码 | 数据区 | CRC高字节 | CRC低字节 |
| SOH/STX | 01 | FE | **…** | … | … |
| 1Byte | 1Byte | 1Byte | 128/1024Byte | 1Byte | 1Byte |

1开启是由接收方开启传输，接收方发送一个字符'C'，然后进入等待（SOH）状态，如果没有回应，就会超时退出。

2发送方开始时处于等待过程中，等待字符'C'。发送方收到'C'后，发送第一帧数据包，内容如下：

SOH 00 FF Foo.c NUL[123] CRC CRC （Foo.c为文件名，NUL[123]补0）

进入等待（ACK）状态。

3接收方收到第一帧数据包后，CRC校验满足，则发送ACK。

4发送方接收到ACK，又进入等待“文件传输开启”信号，即重新进入等待“C”的状态。

上面接收方只是收到了一个文件名，现在正式开启文件传输，Ymodem支持128字节和1024字节一个数据包。128字节以（SOH）开始，1024字节以（STX）开始。

5接收方又发出一个字符'C'，开始准备接收文件。进入等待“SOH”或者“STX”状态。

6发送方收到字符'C'后，开始发送第二帧，第二帧中的数据存放的是第一包数据。内容如下：

（SOH/STX）（01序号）（FE反码）（128/1024字节数据）（CRC校验），等待接收方“ACK”。

7接收方收到数据后，发送一个ACK，然后等待下一包数据传送完毕，继续ACK应答。直到所有数据传输完毕。…

8数据传输完毕后，发送方发EOT，第一次接收方以NAK应答，进行二次确认。发送方收到NAK后，重发EOT，接收方第二次收到结束符，就以ACK应答。最后接收方再发送一个字符'C'开启另一次传输，发送方在没有第二个文件要传输的情况下，发送如下数据：SOH 00 FF 00~00(共128个) CRCH CRCL，接收方应答ACK后，正式结束数据传输。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# OTA、bootloader基础知识点

## 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。

## bootloader(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。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# STM32实现OTA的流程简介

## 参考链接：

<https://zhuanlan.zhihu.com/p/73998216>

硬核高能！解析STM32 ota固件升级的全过程！

<https://www.freesion.com/article/1838925781/>

STM32在线升级OTA，看这一篇就够啦~（PS：不知道为什么要用两个分区来进行更新，不是理解的工厂分区）

<https://blog.csdn.net/qq_42860989/article/details/121699397>

STM32 OTA的实现方法和原理

<https://blog.csdn.net/yinhanxue/article/details/124332684>

手把手教你基于STM32的BootLoader的OTA远程升级（ONENET升级方案）

RT-Thread:

<https://blog.csdn.net/Jerry_Han0/article/details/114154644>

基于STM32F4，实现RT-Thread的串口OTA（Ymodem\_ota的方式）

<https://blog.csdn.net/cc_maoshu/article/details/128983694>

【RT-Thread】使用RT-Thread Studio 配置BootLoader及App实现OTA功能

<https://www.cnblogs.com/wt88/p/12779442.html>

RT-Thread—STM32—在线升级(Ymodem\_OTA、HTTP\_OTA)

<https://blog.csdn.net/weixin_44862509/article/details/127122188>

代号OneNET02 STM32+ESP8266 WIFI系列-第2节-STM32单片机通过ESP8266连接WIFI访问OneNET OTA服务器实现SOTA远程程序升级

<https://zhuanlan.zhihu.com/p/593727402>

国内 3 大物联网设备 OTA 服务选型指南

## OTA简介：

OTA的全称是Over-The-Air，在嵌入式系统中指对某个嵌入式系统的软件在线升级。就MCU系统来说，是针对整个MCU的Firmware进行在线更新，所以也称之为FOTA。也有一种叫法是DFU，即Device firmware upgrade。都是相同的意思，这里统一使用OTA表达。

通用MCU一般是没有无线功能的，所以对MCU的OTA一般用于双芯片实现的系统，无线芯片接收MCU的ota文件，通过串口或USB传给MCU，实现MCU的ota功能。

OTA与其他本地升级的区别就是：获取数据的方式不同。

1、比如串口升级，就是通过上位机传输到MCU串口上的数据；

2、SD卡升级，就是通过读取SD卡，把程序通过SPI传输到MCU上；

3、而OTA升级，就是通过带无线传输的模块，把程序传输到MCU上。例如：蓝牙、Wifi、GSM等等。不过大部分的无线模块，通过串口把数据传输到MCU上的，只是服务端不再是PC端了，而是网络服务器。

SOTA

应用升级，主要升级运行在操作系统之上的软件应用程序，需要软件架构及操作系统支持，比如Linux和Android等，升级过程不需要系统重启

FOTA

固件升级，升级主程序固件，一般需要系统重启进入

Bootloader中重新刷写应用固件到系统存储区

## 方案

1、OneNet移动云OTA升级基本方案

## OneNet移动云OTA升级基本方案

### 参考链接：

<https://www.cnblogs.com/mickey-double/p/15151977.html>

STM32 + ESP8266测试OneNet SOTA（头地址）

<https://open.iot.10086.cn/bbs/forum.php?mod=viewthread&tid=36467&extra=page%3D1%26filter%3Dtypeid%26typeid%3D23>

FOAT、SOTA、通用OTA讲解-新增基于stm32的bootloader - 玩转OneNET - OneNET设备云论坛 (10086.cn)

（含代码资料）

官方API介绍地址：

<https://open.iot.10086.cn/doc/ota/book/develop/auth.html>

API鉴权\_开发者文档\_OneNET (10086.cn)

<https://open.iot.10086.cn/doc/ota/book/manual/ota_develop_document.html#2>

升级操作流程\_开发者文档\_OneNET (10086.cn)

<https://open.iot.10086.cn/doc/ota/book/example/api_debugging_assistant.html>

API调试-网络调试助手\_开发者文档\_OneNET (10086.cn)

### 远程升级OTA （OneNET官方资料）

<https://open.iot.10086.cn/doc/ota/#%E8%BF%9C%E7%A8%8B%E5%8D%87%E7%BA%A7ota>

### FOTA、SOTA、通用OTA

1、FOTA（Firmware Update Over-The-Air ）主要用来实现解决物联网设备的软件远程修复BUG和系统更新的技术能力，是用于给设备的模组进行升级，是模组和OneNET平台之间进行数据交互，和mcu没有直接关系，具体过程请参考附件《OneNET平台FOTA升级操作步骤》。

2、SOTA（Software-Over-The-Air ）主要用来实现解决物联网设备的应用软件远程修复BUG和系统更新的技术能力，赋能与NB模组，NB模组和OneNET通信，下载固件包的过程为模组通过+MIPLWRITE:........指令给mcu返回固件包，需要用户操作mcu实现流程，具体过程请参考附件《M5312 M5311 M5310-A S04 nenet\_sota参考手册》。

3、通用OTA，针对于可以通过http方式get数据的设备，用户上传固件后，设备通过http方式去get固件，具体过程请参考附件《通用OTA设备侧调用API使用说明V1.0》。

### 硬件选择

MCU我这里选用的是STM32F030F4P6的芯片，16K的Flash，应该是ST产品中Flash空间比较小的一种，为的就是体现一下小容量的单片机也可以进行OTA升级。

1、无线模块我使用的是***ESP-8266***，WIfi传输方式，应该也是比较大众化的一款模组。（TTL串口连接MCU）

2、最好有个LED灯，可以明显的看出是否升级成功。

### 网络服务器的选择

网络服务器多种多样，常用的有阿里云、百度云、腾讯云、移动云等等，有条件的，还可以使用自己的服务器。总之需要实现：网络服务器可以与我们的无线模块进行大数据通信。

1、我这里选用的是OneNet移动云（OTA服务之前是免费，现在是前100个设备免费，之后每增加一个设备1元钱永久），我感觉OneNet相对于阿里云较为简单，没有阿里云那么繁琐，

2、不过***阿里云还是比OneNet更专业一点***（个人见解），其他的没有用过，大家都可以去试试。

### OTA升级流程

OneNet的OTA升级流程主要为6步：

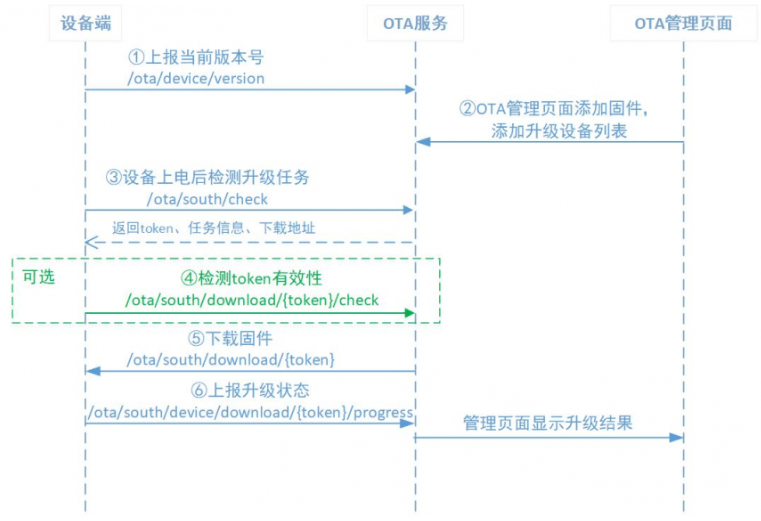
1. 上报版本号---客户端（MCU）上报当前的一个版本号

2. 检测升级任务---检查服务器是否有待升级的版本

3. 检测Token有效性---检查Token密钥，可省略

4. 下载固件---应用程序传输

5. 上报升级状态---上报服务端升级是否成功，不成功有对应的响应码



### OneNet服务端配置

<https://blog.csdn.net/yinhanxue/article/details/124332684>

### 客户端(MCU)API访问服务端进行OTA升级

无线模组用的是ESP8266，由于OneNet的OTA服务用的是HTTP协议，但是ESP8266没有HTTP协议，所以我使用TCP协议，封装成HTTP的报文格式。

#### 1.ESP8266初始化；

连接Wifi，AP\_SSID，AP\_PASS是WiFi的账号和密码；SERVER\_IP和SERVER\_PORT是OneNet的Ip和端口号。

|  |
| --- |
| # defineSERVER\_IP "183.230.40.50"  # defineSERVER\_PORT 80  uint8\_t pro = 0;  uint8\_tESP8266\_Init( void)  {  switch(pro)  {  case0:  //printf("+++");  Uart2\_Send( "+++");  Delay\_S( 2);  if(ESP8266\_SoftReset( 50) == 0)  pro = 1;  break;  case1:  if(ESP8266\_AT\_Send( "ATE0\r\n", 10) == 0)  pro = 2;  break;  case2: if(ESP8266\_AT\_Send( "AT+CWMODE=1\r\n", 50) == 0)  //设置8266为STA模式  pro = 3;  break;  case3:  if(ESP8266\_ConnectionAP(AP\_SSID,AP\_PASS, 200) == 0)  //8266连接AP  pro = 4;  break;  case4:  if(ESP8266\_AT\_Send( "AT+CIPMODE=1\r\n", 50) == 0) //8266开启透传模式  pro = 5;  break;  case5:  if(ESP8266\_Connect\_Server(SERVER\_IP,SERVER\_PORT, 50) == 0) //8266连接TCP服务器  {  pro = 0; //  USART1\_Clear; //清除串口数据  return 1;  }  break;  }  return 0;  } |

#### 2.上报版本号；

dev\_id是设备ID，authorization是鉴权参数，ver要上报的版本号，timeout发送超时时间。



|  |
| --- |
| //上报版本号  //上报版本号  uint8\_t Report\_Version( char\* dev\_id, char\* authorization, char\*ver, uint16\_t timeout)  {  uint16\_t time= 0;  char send\_buf[ 296];  USART1\_Clear; //清除串口数据  snprintf(send\_buf, sizeof(send\_buf),  "POST /ota/device/version?dev\_id=%s HTTP/1.1\r\n""Authorization:%s\r\n""Host:ota.heclouds.com\r\n""Content-Type:application/json\r\n""Content-Length:%d\r\n\r\n""{\"s\_version\":\"%s\"}",  dev\_id, authorization, strlen(ver) + 16, ver);  Uart2\_Send(send\_buf);  while(time<timeout)  {  if( strstr( ( const char\*)usart\_info.buf , ( const char\*) "\"errno\":0"))  break;  Delay\_Ms( 100);  time++;  }  if(time>=timeout)  return 1;  else return 0;  } |

|  |
| --- |
|  |

#### 3.检查升级任务；

dev\_id是设备ID，authorization是鉴权参数，cur\_version是当前的版本号，timeout发送超时时间



|  |
| --- |
| //检查升级任务  uint8\_t Detect\_Task( char\* dev\_id, char\* cur\_version, char\* authorization, uint16\_t timeout)  {  uint16\_t time= 0;  char send\_buf[ 280];  USART1\_Clear; //清除串口数据  snprintf(send\_buf, sizeof(send\_buf), "GET /ota/south/check?""dev\_id=%s&manuf=100&model=10001&type=2&version=%s&cdn=false HTTP/1.1\r\n""Authorization:%s\r\n""Host:ota.heclouds.com\r\n\r\n", dev\_id, cur\_version,authorization);  Uart2\_Send(send\_buf);  while(time<timeout)  {  if( strstr( ( const char\*)usart\_info.buf , ( const char\*) "\"errno\":0"))  break;  Delay\_Ms( 100);  time++;  }  if(time>=timeout)  return 1;  else return 0;  } |

#### 4.下载资源

(我省略了"检查token有效"步骤)；ctoken是上一步“检查升级任务”返回的Token，这个每次请求都不一样，所以注意要记录；size：平台返回的固件大小(字节)；bytes\_range：分片大小(字节)



|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\* 函数名称： OTA\_Download\_Range  \*\* 函数功能： 分片下载固件  \*\* 入口参数： token：平台返回的Token\* size：平台返回的固件大小(字节)\* bytes\_range：分片大小(字节)  \*\* 返回参数： 0-成功 其他-失败  \*\* 说明： \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  uint8\_t Download\_Task( char\*ctoken, unsigned int size, const unsigned short bytes\_range, uint16\_t timeout)  {  MD5\_CTX md5\_ctx; //MD5相关变量  unsigned char md5\_t[ 16];  char md5\_t1[ 16];  char md5\_result[ 40];  uint16\_t time= 0;  char\*data\_ptr = NULL;  char send\_buf[ 256];  unsigned char flash\_buf[OTA\_BUFFER\_SIZE]; //flash读写缓存  unsigned int bytes = 0;  MD5\_Init(&md5\_ctx);  Flash\_cashu;  while(bytes < size)  {  time = 0;  memset(send\_buf, 0, sizeof(send\_buf));  USART1\_Clear; //清除串口数据  snprintf(send\_buf, sizeof(send\_buf), "GET /ota/south/download/""%s HTTP/1.1\r\n""Range:bytes=%d-%d\r\n""Host:ota.heclouds.com\r\n\r\n", ctoken, bytes, bytes + bytes\_range - 1);  Uart2\_Send(send\_buf);  //----------------------------------------------------等待数据---------------------------------------------------------------------  while(time < 30)  {  if(usart\_info.buf[ 0] != 0)  break;  Delay\_Ms( 100);  time++;  }  if(time <= 29)  {  Delay\_Ms( 500);  //----------------------------------------------------跳过HTTP报文头、找到固件数据--------------------------------------------------  data\_ptr = strstr( ( const char\*)usart\_info.buf, "Range");  data\_ptr = strstr(data\_ptr, "\r\n");  data\_ptr += 4;  //----------------------------------------------------将固件数据写入缓存和闪存-----------------------------------------------------  if(data\_ptr != NULL)  {  if((size - bytes) >= OTA\_BUFFER\_SIZE)  {  memcpy(flash\_buf + (bytes % OTA\_BUFFER\_SIZE), data\_ptr, bytes\_range);  STMFLASH\_Write\_NoCheck(FLASH\_APP1\_ADDR + bytes,( uint16\_t\*)flash\_buf,OTA\_BUFFER\_SIZE / 2);  bytes = bytes + OTA\_BUFFER\_SIZE;  MD5\_Update(&md5\_ctx, ( unsigned char\*)data\_ptr, bytes\_range);  }  else  {  memcpy(flash\_buf + (bytes % OTA\_BUFFER\_SIZE), data\_ptr, size - bytes);  STMFLASH\_Write\_NoCheck(FLASH\_APP1\_ADDR + bytes , ( uint16\_t\*)flash\_buf , (size % OTA\_BUFFER\_SIZE) / 2);  MD5\_Update(&md5\_ctx, ( unsigned char\*)data\_ptr, size - bytes);  bytes = size;  }  }  }  }  //----------------------------------------------------MD校验比对------------------------------------------------------------------  memset(md5\_result, 0, sizeof(md5\_result));  MD5\_Final(&md5\_ctx, md5\_t);  for( int i = 0; i < 16; i++)  {  if( md5\_t[i] <= 0x0f)  sprintf(md5\_t1, "0%x", md5\_t[i]);  else  sprintf(md5\_t1, "%x", md5\_t[i]);  strcat(md5\_result, md5\_t1);  }  if( strcmp(md5\_result, ota\_info.md5) == 0)  return 0;  else  return 1;  } |

#### 5.上报升级状态；

这一步由于时间问题，我也省略了，总之程序已经下载到MCU上了，只是没有通知服务器而已，大家最好还是加上这一步。



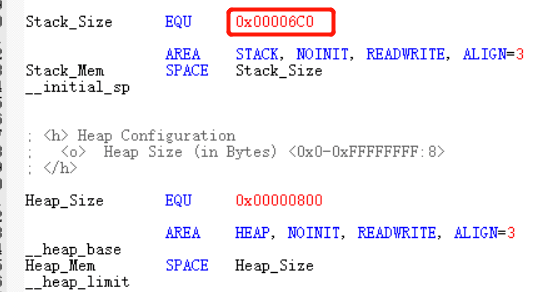


#### 6.main函数循环；

|  |
| --- |
| int mian()  {  char rrr;  char dev\_id[] = { "640600857"};  char Authorization[] = { "version=2018-10-31&res=products%2F378414&et=1735660800&method=sha1&sign=9EgY%2Bk4r%2BlvCooIGf1ghtQFC0%2Bc%3D"};  char Version[] = { "V10"};  while(1)  {  switch(pro)  {  case1: //上报版本  if(Report\_Version(dev\_id,Authorization,Version, 10) == 0)  pro++;  break;  case2: //检查任务  if(Detect\_Task(dev\_id,Version,Authorization, 50) == 0)  pro++;  break;  case3: //接收token、size、md5信息  rrr = json\_get\_value(( char\*)usart\_info.buf, "token",ota\_info.token);  rrr = json\_get\_value(( char\*)usart\_info.buf, "size",ota\_info.csize);  rrr = json\_get\_value(( char\*)usart\_info.buf, "md5",ota\_info.md5);  ota\_info.size = atoi(ota\_info.csize);  pro++;  break;  case4: //进行下载  res = Download\_Task(ota\_info.token,ota\_info.size,OTA\_BUFFER\_SIZE, 10);  if(res == 0) //校验成功  {pro++;}  else if(res == 1) //校验失败  {pro = 1; }  break;  case5: //Flash写入升级完成的标志位  USART1\_Clear;  STMFLASH\_Unlock;  STMFLASH\_WriteHalfWord(FLASH\_APP1\_ADDR - 0x64, 0xFF02); //写入数据  STMFLASH\_Lock;  pro++;  break;  case6: //复位或者跳转到  APPSys\_Soft\_Reset;  //iap\_load\_app(FLASH\_APP1\_ADDR);  break;  }  }  } |

### 注意事项

由于用的是STM32F030F4P6，RAM也非常小，所以局部变量和全局变量的数组不要超过4K，堆栈大小有改动。当前用内存管理的话就不用了。



3.OTA校验用的是MD5，需要把MD5的算法移植一下。

4.别的想不到了，太长时间了。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# STM32 F407 ZGT6 探索者实现OTA过程

## 参考示例

<http://www.xbhp.cn/news/49223.html>

<https://github.com/loogg/agile_upgrade_mcu_demos>

基于RT-Thread完整版搭建的极简Bootloader

<https://blog.csdn.net/yangxiangzhan/article/details/86555117?spm=1001.2014.3001.5501>

适用stm32的命令行解释器shell

## 名词解释

此处将对艾拉比相关的术语进行定义与解析。如果在平台使用或开发过程中遇到一些术语不太理解的可以参阅此章节。

OTA

定义：OTA英文全称是Over-the-Air Technology，即空中下载技术的意思。目前，OTA升级是Android、RTOS、Linux等系统提供的标准软件升级方式。它功能强大，可以无损失升级系统，主要通过网络例如WiFi、4G、3G、2G、NB-IoT等自动下载OTA升级包、自动升级，也支持通过下载OTA升级包到SD卡进行升级。

FOTA

定义：FOTA(Firmware Over-the-Air)移动终端的空中下载软件升级，指通过云端升级技术，为具有连网功能的设备提供固件升级服务，用户使用网络以按需、易扩展的方式获取智能终端系统升级包，并通过FOTA进行云端升级，完成系统修复和优化。

解析：FOTA的本质是固件升级，包括驱动、系统、功能、应用等的升级，和硬件没有直接关系。使用的终端范围很广，基本可以为市场上所有的终端提供升级服务，无论对于电信运营商还是终端设备制造商，通过集群应用、网络技术和分布式服务端，能够在同一时间内处理大量用户的终端升级需求。FOTA和OS的关系较密切，不同的OS版本，需要开发不同的FOTA适配版本，同时通过FOTA模块下载的系统升级包，也要和OS进行密切的匹配，不但要进行硬件驱动的调试，还要进行版本的兼容测试，但这样的升级包一般是由终端厂商提供，FOTA更多的是保证升级包的下载以及终端设备的安装，在智能时代，FOTA云升级将成为智能终端的标准配置。

Recovery

定义：Recovery模式指的是一种可以对系统内部数据或系统进行修改的模式。在这个模式下我们可以刷入新的系统程序，或者对已有的系统进行备份或升级，也可以在此恢复出厂设置。

OTA Package

定义：升级包，一种具有特定格式的文件集。

差分升级

定义：差分升级方式需要对源版本和目标版本进行差分操作，生成两个版本的patch差分进行升级。其优势在于生成的差分包可以很小甚至只有几KB，非常有利于通过网络进行下载。缺点是在升级过程中比较容易出错，流程控制上更严谨。

整包升级

定义：整包升级方式不需要源版本包，直接使用目标版本包进行升级，一般用于整个固件升级。其优势是安全，不容易出错，不需要依赖特殊的固件。缺点是整包的大小比较接近整个固件的镜像，下载更新包将是一个很漫长的过程。哪怕是版本功能上的一个小小改动，都需要下载整个版本包进行升级。

差分算法

定义：差分算法是一个差量更新算法，它在服务器运行响应算法产出patch包，在客户端运行反向补丁算法，将旧文件和patch包合成新文件。

UA

定义：艾拉比差分升级引擎，可以实现零件的差分升级，UA会以SDK的形式交付给客户。

UB

艾拉比推出的标准化Bootloader产品，具备引导系统启动并支持差分升级，提供可视化的图形配置工具来制作，无需开发，烧录即用。

ELB

艾拉比推出的轻量化Bootloader程序，具备与设备中其他节点进行通讯及数据传输，是艾拉比推出的产品级OTA方案-OpenFOTA方案的标准化产品，旨在借助一个产品体系内其他节点的资源及升级能力来完成自身的版本更新。

DPC刷写升级

基于总线支持艾拉比艾拉比标准协议/UDS协议的诊断刷写升级。

Product ID

定义：产品标识码，用户通过艾拉比平台创建产品及不同产品配置后，自动生成的一个字符串，在艾拉比的数据库中是一个唯一的号码，开发者将Product ID写入设备后，艾拉比通过此标识码对设备进行识别并完成注册。

Product Secret

定义：产品密钥，在生成Product ID的时候云端会对应生成一个Product Secret，用于确认用户的身份和使用权限。

## 升级方案

### 方案1

bootloader 上集成 和 8266 模块驱动，可以实现连接服务器并下载数据包到flash中

好处：主要bootloader启动，就可以进行升级

坏处：bootloader 程序比较大

### 方案2

bootloader 程序支持IAP升级。

进行app系统之后才能够使用wifi模块进行OTA更新，由于阿里云提供的SDK是基于OS系统做的。bootloader程序一般是裸机程序

艾拉比也是使用OS系统做的SDK

## 硬件容量

程序存储容量：1MB(1Mx8)

程序存储器类型：闪存

RAM大小：192Kx8

## 硬件使用

* 串口1 调试串口

#define USARTx\_RX\_DMA\_STREAM DMA2\_Stream5

#define USARTx\_RX\_DMA\_CHANNEL DMA\_Channel\_4

#define USARTx\_RINGBUF\_SIZE 256

PA9 PA10

* 串口3

USART3\_TX PB10 PB11

## bootloader

### 基础工程选择

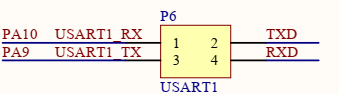
就选择ATK的示例工程

### 硬件

串口3 8266/32的AT通信端口



串口1 调试串口 控制台串口



### 调试端口控制台

argtable3

这里控制台是参考的esp32

<https://github.com/argtable/argtable3>

下载这个版本

|  |
| --- |
| Administrator@XTTD-2023ONVFGT MINGW64 ~/Desktop/argtable3 (master)  $ git checkout  HEAD v3.0.1 v3.1.2.bb37058 v3.2.1.52f24e5  master v3.0.2 v3.1.3.09d011d v3.2.2.f25c624  origin/HEAD v3.0.3 v3.1.4.336b599  origin/master v3.1.0 v3.1.5.1c1bb23  v3.0.0 v3.1.1.432a160 v3.2.0.7402e6e  Administrator@XTTD-2023ONVFGT MINGW64 ~/Desktop/argtable3 (master)  $ git checkout v3.0.0 |

这个代码量太大了，不太适合

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

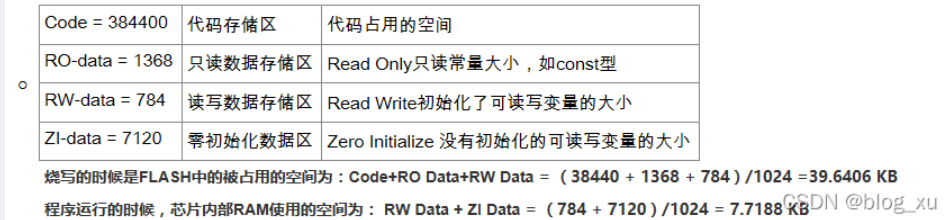
|  |
| --- |
|  |

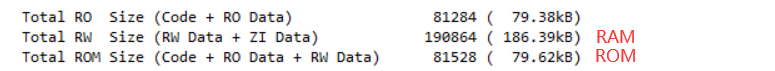
|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

### 目标代码占用ROM和RAM







STM32F407ZGT6的存储大小：

RAM: 192K

ROM: 1024K

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

### 适用stm32的命令行解释器shell

#### 记录一下这个shell的应用说明

源码的地址：

<https://gitee.com/somebug/atomlib>

shell 层与硬件无关，有无操作系统都可以使用，适用于stm32等32位小端的单片机，支持历史纪录功能，tab 键补全命令，提供命令行参数解释函数，可以响应tab、backspace、上下左右编辑等功能，并提供一个简易版的vi 文本编辑功能。支持多交互接口，各个接口间数据流互不影响。

其中此文件夹中的文件都与硬件平台无关，但可能与编译平台有关。此库编译版本要在**C99以上**，或用GCC编译。根据不同的硬件平台分别写了不同的控制台demo， F1 文件夹的是STM32F1 相关的控制台， F4 则是 F4xx 系列的控制台，但是L1平台的我并没有调通，手上也没有这个板子，所以先放着。

* 文件说明：

***avltree.c*** :平衡二叉树相关实现代码，在注册命令较多的时候可以启用平衡二叉树进行索引匹配。

***getopt.c*** :有些编译环境，如 KEIL5 中，没有函数getopt() ,这是其源码，是我在网上找的。要是用 gcc 相关的编译平台可以 #include <unistd.h> 找到 getopt() 函数支持。

***heaplib.c*** : 内存管理，其实这个是我原封不动从freertos 的 heap\_4.c 拷过来的，在heaplib.h 提供了部分宏支持，可以脱离操作系统使用。

***shell.c***:命令行相关，支持 table 键补全，支持上下左右箭头响应，提供参数解析，历史纪录。支持多个交互，如串口，telnet，或者usb，可各自建立交互。

***tasklib.c***:协程控制器。有需要的话使用协程可以简化代码的编写。我把它模拟成一个操作系统。

***ustdio.c***:提供 printk 函数，重定义 printf 函数

***vim.c***:这是我仿照 linux 的 vi 写的一个建议的文本编辑器，依赖 shell ，可以实现简易的文本编辑。

F1/F4/L1:不同硬件平台的相关串口控制台实现，提供串口在线升级功能。

***LittleFS***: LittleFS 是一个用于 spi flash 的文件系统，先放这。

#### 基本使用（keil5工程）

使用这个库的基本功能只需要把 shell.c 、shell.h 、ustdio.c 、ustdio.h 和 kernel.h 这几个文件包含进文件工程里面即可。系统的使用可以大致分为以下几个步骤：

0.初始化硬件部分。

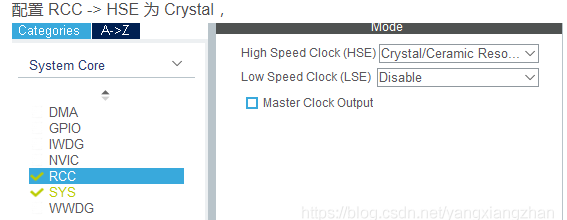
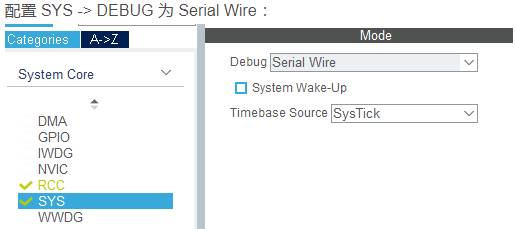
1.编写硬件对应的void puts(char \* buf , uint16\_t len) 发送函数。

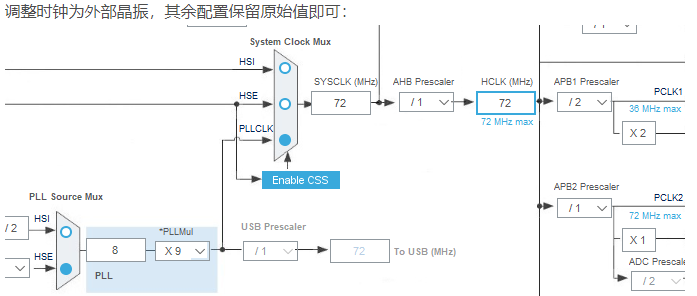
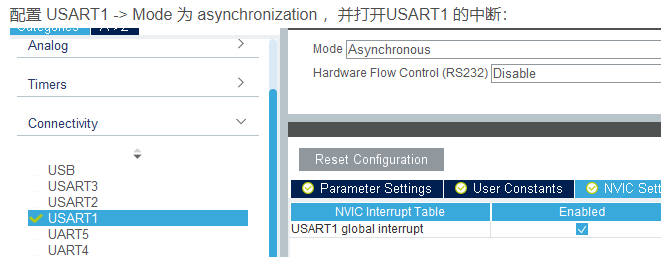
2.shell\_init(sign,puts)初始化输入标志和默认输出。

3.新建一个交互shellx 并初始化shell\_input\_init(&shellx,puts);

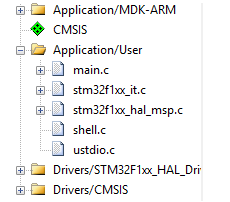
4.接收到一包数据后，调用shell\_input(&shellx,buf,len)

以下用stm32cubemx新建串口工程实际说明。Mcu选择为STM32F103VET6；





最后生成MDK-KEIL5 工程。把 shell.c 、ustdio.c 复制到工程文件夹 src 中，并加入文件， containerof.h、shell.h、ustdio.h复制到 inc 文件夹中。



#### 适配shell命令行（keil5工程）

##### 移植文件

<https://gitee.com/somebug/atomlib>

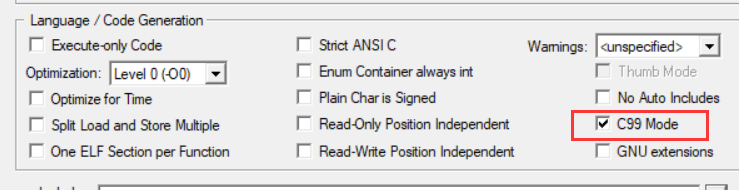
shell.c 、ustdio.c

shell.h、ustdio.h

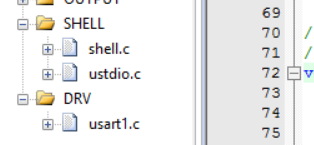
containerof.h 这个文件指针转换使用

getopt.c和getopt.h 如果需要命令选项的话。就需要这个（这个容易报错，没有这个无法使用选项，但是可以获取命令参数）

##### 启动C99



##### 适配串口（标准库版本）



由于是在正点原子工程上适配的，usart.c改名为usart1，移动到DRV中

###### 删除原来的如下函数

|  |
| --- |
| ..\OBJ\atk\_bootloader.axf: Error: L6200E: Symbol \_\_stdout multiply defined (by usart1.o and ustdio.o).  ..\OBJ\atk\_bootloader.axf: Error: L6200E: Symbol \_sys\_exit multiply defined (by usart1.o and ustdio.o).  ..\OBJ\atk\_bootloader.axf: Error: L6200E: Symbol fputc multiply defined (by usart1.o and ustdio.o).  Not enough information to list image symbols.  Not enough information to list load addresses in the image map.  Finished: 2 information, 0 warning and 3 error messages.  "..\OBJ\atk\_bootloader.axf" - 3 Error(s), 0 Warning(s). |

|  |
| --- |
| //加入以下代码,支持printf函数,而不需要选择use MicroLIB  #if 1  #pragma import(\_\_use\_no\_semihosting)  //标准库需要的支持函数  struct \_\_FILE  {  int handle;  };  FILE \_\_stdout;  //定义\_sys\_exit()以避免使用半主机模式  void \_sys\_exit(int x)  {  x = x;  }  //重定义fputc函数  int fputc(int ch, FILE \*f)  {  while((USART1->SR&0X40)==0);//循环发送,直到发送完毕  USART1->DR = (u8) ch;  return ch;  }  #endif |

###### shell.c教我们怎么适配

|  |
| --- |
| \* @file shell.c  \* @author 古么宁  \* @brief shell 命令解释器，支持 TAB 键命令补全，上下左右箭头 ，BACKSPACE回删  \* @note  \* <pre>  \* 使用步骤:  \* 0.初始化硬件部分。  \* 1.编写硬件对应的void puts(char \* buf , uint16\_t len) 发送函数。  \* 2.shell\_init(sign,puts) 初始化输入标志和默认输出。  \* 3.新建一个 shellinput\_t shellx , 初始化输出 shell\_input\_init(&shellx,puts,...);  \* 4.接收到一包数据后，调用 shell\_input(shellx,buf,len)  \* \*. 需要注册命令则调用宏 shell\_register\_command 进行注册。  \* \*.. shell\_register\_confirm() 可注册带选项命令([Y/N]选项)  \* </pre> |

###### shell\_init和shell\_input\_init

|  |
| --- |
| #include "stm32f4xx.h"  #include "stm32f4xx.h"  #include "delay.h"  #include "usart1.h"  static void hardware\_init()  {  uart1\_init(115200);  delay\_init(84);  // 初始化 shell 控制台  shell\_init("shell >" ,usart1\_puts); // 初始化 控制台输出  shell\_input\_init(&shell\_1,usart1\_puts); // 初始化 交互  }  int main(void)  {  hardware\_init();  while(1){  delay\_ms(500);  }  } |

###### usart1.c(暂时没有适配DMA接收)

|  |
| --- |
| #include "sys.h"  #include "usart1.h"  shellinput\_t shell\_1;  // 串口1的发送函数  // 发送字符串  void usart1\_puts(const char \* strbuf, unsigned short len)  {  while(len--)  {  USART\_ClearFlag(USART1, USART\_FLAG\_TC);  USART1->DR = \*strbuf;  while (USART\_GetFlagStatus(USART1, USART\_FLAG\_TC) == RESET);  strbuf++ ;  }  }  //串口1中断服务程序  void USART1\_IRQHandler(void)  {  char Res;  if(USART\_GetITStatus(USART1, USART\_IT\_RXNE) != RESET) //接收中断  {  Res =USART\_ReceiveData(USART1); //读取接收到的数据  shell\_input(&shell\_1,&Res,1);  }  } |

###### 效果

|  |
| --- |
| \_\_\_\_\_ \_\_  / \_\_\_\_\ /\ \  /\ \\_\_\_/ \_\_\_\_ \_\_\_\_ \_\_\_\_ \_\\_\ \ \_\_\_\_  \ \ \ / \_\_ \/ \_\_ \/ \_\_\_\/ \_\_ \ \ / \_\_ \  \ \ \\_\_\_L\ \L\ \ \/\ \\_\_\_\_ \ \L\ \ \\_L\ \_\_\_L  \ \\_\_\_\_\_\_\\_\_\_\_/\\_\ \\_\\_\_\_\_/\\_\_\_\_/\\_\_\_\_\\_\_\_\_/  \/\_\_\_\_\_\_/\_\_\_/\/\_/\/\_/\_\_\_/\/\_\_\_/\/\_\_\_\_/\_\_\_/  COPYRIGHT(c):GoodMorning 2019/06  shell >  shell >cmd-list  (c)------  clear  cmd-list  (d)------  debug-info  (s)------  shell-version  shell > |

###### 适配DMA

|  |
| --- |
| #include "sys.h"  #include "usart1.h"  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*user config\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  // 定义控制台调试串口的硬件  // 串口1 引脚配置  #define USARTx USART1  #define USARTx\_IRQn USART1\_IRQn  #define USARTx\_GPIO\_RCC RCC\_AHB1Periph\_GPIOA  #define USARTx\_RCC RCC\_APB2Periph\_USART1  #define USARTx\_TX\_GPIO GPIOA  #define USARTx\_TX\_PIN\_NUM GPIO\_PinSource9  #define USARTx\_TX\_PIN GPIO\_Pin\_9  #define USARTx\_RX\_GPIO GPIOA  #define USARTx\_RX\_PIN\_NUM GPIO\_PinSource10  #define USARTx\_RX\_PIN GPIO\_Pin\_10  // 串口1DMA\_RX配置  #define USARTx\_RX\_DMA\_STREAM DMA2\_Stream5  #define USARTx\_RX\_DMA\_CHANNEL DMA\_Channel\_4  #define USARTx\_RINGBUF\_SIZE 20  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*user config\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  // shell 终端使用变量  shellinput\_t shell\_1;  static char shell\_ringbuf[USARTx\_RINGBUF\_SIZE]={0};  static unsigned short Read\_Index;  static unsigned short Write\_Index;  static volatile char Receive\_flag; // 串口空闲中断接收标志，shell开始进行数据处理  //初始化IO 串口1  //bound:波特率  static void UART\_Init(u32 bound)  {  //GPIO端口设置  GPIO\_InitTypeDef GPIO\_InitStructure={0};  USART\_InitTypeDef USART\_InitStructure={0};  RCC\_AHB1PeriphClockCmd(USARTx\_GPIO\_RCC,ENABLE); //使能GPIOA时钟  RCC\_APB2PeriphClockCmd(USARTx\_RCC,ENABLE); //使能USART时钟  //串口1对应引脚复用映射  GPIO\_PinAFConfig(USARTx\_TX\_GPIO,USARTx\_TX\_PIN\_NUM,GPIO\_AF\_USART1); //GPIOA9复用为USART  GPIO\_PinAFConfig(USARTx\_RX\_GPIO,USARTx\_RX\_PIN\_NUM,GPIO\_AF\_USART1); //GPIOA10复用为USART  //USART 端口配置  GPIO\_InitStructure.GPIO\_Pin = USARTx\_TX\_PIN | USARTx\_RX\_PIN; //GPIOA9与GPIOA10  GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_AF;//复用功能  GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz; //速度50MHz  GPIO\_InitStructure.GPIO\_OType = GPIO\_OType\_PP; //推挽复用输出  GPIO\_InitStructure.GPIO\_PuPd = GPIO\_PuPd\_UP; //上拉  GPIO\_Init(USARTx\_TX\_GPIO,&GPIO\_InitStructure); //初始化PA9，PA10  //USART 初始化设置  USART\_InitStructure.USART\_BaudRate = bound;//波特率设置  USART\_InitStructure.USART\_WordLength = USART\_WordLength\_8b;//字长为8位数据格式  USART\_InitStructure.USART\_StopBits = USART\_StopBits\_1;//一个停止位  USART\_InitStructure.USART\_Parity = USART\_Parity\_No;//无奇偶校验位  USART\_InitStructure.USART\_HardwareFlowControl = USART\_HardwareFlowControl\_None;//无硬件数据流控制  USART\_InitStructure.USART\_Mode = USART\_Mode\_Rx | USART\_Mode\_Tx; //收发模式  USART\_Init(USARTx, &USART\_InitStructure); //初始化串口  USART\_Cmd(USARTx, ENABLE); //使能串口1  }  // 串口 DMA Rx 配置  static void UART\_DMA\_RxConfig(DMA\_Stream\_TypeDef \*DMA\_Streamx,u32 channel,u32 addr\_Peripherals,u32 addr\_Mem,u16 Length) //DMA的初始化  {  // NVIC\_InitTypeDef NVIC\_InitStructure={0};  DMA\_InitTypeDef UARTDMA\_InitStructure={0};  if((u32)DMA\_Streamx>(u32)DMA2)//DMA1?DMA2  RCC\_AHB1PeriphClockCmd(RCC\_AHB1Periph\_DMA2,ENABLE);//DMA2  else  RCC\_AHB1PeriphClockCmd(RCC\_AHB1Periph\_DMA1,ENABLE);//DMA1  DMA\_DeInit(DMA\_Streamx);  while (DMA\_GetCmdStatus(DMA\_Streamx) != DISABLE){}//等待DMA可配置  UARTDMA\_InitStructure.DMA\_Channel = channel; //通道  UARTDMA\_InitStructure.DMA\_PeripheralBaseAddr = ( uint32\_t)(addr\_Peripherals) ;//外设地址  UARTDMA\_InitStructure.DMA\_Memory0BaseAddr = ( uint32\_t)addr\_Mem; //缓存地址  UARTDMA\_InitStructure.DMA\_DIR = DMA\_DIR\_PeripheralToMemory; //方向-外设到内存  UARTDMA\_InitStructure.DMA\_BufferSize = Length; //传输长度  UARTDMA\_InitStructure.DMA\_PeripheralInc = DMA\_PeripheralInc\_Disable; //外设非增量模式  UARTDMA\_InitStructure.DMA\_MemoryInc = DMA\_MemoryInc\_Enable; //存储器增量模式  UARTDMA\_InitStructure.DMA\_PeripheralDataSize = DMA\_PeripheralDataSize\_Byte ; //外设数据长度:8位  UARTDMA\_InitStructure.DMA\_MemoryDataSize = DMA\_MemoryDataSize\_Byte ; //存储器数据长度:8位  UARTDMA\_InitStructure.DMA\_Mode = DMA\_Mode\_Circular ; //使用循环模式  UARTDMA\_InitStructure.DMA\_Priority = DMA\_Priority\_Medium; //中等优先级  UARTDMA\_InitStructure.DMA\_FIFOMode = DMA\_FIFOMode\_Disable;  UARTDMA\_InitStructure.DMA\_FIFOThreshold = DMA\_FIFOThreshold\_Full;  UARTDMA\_InitStructure.DMA\_MemoryBurst = DMA\_MemoryBurst\_Single ; //存储器突发单次传输  UARTDMA\_InitStructure.DMA\_PeripheralBurst = DMA\_PeripheralBurst\_Single; //外设突发单次传输  DMA\_Init(DMA\_Streamx, &UARTDMA\_InitStructure); //初始化DMA Stream  // 这里没有使用到DMA的接收中断  // NVIC\_InitStructure.NVIC\_IRQChannel = DMA\_USART\_RX\_IRQ ; //这里使用串口的空闲中断，所以这里就没有开启  // NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority = 1;  // NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 0;  // NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE;  // NVIC\_Init(&NVIC\_InitStructure);  // DMA\_ITConfig(DMA\_Streamx, DMA\_IT\_TC, ENABLE);//开启传输完成中断触发 //DMA中断使能  DMA\_Cmd(DMA\_Streamx, ENABLE); //DMA开启  }  // 串口1 DMA 初始化  static void UART\_DMA\_Init()  {  NVIC\_InitTypeDef NVIC\_InitStructure={0};  // DMA接收配置  UART\_DMA\_RxConfig(USARTx\_RX\_DMA\_STREAM, // DMA串口接收流号码  USARTx\_RX\_DMA\_CHANNEL, // DMA流对应的通道  (u32)&USARTx->DR, // 寄存器  (u32)shell\_ringbuf, // DMA接收区域  USARTx\_RINGBUF\_SIZE);  USART\_DMACmd(USARTx, USART\_DMAReq\_Rx , ENABLE); // 使能串口1的DMA接收  // 配置中断  USART\_ITConfig(USARTx, USART\_IT\_IDLE, ENABLE); // 开启空闲中断    NVIC\_InitStructure.NVIC\_IRQChannel = USARTx\_IRQn; // 串口1中断通道  NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority=1; // 抢占优先级  NVIC\_InitStructure.NVIC\_IRQChannelSubPriority =0; // 子优先级  NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE; // IRQ通道使能  NVIC\_Init(&NVIC\_InitStructure);  }  /\* 获取缓冲区中剩余未读取数据长度 \*/  static uint16\_t UART\_GetRemain(void) {  uint16\_t remain\_length; // 剩余数据长度  uint16\_t write\_index=Write\_Index; // 拷贝当前写索引位置  //获取剩余数据长度  if(write\_index >= Read\_Index) {  remain\_length = write\_index - Read\_Index;  } else {  remain\_length = USARTx\_RINGBUF\_SIZE - Read\_Index + write\_index; // 此时说明串口缓存区数据从头开始缓存  }  return remain\_length;  }  // shell 硬件初始化  void shell\_hw\_init(u32 bound)  {  UART\_Init(bound);  UART\_DMA\_Init();  }  // 串口1的发送函数  // 发送字符串  void usart\_puts(const char \* strbuf, unsigned short len)  {  while(len--)  {  USART\_ClearFlag(USARTx, USART\_FLAG\_TC);  USARTx->DR = \*strbuf;  while (USART\_GetFlagStatus(USARTx, USART\_FLAG\_TC) == RESET);  strbuf++ ;  }  }  // shell控制台获取输入数据  void shell\_hw\_input()  {  if(Receive\_flag){  unsigned short data\_len= UART\_GetRemain(); // 获取当前数据长度  shell\_input(&shell\_1, shell\_ringbuf + Read\_Index, data\_len);  Read\_Index = (Read\_Index+data\_len)% USARTx\_RINGBUF\_SIZE; // 下次读取数据的起始位置，防止超出缓存区最大索引    USARTx->CR1 &= ~USART\_CR1\_IDLEIE; // 临界段保护  Receive\_flag=0;  USARTx->CR1 |= USART\_CR1\_IDLEIE;  }  }  //串口1中断服务程序  void USART1\_IRQHandler(void)  {  if(USART\_GetITStatus( USARTx, USART\_IT\_IDLE ) == SET )  {  USART\_ReceiveData(USARTx);//清除IDLE中断标志位  // 更新当前串口接收的缓存末端  Write\_Index = (USARTx\_RINGBUF\_SIZE-DMA\_GetCurrDataCounter(USARTx\_RX\_DMA\_STREAM)) % USARTx\_RINGBUF\_SIZE;  Receive\_flag=1;  }  } |

|  |
| --- |
|  |

#### 用户命令的注册和使用

目前这个库提供了两个宏定义来实现用户命令的注册，分别是

|  |
| --- |
| shell\_register\_command(name,func);  shell\_register\_confirm(name,func,info); |

shell\_register\_command用于一般的命令注册，name为命令字符，如“reboot”,需要注意的是，同一串字符串不能注册多次，即不能注册同一条命令多次，以先到先得的原则会只注册第一条。func为命令对应的执行代码，类型为

|  |
| --- |
| void cmd\_fn(void\* arg); |

其中输入形参void \* arg 为命令行的输入内存指针。以hello-world命令为例，编写对应的执行代码：

|  |
| --- |
| void helloworld(void \* arg)  {  printk("this is hello world command\r\n");//或者printf  } |

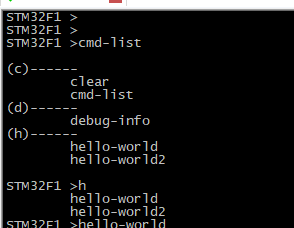
然后在main()函数调用注册：

shell\_register\_command(“hello-world”,helloworld);

便可以在cmd-list里面找到对应的命令，在终端输入hello-world,会反馈打印内容。此外还有另外一种带确认信息的注册方式:

shell\_register\_confirm (“hello-world2”,helloworld,“sure to test this command?”);

这种方式在输入命令以后需要输入[y/Y/n/N]确认命令执行。如下



#### 命令行的参数解析

上文提到，命令对应的执行函数类型为

|  |
| --- |
| void cmd\_fn(void \* arg); |

其中的void \* arg是把命令行内存首地址传入，如在终端输入"hello-world 1234" 回车，shell会先匹配到 helloworld 对应的执行代码，并把整串输入字符串的首地址作为arg输入给函数，用户可对此进行参数解析。当前库提供两个函数用于解析命令行参数

|  |
| --- |
| int cmdline\_param (char \* str,int \* argv,uint32\_t maxread);  int cmdline\_strtok(char \* str ,char \*\* argv,uint32\_t maxread); |

##### cmdline\_param()

cmdline\_param()可以把命令行后面所跟参数转为***整形数***据，不过仅支持整形的转换，包括正负数和十六进制，转换结果存储于 argv 内,字符串转换正常返回参数个数，有非数字字符会返回<0。大致如下：

|  |
| --- |
| void demo\_cmd(void \* arg)  {  int argv[4];  int argc = cmdline\_param((char\*)arg,argv,4);  printk("get %d parameters\r\n",argc);  for(int i = 0 ; i < argc ; ++i)  printk("argv[%d]:%d\r\n",i,argv[i]);  } |

##### cmdline\_strtok()不需要getopt也可以用

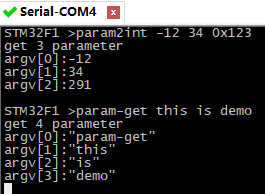
cmdline\_strtok()是对命令行的输入***字符串***进行分割，把输入分割为一串指针数组输出到argv[]中。需要注意的是这个函数会改变命令行的输入内存内容(把空格替换了字符串结束符)。如下

|  |
| --- |
| void demo2\_cmd(void \* arg)  {  char \* argv[4];  int argc =cmdline\_strtok((char\*)arg,argv,4);  printk("get %d parameters\r\n",argc);  for(int i = 0 ; i < argc ; ++i)  printk("argv[%d]:\"%s\"\r\n",i,argv[i]);  } |

##### 然后注册命令测试

shell\_register\_command(“param2int”,demo\_cmd);

shell\_register\_command(“param-get”,demo2\_cmd);



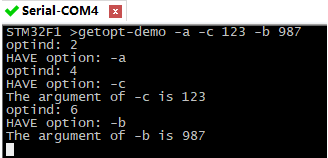
##### getopt()

利用 cmdline\_strtok() ，可以实现linux 的 getopt() 选项输入支持。由于getopt() 是gnuc的库，所以上述库还收录了 getopt() 的源码，有需要的可以把getopt.c和getopt.h加入工程便可使用getopt()函数进行输入选项分类，代码如下：

不知道怎么使用，但是报错了，后期有时间试试

|  |
| --- |
| void demo3\_cmd(void \* arg)  {  int opt;  int argc;  char\* argv[8];    argc = cmdline\_strtok((char\*)arg,argv,8);    optind = 1;//getopt()之前，这个值要为 1    while ((opt = getopt(argc, argv, "ab:c:de::")) != -1)  {  printk("optind: %d\r\n", optind);  switch (opt)  {  case 'a':  printk("HAVE option: -a\r\n");  break;  case 'b':  printk("HAVE option: -b\r\n");  printk("The argument of -b is %s\r\n", optarg);  break;  case 'c':  printk("HAVE option: -c\r\n");  printk("The argument of -c is %s\r\n", optarg);  break;  case 'd':  printk("HAVE option: -d\r\n");  break;  case 'e':  printk("HAVE option: -e\r\n");  printk("The argument of -e is %s\r\n", optarg);  break;  case '?':  printk("Unknown option: %c\r\n",(char)optopt);  break;  }  }  } |

注册shell\_register\_command(“getopt-demo”,demo3\_cmd);结果如下：



另外，getopt.c不是作者所写，编译可能会有些许警告，而且其源码编译下来也不小，还是按需使用吧。

源码链接：https://gitee.com/somebug/atomlib/blob/master/appdemo/f103-shell-demo-2.0.zip

|  |
| --- |
|  |

#### welcome的调整

##### 版本1

由于系统是每次启动后，点击输入后，才会显示欢迎。这里修改为主动显示一次

F:\1\_git\_code\RT-Thread-project\4.0.3\ATK\_IAR\_RTT\_TEMPLATE\ATK\_bootloader\SHELL\shell.c

|  |
| --- |
| void shell\_input\_init(struct shell\_input \* shellin , fmt\_puts\_t shellputs,...)  {  unsigned int arg ;  char \* shellsign = DEFAULT\_INPUTSIGN;  shellgets\_t shellgets = welcome\_gets;    va\_list ap;  va\_start(ap, shellputs); //检测有无新定义  arg = va\_arg(ap, unsigned int) ;  for (; MODIFY\_MASK == (arg & (~0x0f)) ; arg = va\_arg(ap, unsigned int) ) {  if (MODIFY\_SIGN == arg) //如果重定义当前交互的输入标志  shellsign = va\_arg(ap, char\*);  else  if (MODIFY\_GETS == arg) //如果重定义当前交互的输入流向  shellgets = (shellgets\_t)va\_arg(ap, void\*);  }  va\_end(ap);  shellin->tail = 0;  shellin->edit = 0;  shellin->puts = shellputs;  shellin->gets = shellgets;  shellin->gets = cmdline\_gets;添加一句话即可  shellin->htywrt = 0;  shellin->htyread = 0;  shellin->apparg = NULL;  strcpy(shellin->sign, shellsign);  } |

shell.h文件中添加这个申明

void welcome\_gets(struct shell\_input \* shellin,char \* recv,int len);

|  |
| --- |
| // 应用初始化  static void app\_init()  {  register\_user\_cmd();  welcome\_gets(&shell\_1,"\r",1); // 主动显示 welcome 并显示一个输入位置    esp32\_at\_app\_init();  led\_app\_init();  } |

##### 版本2

可以让shell >出现前显示一些信息

|  |
| --- |
| / \_\_\_\_\ /\ \  /\ \\_\_\_/ \_\_\_\_ \_\_\_\_ \_\_\_\_ \_\\_\ \ \_\_\_\_  \ \ \ / \_\_ \/ \_\_ \/ \_\_\_\/ \_\_ \ \ / \_\_ \  \ \ \\_\_\_L\ \L\ \ \/\ \\_\_\_\_ \ \L\ \ \\_L\ \_\_\_L  \ \\_\_\_\_\_\_\\_\_\_\_/\\_\ \\_\\_\_\_\_/\\_\_\_\_/\\_\_\_\_\\_\_\_\_/  \/\_\_\_\_\_\_/\_\_\_/\/\_/\/\_/\_\_\_/\/\_\_\_/\/\_\_\_\_/\_\_\_/  COPYRIGHT(c):GoodMorning 2019/06  INFORMATION:Read parameter success!  INFORMATION:ssid:asadfa  INFORMATION:pwd:wefqwerqwerweqr  shell > |

|  |
| --- |
| // 系统参数保存区域  sys\_parameter\_t sys\_parameter;  static void get\_system\_parameter()  {  SYS\_PARAMETER\_READ;  if(sys\_parameter.flag==SYS\_PARAMETER\_OK){  debug\_info(INFO"Read parameter success!\r\n");  debug\_info(INFO"ssid:%s\r\n",sys\_parameter.wifi\_ssid);  debug\_info(INFO"pwd:%s\r\n",sys\_parameter.wifi\_pwd);  }  else{  debug\_info(INFO"Please use %s cmd set wifi parameter!\r\n",ESP\_SET\_SSID\_PASS\_CMD);  }  }  // 系统初始化  static void system\_init()  {  NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4 ); // 中断分组  /\* 初始化 shell 控制台 \*/  shell\_hw\_init(115200); // 初始化 控制台串口硬件  shell\_init("shell >" ,usart1\_puts); // 初始化 控制台输出  shell\_input\_init(&shell\_1,usart1\_puts); // 初始化 交互  welcome\_gets(&shell\_1,0,0); // 主动显示 welcome    get\_system\_parameter(); // 读取参数    /\* shell 控制台进行用户输入 \*/  cmdline\_gets(&shell\_1,"\r",1); // 一次换行  }  // 硬件初始化  static void hardware\_init()  {  led\_init(); // 初始化 LED  systick\_init(); // 时钟初始化  softTimer\_Init(); // 软件定时器初始化  /\* 初始化 ESP8266 WiFi at串口 \*/  esp32\_at\_hw\_init(115200); // 初始化 ESP8266 WiFi at串口  }  // 应用初始化  static void app\_init()  {  register\_user\_cmd();  esp32\_at\_app\_init();  led\_app\_init();  }  // 主函数  int main(void)  {  system\_init();  hardware\_init();  app\_init();  while(1){  softTimer\_Update(); // 软件定时器扫描  shell\_app\_cycle(); // shell 控制台应用循环  esp32\_at\_app\_cycle(); // esp32 的应用循环  }  } |

|  |
| --- |
|  |

#### 常见错误

1、..\SHELL\shell.h(150): error: #20: identifier "size\_t" is undefined

|  |
| --- |
| // 以下为 shell 所依赖的基本库  #include "ustdio.h"  #include "string.h" |

2、..\OBJ\atk\_bootloader.axf: Error: L6200E: Symbol \_\_stdout multiply defined (by stdio\_streams.o and ustdio.o).

F:\1\_git\_code\RT-Thread-project\4.0.3\ATK\_IAR\_RTT\_TEMPLATE\ATK\_bootloader\SHELL\ustdio.c

|  |
| --- |
| 104 FILE \_\_stdout; |

这里和标准库冲突，可以注释掉这里

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

### 配置esp32 8266 WiFi

#### 常用AT

|  |
| --- |
| AT+RST：重启模块  AT+CMD：查询当前固件支持的所有命令及命令类型  ATE：开启或关闭 AT 回显功能    AT+SYSSTORE：设置参数存储模式 AT+SYSSTORE=1  AT+RESTORE：恢复出厂设置  AT+CWMODE：查询/设置 Wi-Fi 模式 (Station/SoftAP/Station+SoftAP) 1: Station 模式 2: SoftAP 模式  AT+CWJAP：连接 AP  AT+CWLAP：扫描当前可用的 AP  AT+CWDHCP：启用/禁用 DHCP |

连接wifi之后，直接可以使用http指令

|  |
| --- |
| AT+HTTPCLIENT=1,0,"http://httpbin.org/get","httpbin.org","/get",1  +HTTPCLIENT:35,Date: Mon, 26 Jun 2023 09:00:36 GMT  +HTTPCLIENT:30,Content-Type: application/json  +HTTPCLIENT:19,Content-Length: 329  +HTTPCLIENT:22,Connection: keep-alive  +HTTPCLIENT:23,Server: gunicorn/19.9.0  +HTTPCLIENT:30,Access-Control-Allow-Origin: \*  +HTTPCLIENT:38,Access-Control-Allow-Credentials: true |

INFORMATION:Read parameter success!

INFORMATION:ssid:Xiaomi\_B596

INFORMATION:pwd:WH15572388670LOL

|  |
| --- |
| http://v0.yiketianqi.com/api?version=v61&appid=38119558&appsecret=8k2oHWWA&cityid=101200101 |

AT+HTTPCLIENT=2,0,"http://v0.yiketianqi.com/api?version=v61&appid=38119558&appsecret=8k2oHWWA&cityid=101200101"

|  |
| --- |
|  |

#### 串口数据错误情况

|  |
| --- |
| shell >recv\_cmd\_buf:  OK  recv\_cmd\_buf:OK  valid\_cmd:OK  recv\_cmd\_buf:  recv\_cmd\_buf:  recv\_cmd\_buf:  recv\_cmd\_buf:  recv\_cmd\_buf:  recv\_cmd\_buf:  recv\_cmd\_buf:  recv\_cmd\_buf:  ERROR:recv\_cmd\_buf processing cycles exceeded  recv\_cmd\_buf:  OK  recv\_cmd\_buf:OK |

|  |
| --- |
|  |

#### 代码

F:\1\_git\_code\RT-Thread-project\4.0.3\ATK\_IAR\_RTT\_TEMPLATE\ATK\_bootloader\DRV\usart\usart3.c

|  |
| --- |
| // esp32 应用程序获取输入数据  void esp32\_usart\_data\_handle()  {  if(Write\_Index!=Read\_Index){  /\* 取环形缓存区剩余数据 \*/  char temp[USARTx\_RINGBUF\_SIZE]={0};  unsigned short data\_len= UART\_GetRemain(); // 获取当前数据长度  if(Read\_Index+data\_len>USARTx\_RINGBUF\_SIZE){  // 索引处读取长度超出缓存  int len1=USARTx\_RINGBUF\_SIZE-Read\_Index; // 环形末尾读长度  int len2=data\_len-len1;  memcpy(temp,esp32\_at\_ringbuf + Read\_Index,len1);  memcpy(temp+len1,esp32\_at\_ringbuf,len2);  }else{  memcpy(temp,esp32\_at\_ringbuf + Read\_Index,data\_len);  }    /\* 接收并处理此处数据 \*/  int ret=esp32\_command\_handle(temp,data\_len); // 回调应用层的数据处理函数  if(ret==-1){  debug\_err(ERR"Read\_Index:%d usart3\_data:%s\r\n",Read\_Index,temp);  }    /\* 数据处理结束 \*/  Read\_Index = (Read\_Index+data\_len)% USARTx\_RINGBUF\_SIZE; // 下次读取数据的起始位置，防止超出缓存区最大索引  }  } |

F:\1\_git\_code\RT-Thread-project\4.0.3\ATK\_IAR\_RTT\_TEMPLATE\ATK\_bootloader\APP\esp32\esp32\_at.c

|  |
| --- |
| #include "esp32\_at.h"  #include "usart3.h"  #include "stdio.h"  #include "stdlib.h"  #include "string.h"  #include "systick.h"  #include "soft\_timer.h"  #define PASS "\"Xiaomi\_B596\",\"WH15572388670LOL\""  #define RECV\_CMD\_BUF\_SIZE 256  #define ESP32\_RES "ESP32-->"  #define ESP32\_SEND "ESP32<--"  #define CMD\_OK {esp32\_flag |= BIT\_3;}  #define CMD\_SENDED {esp32\_flag &= ~BIT\_3;}  static char recv\_cmd\_buf[RECV\_CMD\_BUF\_SIZE];  static char send\_cmd\_count; // 发送命令的计数  static short esp32\_flag; // esp32的标志位  // 位0 进行连接AP  // 位1 备用  // 位2 备用  // 位3 0：当前命令正在处理 1：处理结束  // 位4 备用  static const char\* cmd\_list[]={"ATE0\r\n","AT+CWMODE","AT+CWJAP",};  static const char\* ReplyOk\_cmd\_list[]={"ATE0","AT+CWMODE",};  static char current\_cmd[100]={0};  // esp32发送命令  static void esp32\_send\_cmd(const char \* strbuf, unsigned short len)  {  debug\_at(ESP32\_SEND"%s",strbuf);  usart3\_puts(strbuf,len);  }  // 发送命令处理系统复位  static void esp32\_cmd\_handle\_reset()  {  // 清理控制位  esp32\_flag=0;  send\_cmd\_count=0;  // 停止命令响应超时定时器  softTimer\_stop(ESP32\_TIMEOUT\_TIMER\_ID);  memset(recv\_cmd\_buf,0,RECV\_CMD\_BUF\_SIZE);    // 可以发送一次命令  CMD\_OK;  }  // esp32 连接AP的处理函数  static void esp32\_connect\_ap\_handle(void)  {  // 控制位  if(!(esp32\_flag & BIT\_0))return;    // 命令能够发送标志位  if(!(esp32\_flag & BIT\_3))return;    // 开始发送命令  memset(current\_cmd,0,strlen(current\_cmd));  switch(send\_cmd\_count){  case 0:{  send\_cmd\_count=send\_cmd\_count%ARRAY\_SIZE(cmd\_list);  strcpy(current\_cmd,(char\*)cmd\_list[send\_cmd\_count]);  esp32\_send\_cmd(current\_cmd,strlen(current\_cmd));  softTimer\_start(ESP32\_TIMEOUT\_TIMER\_ID,1000);  }break;  case 1:{  send\_cmd\_count=send\_cmd\_count%ARRAY\_SIZE(cmd\_list);  sprintf(current\_cmd,"%s=1\r\n",(char\*)cmd\_list[send\_cmd\_count]);  esp32\_send\_cmd(current\_cmd,strlen(current\_cmd));  softTimer\_start(ESP32\_TIMEOUT\_TIMER\_ID,1000);  }break;  case 2:{  send\_cmd\_count=send\_cmd\_count%ARRAY\_SIZE(cmd\_list);  sprintf(current\_cmd,"%s=%s\r\n",(char\*)cmd\_list[send\_cmd\_count],PASS);  esp32\_send\_cmd(current\_cmd,strlen(current\_cmd));  softTimer\_start(ESP32\_TIMEOUT\_TIMER\_ID,16000); // 默认连接超时为15s  }break;  case 3:{  esp32\_cmd\_handle\_reset();  debug\_info(INFO"CONNECT AP SUCCESS!\r\n");  }  return;  default:{  softTimer\_start(ESP32\_TIMEOUT\_TIMER\_ID,1000);  }break;  }  CMD\_SENDED;  send\_cmd\_count++;  }  // 列表中是否包含字符串  // obj 目标字符串 list 字符串列表 len 列表长度  static int list\_contains\_str(char\* str,char\*\* list,int len)  {  for(int i=0;i<len;i++){  if(strstr(str,list[i]))  return 1;  }  return 0;  }  // at 命令 处理结果1  int esp32\_command\_handle(const char\* buf,unsigned short len)  {  #define REPLY\_PROCESSED do{ char\* \_\_new\_buf=substring+flag\_len; \  int \_\_new\_len=strlen(\_\_new\_buf); \  my\_memcpy(recv\_cmd\_buf,\_\_new\_buf,\_\_new\_len);\  char\* \_\_remain\_buf=recv\_cmd\_buf+\_\_new\_len; \  memset(\_\_remain\_buf,0,strlen(\_\_remain\_buf)); \  }while(0);  /\*\* 条件判断 \*\*/  // 当前 指令处理缓存 空间足够  if(strlen(recv\_cmd\_buf)+len>RECV\_CMD\_BUF\_SIZE){  debug\_err(ERR"recv\_cmd\_buf not enough space\r\n");  esp32\_cmd\_handle\_reset();  return -1;  }    /\*\* 追加到缓存区 \*\*/  strncat(recv\_cmd\_buf,buf,len);    /\*\* 处理接收缓存 \*\*/  char cycle\_counts=0;  while(1){  /\*\* 记录信息 \*\*/  char\* reply\_flag="\r\n";  int flag\_len=strlen(reply\_flag);  char\* substring=strstr(recv\_cmd\_buf,reply\_flag);  debug("recv\_cmd\_buf:%s\r\n",recv\_cmd\_buf);  if(substring==NULL)break;  // 循环检测  cycle\_counts++;  if(cycle\_counts > 5){  debug\_err(ERR"recv\_cmd\_buf processing cycles exceeded,recv\_cmd\_buf:%s,substring:%s\r\n",  recv\_cmd\_buf,substring);  esp32\_cmd\_handle\_reset();  return -1;  }    /\*\* 去除多余字符 \*\*/  memset(substring,0,flag\_len); // 删除 recv\_cmd\_buf 中第一个出现"\r\n"  char \* valid\_reply=recv\_cmd\_buf; // 得到一条有效回复  // 空的\r\n  if(strlen(valid\_reply)==0){  REPLY\_PROCESSED;  continue;  }    /\*\* 处理有效回复 \*\*/  debug("valid\_reply:%s\r\n",valid\_reply);  // 处理"OK\r\n"  if(strstr(valid\_reply, "OK")){  // 显示  debug\_at(ESP32\_RES"%s\r\n",valid\_reply);  // 处理内容  if(list\_contains\_str(current\_cmd,(char\*\*)ReplyOk\_cmd\_list,  ARRAY\_SIZE(ReplyOk\_cmd\_list)))  {  CMD\_OK;  }  }  // 显示错误  else if(strstr(valid\_reply, "ERROR")){  debug\_at(ESP32\_RES"%s\r\n",valid\_reply);  }  // 处理 AT+CWJAP 连接AP的错误情况  else if(strstr(valid\_reply, "+CWJAP:")){  // 显示  debug\_at(ESP32\_RES"%s\r\n",valid\_reply);  // 处理信息 类似 +CWJAP:4  char\* \_flag="+CWJAP:";  char info\_buf[50]={0};  int info = atoi(valid\_reply+strlen(\_flag));  switch(info){  case 1:{  strcpy(info\_buf,"connect timeout");  }break;  case 2:{  strcpy(info\_buf,"password error");  }break;  case 3:{  strcpy(info\_buf,"Unable to find target AP");  }break;  case 4:{  strcpy(info\_buf,"connect failed");  }break;  default:{  strcpy(info\_buf,"Unknown error occurred");  }break;  }  esp32\_cmd\_handle\_reset();  debug\_err(ERR"AT+CWJAP:%s\r\n",info\_buf);  }  // 处理 AT+CWJAP 连接AP的正确情况  else if(strstr(valid\_reply, "WIFI CONNECTED")){  // 显示  debug\_at(ESP32\_RES"%s\r\n",valid\_reply);  }  else if(strstr(valid\_reply, "WIFI GOT IP")){  // 显示  debug\_at(ESP32\_RES"%s\r\n",valid\_reply);  CMD\_OK;  }  else if(strstr(valid\_reply, "WIFI DISCONNECT")){  // 显示  debug\_at(ESP32\_RES"%s\r\n",valid\_reply);  }  // 当前回复没有处理  else{  debug\_err(ERR"valid\_reply:%s not processed\r\n",valid\_reply);  }    /\*\* 删除此条有效回复 \*\*/  REPLY\_PROCESSED;  }  return 0;  }  // at 命令 处理结果2 发送命令超时  static void esp32\_send\_cmd\_timeout()  {  esp32\_cmd\_handle\_reset();  debug\_war(WARNING"send\_cmd\_timeout\r\n");  }  // esp32\_at 应用层初始化  void esp32\_at\_app\_init(void)  {  softTimer\_create(ESP32\_TIMEOUT\_TIMER\_ID,MODE\_ONE\_SHOT,esp32\_send\_cmd\_timeout);  // softTimer\_create(TEST\_TIMER\_ID,MODE\_PERIODIC,esp32\_connect\_ap\_start);  // softTimer\_start(TEST\_TIMER\_ID,1000);  esp32\_cmd\_handle\_reset(); // 首次复位  }  // esp32\_at 应用周期循环函数  void esp32\_at\_app\_cycle(void)  {  esp32\_connect\_ap\_handle(); // 控制连接esp32连接AP  esp32\_usart\_data\_handle(); // 周期处理串口数据  }  // esp32 开始连接 ap  void esp32\_connect\_ap\_start(void)  {  if(!(esp32\_flag & BIT\_0)){  esp32\_cmd\_handle\_reset();  esp32\_flag|=BIT\_0;  }  else{  debug\_info(INFO"ESP32 connecting ap\r\n");  }  } |

|  |
| --- |
|  |

#### 常见问题

1、无法连接WiFi AT+CWLAP无法扫描

AT+CWMODE?

默认为2，要设置为1

### 内部flash操作

参考

|  |
| --- |
| #include "flash.h"  #include <string.h>  //读取指定地址的字(32位数据)  //faddr:读地址  //返回值:对应数据.  static u32 STMFLASH\_ReadWord(u32 faddr)  {  return \*(vu32\*)faddr;  }  //获取某个地址所在的flash扇区  //addr: flash地址  //返回值:0~11,即addr所在的扇区  static uint16\_t STMFLASH\_GetFlashSector(u32 addr)  {  if(addr<ADDR\_FLASH\_SECTOR\_1)return FLASH\_Sector\_0;  else if(addr<ADDR\_FLASH\_SECTOR\_2)return FLASH\_Sector\_1;  else if(addr<ADDR\_FLASH\_SECTOR\_3)return FLASH\_Sector\_2;  else if(addr<ADDR\_FLASH\_SECTOR\_4)return FLASH\_Sector\_3;  else if(addr<ADDR\_FLASH\_SECTOR\_5)return FLASH\_Sector\_4;  else if(addr<ADDR\_FLASH\_SECTOR\_6)return FLASH\_Sector\_5;  else if(addr<ADDR\_FLASH\_SECTOR\_7)return FLASH\_Sector\_6;  else if(addr<ADDR\_FLASH\_SECTOR\_8)return FLASH\_Sector\_7;  else if(addr<ADDR\_FLASH\_SECTOR\_9)return FLASH\_Sector\_8;  else if(addr<ADDR\_FLASH\_SECTOR\_10)return FLASH\_Sector\_9;  else if(addr<ADDR\_FLASH\_SECTOR\_11)return FLASH\_Sector\_10;  return FLASH\_Sector\_11;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 特别注意:因为STM32F4的扇区实在太大,没办法本地保存扇区数据,所以本函数  写地址如果非0XFF,那么会先擦除整个扇区且不保存扇区数据.所以  写非0XFF的地址,将导致整个扇区数据丢失.建议写之前确保扇区里  没有重要数据,最好是整个扇区先擦除了,然后慢慢往后写.  \* 函数功能：从指定地址开始写入指定长度的数据  \* 该函数对OTP区域也有效!可以用来写OTP区! OTP区域地址范围:0X1FFF7800~0X1FFF7A0F  \* WriteAddr: 起始地址(此地址必须为4的倍数!!) 0x8000 0000 - 0x080F FFFF  \* pBuffer: 数据指针  \* NumToWrite: 字(32位)的数量，多少个字(32位)  \* 返回：0表示成功 写入成功字(32位)的数量  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  u32 STMFLASH\_Write(u32 WriteAddr,u32\* pBuffer,u32 NumToWrite)  {  FLASH\_Status status = FLASH\_COMPLETE;  u32 addrx=WriteAddr; // 写入的起始地址  u32 end\_addr=WriteAddr+NumToWrite\*4; // 写入的结束地址  u32 word\_num=0;  /\* 地址检查 \*/  if( addrx < STM32\_FLASH\_BASE ||  addrx >= ADDR\_FLASH\_SYSTEM\_MEM || // 起始地址检查  end\_addr >= ADDR\_FLASH\_SYSTEM\_MEM || // 终止地址检查  addrx % 4  ){  debug\_err(ERR"STMFLASH\_Write Address error! addrx:0x%08x\r\n",addrx);  return 0;  }    /\* 准备写入FLASH \*/  FLASH\_Unlock(); // 解锁  FLASH\_DataCacheCmd(DISABLE); // FLASH擦除期间,必须禁止数据缓存    /\* 先擦除所在的扇区 FLASH \*/  while(addrx < end\_addr) // 扫清一切障碍.(对非FFFFFFFF的地方,先擦除)  {  if(STMFLASH\_ReadWord(addrx)!=0XFFFFFFFF)// 有非0XFFFFFFFF的地方,要擦除这个扇区  {  uint16\_t Sector\_num = STMFLASH\_GetFlashSector(addrx);  status=FLASH\_EraseSector(Sector\_num,VoltageRange\_3); //VCC=2.7~3.6V之间!!  if(status!=FLASH\_COMPLETE){  debug\_err(ERR"FLASH\_EraseSector error! Sector\_num:0x%04x\r\n",Sector\_num);  return 0;  }  }else addrx+=4; // 当前可写终止地址  }  /\* 开始编程 FLASH \*/  while(WriteAddr < addrx)  {  if(FLASH\_ProgramWord(WriteAddr,\*pBuffer)!=FLASH\_COMPLETE)// 写入数据  {  debug\_err(ERR"FLASH\_ProgramWord error! WriteAddr:0x%08x\r\n",WriteAddr);  return word\_num;  }  WriteAddr+=4;  pBuffer++;  word\_num++;  }  /\* 写入FLASH结束 \*/  FLASH\_DataCacheCmd(ENABLE); // FLASH擦除结束,开启数据缓存  FLASH\_Lock(); // 上锁  return word\_num;  }  //从指定地址开始读出指定长度的数据  //ReadAddr:起始地址  //pBuffer:数据指针  //NumToRead:字(4位)数  void STMFLASH\_Read(u32 ReadAddr,u32\* pBuffer,u32 NumToRead)  {  u32 i;  for(i=0;i<NumToRead;i++)  {  pBuffer[i]=STMFLASH\_ReadWord(ReadAddr); // 读取4个字节.  ReadAddr+=4; // 偏移4个字节.  }  }  // 要写入到STM32 FLASH的字符串数组  const u8 TEXT\_Buffer[]={"This a FLASH TEST Program"};  #define TEXT\_LENTH sizeof(TEXT\_Buffer) // 数组长度  #define SIZE (TEXT\_LENTH/4+((TEXT\_LENTH%4)?1:0)) // 多少个字  // flash 测试程序  // start\_add: 起始地址(此地址必须为4的倍数!!)  // (废弃)NumToWrite 测试写入多少个TEXT\_Buffer  // end\_add：结束地址  // STMFLASH\_Read\_Write\_test(ADDR\_FLASH\_SECTOR\_3-96,ADDR\_FLASH\_SECTOR\_3+1000);  void STMFLASH\_Read\_Write\_test(u32 start\_add,u32 end\_add)  {  static char count=0;  char datatemp[TEXT\_LENTH+1]={0};  // u32 end\_add=start\_add+NumToWrite\*SIZE\*4;  while(start\_add < end\_add && count==0){  int write\_len=STMFLASH\_Write(start\_add,(u32\*)TEXT\_Buffer,SIZE);  STMFLASH\_Read(start\_add,(u32\*)datatemp,SIZE);  if(memcmp(datatemp,TEXT\_Buffer,TEXT\_LENTH)==0){  debug\_info(INFO"STMFLASH\_Read Success,start\_add:0x%08x writelen:%d str:%s\r\n",start\_add,write\_len,datatemp);  }  else{  debug\_err(ERR"STMFLASH\_Read\_Write\_test error!\r\n");  }  start\_add+=SIZE\*4;  }  count=1;  } |

### 适配IAP升级部分

#### 参考链接

<https://blog.csdn.net/qq_39854159/article/details/123497537>

Ymodem下载协议

<https://blog.csdn.net/u012993936/article/details/123009669>

STM32 IAP 在线升级原理全解析

<http://www.openedv.com/thread-78079-1-1.html>

IAP+YMODEM+CRC16+AES256+PC端软件+hex合并

<https://blog.csdn.net/qq_41959288/article/details/107192835>

YModem协议总结

#### Ymodem协议

Ymodem协议是一种用于在串行通信中进行文件传输的协议。它是从Xmodem协议演化而来，提供了更高的传输速率和更强的容错能力。

Ymodem协议使用了CRC校验和数据包序号等机制，以确保可靠的文件传输。下面是Ymodem协议的主要特点：

* + 数据传输：Ymodem协议允许将文件分成较小的数据块进行传输。每个数据块包含一个数据包头、数据内容、CRC校验和和结束字符。通过以数据块为基本单位进行传输，可以提高传输效率。
  + 文件名和文件大小：Ymodem协议支持传输文件的名称和大小信息。发送方会将这些信息放在文件传输开始前的数据包头中，接收方可以用于确认传输文件的属性。
  + 校验和：Ymodem协议使用CRC校验（循环冗余校验）来验证数据包的完整性。接收方会对接收到的数据进行校验，并在校验失败时要求发送方重新发送。
  + 实时反馈：接收方会向发送方发送特殊的ACK（确认）或NAK（否定确认）信号，以指示接收状态。ACK表示接收成功，NAK表示接收失败，需要重新发送。
  + 重传机制：如果发送的数据包在一定时间内未收到接收方的确认信号，发送方会重新发送该数据包。

总体而言，Ymodem协议在串行通信中提供了可靠的文件传输机制。它通过有效的数据封装、校验和反馈机制，确保了数据的完整性和准确性。由于其较高的传输速率和可靠性，Ymodem协议被广泛用于嵌入式系统等领域的文件传输需求。

#### 帧格式



<https://blog.csdn.net/qq_41959288/article/details/107192835>

#### 帧头

SOH 01H（Ymodem 128数据帧头）

STX 02H（Ymodem 1024数据帧头）

EOT 04H（发送结束）

ACK 06H（应答）

NAK 15H（非应答）

CAN 18H（取消发送）



#### 起始帧

| **Byte1** | **Byte2** | **Byte3** | **Byte4~Byte （4+N\* ）** | **Byte （4+N+1 ）~Byte（4+N +1+M\* ）** | **NULL** | **Byte 127** | **Byte 128** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SOH | 0x00 | 0xFF | filename | filezise | 0x00 | CRC\_H | CRC\_L |



首先超级终端中Ymodem协议是这样的，定义单片机位接受者，超级终端为发送者：

SOH开头： 128字节的数据，总包大小128+5，SOH即0x01

STX开头： 1024字节的数据，总包大小1024+5 ，STX即0x02

EOT开头： 单个字节，传输完成，总包大小1 ，EOT即0x04

解析：

Byte1

SOH = 0x01,起始数据帧只有128个字节

Byte2

0x00表示数据帧序号，初始时0x00，下个时0x01

Byte3

帧序号取反

Byte4~Byte （4+N\* ）

N\*表示文件文件名长度+1（文件名长度+结束符）

filename是传输文件的文件名，把ASCII转换成十六进制还要加个结束符0x00,例如abc.bin,转换后是61 62 63 2e 62 69 6e 00

Byte （4+N+1 ）~Byte（4+N +1+M\* ）

M\* 文件长度的ASCII转换成HEX后的字节数+1（结束符）

filezise是文件的大小，如果文件时1KB，则是1024Byte，也是要ASCII转换成十六进制加结束符0x00的，31 30 32 34 00

NULL

NULL是使用0x00填充起始帧128字节的其他字节

Byte 127、Byte 128

CRC16校验，然后高8位在前，低8位在后

#### 数据帧

| **Byte1** | **Byte2** | **Byte3** | **Byte4~Byte1027** | **Byte 1028** | **Byte 1029** |
| --- | --- | --- | --- | --- | --- |
| STX | 0x01 | 0xFE | data[1024] | CRC\_H | CRC\_L |



首先超级终端中Ymodem协议是这样的，定义单片机位接受者，超级终端为发送者：

SOH开头： 128字节的数据，总包大小128+5，SOH即0x01

STX开头： 1024字节的数据，总包大小1024+5 ，STX即0x02

EOT开头： 单个字节，传输完成，总包大小1 ，EOT即0x04

解析：

Byte 1

STX是0x02，表示这个帧数据部分包含1024个字节

Byte 2

与起始帧一样表示帧序号，第二个数据帧则是0x02，其取反则是0xFD，第三帧则是0x03

Byte 3

Byte 2 的取反

Byte3~Byte1027

1024Byte的数据。

Byte 1028 Byte 1029

CRC16校验，然后高8位在前，低8位在后

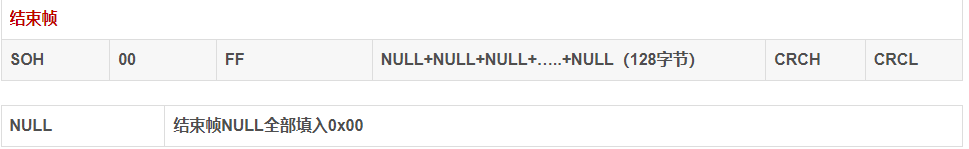
* 特殊情况

若文件最后剩余部分小于1024Byte，则拆分成128字节传输，还是不足之处使用0x1A填充。

参考Ymodem手册

#### 结束帧

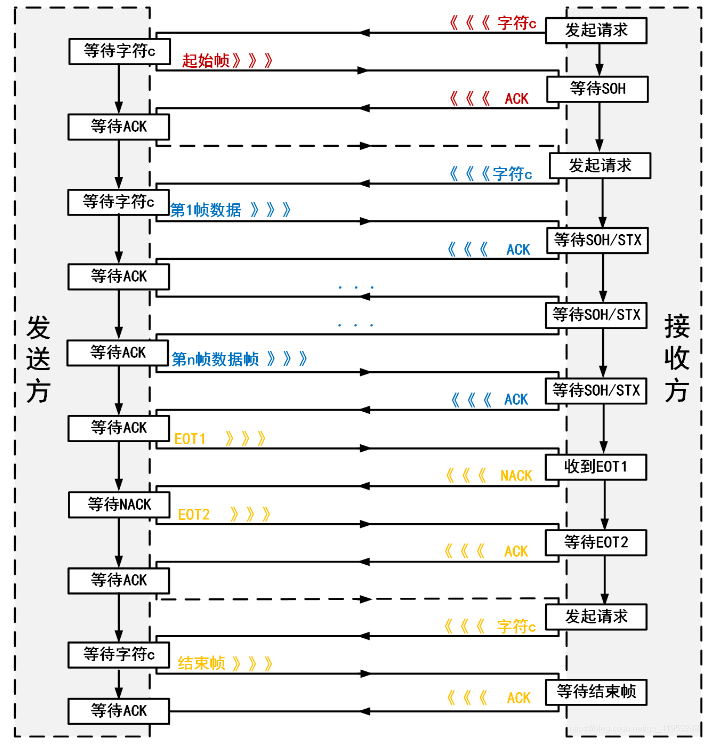
| **Byte1** | **Byte2** | **Byte3** | **Byte4~Byte131** | **Byte 132** | **Byte 133** |
| --- | --- | --- | --- | --- | --- |
| SOH | 0x00 | 0xFF | 0x00 | CRC\_H | CRC\_L |



Byte3~Byte131

全部使用0x00填充。

#### 传输流程说明



注意的点：

1. 帧的发送之前，都是通过接收端发送字符‘c’来请求；

2. 接收端接收到帧之后，都会进行一次ACK应答。

流程文字说明：

1. 接收方开启传输，接收方发送一个字符’c’，进入等待（SOH）状态，没回应，超时退出；

2. 发送方开始时处于等待状态，等待字符’c’。若发送方收到’C’后，发送第一帧（起始帧）发送完毕后，进入等待(ACK)状态；

3. 接收方收到第一帧数据包后，进行CRC校验，校验通过，则发送ACK；

4. 发送方收到ACK，又进入等待“文件传输开启”信号，进入等待’c’状态；

5. 接收方又发出一个字符’c’，开始准备接收文键。进入等待SOH或STX状态；

6. 发送方收到字符’c’后，开始发送数据帧；

7. 接收方收到数据后，发送一个ACK，若要传输数据包含多帧数据，继续ACK应答，直到所有数据传输完毕；

8. 数据传输完毕后，发送方发送EOT，第一次···以NACK应答，进行二次确认。发送方收到NAK后，重发EOT，接收方第二次收到结束符，就发送ACK应答。最后接收方在发送一个字符’c’开启另一次传输，发送方在没有第二个文件要传输的情况下，发送结束帧，正式结束数据传输。

————————————————

版权声明：本文为CSDN博主「Ming天过后」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/qq\_41959288/article/details/107192835

#### 源码下载

1、可以使用第三方的，可以就使用shell类似作者的源码

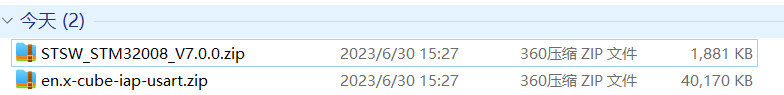
2、可以使用官方的

<https://www.stmcu.com.cn/Search/index?csrf_token=2bb7f1aa88437ad8070c120a96525f85&search_keywords=IAP&type=design_resource&page=1&product_family=81>

在官网中搜索IAP即可



这里前面几个都是hal版本的，而最后一个才是标准库版本的，这里我们可以使用这个版本的



然后下载后，上面一个是标准库的示例。下面一个cube的hal库示例

##### hal库的参考

例程获取

可以通过访问官方网站www.st.com获取示例代码和应用笔记

示例代码：x-cube-iap-usart

应用笔记：an4657

同时本文涉及的所有资料可以从此下载：

链接：https://pan.baidu.com/s/19nKPc\_oOyRZCTfaNKTNHbw

提取码：q0ge

下载后得到这个文件

en.x-cube-iap-usart.zip

解压后得到文件夹

AN4657-STM32Cube\_IAP\_using\_UART

##### 标准库

STSW\_STM32008\_V7.0.0.zip

#### 基于官方IAP工程适配自己的IAP（HAL库）

en.x-cube-iap-usart.zip

工程文件结构

主要就两个文件夹：Drivers和Projects，前者是HAL库的驱动文件和示例开发板BSP包，后者是示例工程，有F1、L0、L4这三个系列单片机开发板的示例，可惜没有我手头的stm32g412dicovery开发板的例程，所以后面要手动移植。Bootloader主要使用到了USART和CRC外设，前者用于用户交互和文件传输，后者用于文件校验。Projects中包含两个文件夹：IAP\_Main和IAP\_Binary\_Template，前者是bootloader的实现（主要研究对象），后者是用户app模板。

IAP\_Main包含了bootloader的主要实现文件和适配IDE的工程，工程文件没什么好说的，主要需要分析的源代码文件如下：

main.c main.h : 主函数入口，进行外设和HAL库初始化，调用bootloader或跳转用户程序；

menu.c menu.h : 命令行菜单实现

ymodem.c ymodem.h : ymodem文件传输协议实现

flash\_if.c flash\_if.h : FLASH读写函数

common.c common.h : 一些通用的函数和宏定义，主要在menu.c和ymodem.c中调用

##### 移植文件

由于这里使用的F4的探索者，所以参考L4的

AN4657-STM32Cube\_IAP\_using\_UART\Projects\STM32L476G\_EVAL\IAP\_Main\Inc

ymodem.h

common.h

C:\Users\Administrator\Desktop\AN4657-STM32Cube\_IAP\_using\_UART\Projects\STM32L476G\_EVAL\IAP\_Main\Src

common.c

ymodem.c

|  |
| --- |
|  |

|  |
| --- |
|  |

##### 修改适配

###### ymodem.c

|  |
| --- |
| /\* Includes ------------------------------------------------------------------\*/  #include "flash\_if.h"  #include "common.h"  #include "ymodem.h"  #include "string.h"  #include "main.h"  #include "menu.h" |

|  |
| --- |
|  |

###### common.h

|  |
| --- |
| #include "stm32l4xx.h"  #include "stm32l476g\_eval.h" |

|  |
| --- |
|  |

|  |
| --- |
|  |

#### 基于标准库的IAP移植适配ymodem (基于STSW\_STM32008\_V7.0.0.zip)

##### 移植文件

common.c、ymodem.c及其头文件

|  |
| --- |
|  |

|  |
| --- |
|  |

##### 适配文件

###### common.c（后期删除）

删除这些函数。没哟用到

|  |
| --- |
| void Main\_Menu(void)  void FLASH\_DisableWriteProtectionPages(void)  uint32\_t FLASH\_PagesMask(\_\_IO uint32\_t Size)  下面这两个就是从串口读取数据  uint32\_t SerialKeyPressed(uint8\_t \*key)  uint8\_t GetKey(void)  void SerialPutChar(uint8\_t c)  void Serial\_PutString(uint8\_t \*s)  void GetInputString (uint8\_t \* buffP)  uint32\_t GetIntegerInput(int32\_t \* num) |

这两个

|  |
| --- |
| uint32\_t Str2Int(uint8\_t \*inputstr, int32\_t \*intnum)  void Int2Str(uint8\_t\* str, int32\_t intnum) |

保留移动到ymodem.c

###### common.h（后期删除）

这个文件中的这些移动到config.h项目文件中。

|  |
| --- |
| /\*\* 后面调整 \*\*/  typedef void (\*pFunction)(void);  #define ApplicationAddress APP1\_START\_ADDR  #define PAGE\_SIZE (0x800) /\* 2 Kbytes \*/  #define FLASH\_SIZE (0x100000) /\* 1 MByte \*/  /\* Compute the FLASH upload image size \*/  #define FLASH\_IMAGE\_SIZE (uint32\_t) (FLASH\_SIZE - (ApplicationAddress - 0x08000000)) |

这些项目移动到ymodem.h文件中，

|  |
| --- |
| /\* Common routines \*/  #define IS\_AF(c) ((c >= 'A') && (c <= 'F'))  #define IS\_af(c) ((c >= 'a') && (c <= 'f'))  #define IS\_09(c) ((c >= '0') && (c <= '9'))  #define ISVALIDHEX(c) IS\_AF(c) || IS\_af(c) || IS\_09(c)  #define ISVALIDDEC(c) IS\_09(c)  #define CONVERTDEC(c) (c - '0')  #define CONVERTHEX\_alpha(c) (IS\_AF(c) ? (c - 'A'+10) : (c - 'a'+10))  #define CONVERTHEX(c) (IS\_09(c) ? (c - '0') : CONVERTHEX\_alpha(c)) |

然后common.h可以删除了

###### ymodem.c（需要调整）

|  |
| --- |
| /\* Includes ------------------------------------------------------------------\*/  #include "sys.h"  #include "ymodem.h"  #include "stdio.h"  #include "string.h" |

|  |
| --- |
| /\*\*  \* @brief Receive byte from sender  \* @param c: Character  \* @param timeout: Timeout  \* @retval 0: Byte received  \* -1: Timeout  \*/  static int32\_t Receive\_Byte (uint8\_t \*c, uint32\_t timeout)  {  while (timeout-- > 0)  {        if (SerialKeyPressed(c) == 1)  {  return 0;  }  }  return -1;  }  /\*\*  \* @brief Send a byte  \* @param c: Character  \* @retval 0: Byte sent  \*/  static uint32\_t Send\_Byte (uint8\_t c)  {  SerialPutChar(c);  return 0;  } |

调整完成后，这两个函数就需要重点的适配了。

适配Send\_Byte

|  |
| --- |
| static uint32\_t Send\_Byte (uint8\_t c)  {  SerialPutChar(c);  return 0;  } |

|  |
| --- |
| static uint32\_t Send\_Byte (uint8\_t c)  {  usart1\_put(c);  return 0;  } |

适配Receive\_Byte

|  |
| --- |
| static int32\_t Receive\_Byte (uint8\_t \*c, uint32\_t timeout)  {  while (timeout-- > 0)  {  // printk("\033[2J\033[%d;%dH%s",0,0,shell\_1.sign); // 清屏  if (usart1\_getchar(c) == 1)  {  return 0;  }  }  return -1;  } |

|  |
| --- |
| // 从缓存区读取一个字节  int usart1\_getchar(uint8\_t\* data)  {  if(Write\_Index!=Read\_Index){  \*data=shell\_ringbuf[Read\_Index];  Read\_Index = (Read\_Index+1)% USARTx\_RINGBUF\_SIZE; // 读取索引加1  return 1;  }else return 0;  } |

适配Ymodem\_Receive

|  |
| --- |
| Ymodem\_Receive |

我们写的时候自动检查

|  |
| --- |
| /\*\*  \* @brief Receive a file using the ymodem protocol  \* @param buf: Address of the first byte  \* @retval The size of the file  \*/  int32\_t Ymodem\_Receive (uint8\_t \*buf)  {  uint8\_t packet\_data[PACKET\_1K\_SIZE + PACKET\_OVERHEAD], file\_size[FILE\_SIZE\_LENGTH], \*file\_ptr, \*buf\_ptr;  int32\_t i, j, packet\_length, session\_done, file\_done, packets\_received, errors, session\_begin, size = 0;  /\* 初始化写flash的地址 \*/  FlashDestination = ApplicationAddress;  for (session\_done = 0, errors = 0, session\_begin = 0; ;)  {  for (packets\_received = 0, file\_done = 0, buf\_ptr = buf; ;)  {  /\* 开始接收数据包 \*/  switch (Receive\_Packet(packet\_data, &packet\_length, NAK\_TIMEOUT))  {  /\* 正常处理数据包 \*/  case 0:  errors = 0;  switch (packet\_length)  {  /\* Abort by sender \*/  case - 1:  Send\_Byte(ACK);  return 0;  /\* End of transmission \*/  case 0:  Send\_Byte(ACK);  file\_done = 1;  break;  /\* Normal packet \*/  default:  if ((packet\_data[PACKET\_SEQNO\_INDEX] & 0xff) != (packets\_received & 0xff))  {  Send\_Byte(NAK);  }  else  {  if (packets\_received == 0)  {  /\* Filename packet \*/  if (packet\_data[PACKET\_HEADER] != 0)  {  /\* Filename packet has valid data \*/  for (i = 0, file\_ptr = packet\_data + PACKET\_HEADER; (\*file\_ptr != 0) && (i < FILE\_NAME\_LENGTH);)  {  file\_name[i++] = \*file\_ptr++;  }  file\_name[i++] = '\0';  for (i = 0, file\_ptr ++; (\*file\_ptr != ' ') && (i < FILE\_SIZE\_LENGTH);)  {  file\_size[i++] = \*file\_ptr++;  }  file\_size[i++] = '\0';  Str2Int(file\_size, &size);  /\* Test the size of the image to be sent \*/  /\* Image size is greater than Flash size \*/  u32 addrx=FlashDestination; // 获取当前要写入的地址  u32 end\_addr=FlashDestination+size;    /\* 地址检查 \*/  if (size > (STM32\_FLASH\_SIZE - 1) || addrx < STM32\_FLASH\_BASE ||  addrx >= STM32\_FLASH\_END || // 起始地址检查  end\_addr >= STM32\_FLASH\_END || // 终止地址检查  addrx % 4 )  {  /\* End session \*/  Send\_Byte(CA);  Send\_Byte(CA);  return -1;  }  /\* Erase the needed pages where the user application will be loaded \*/  /\* Define the number of page to be erased \*/  FLASH\_Unlock(); // 解锁  FLASH\_DataCacheCmd(DISABLE); // FLASH擦除期间,必须禁止数据缓存    /\* 先擦除所在的扇区 FLASH \*/  while(addrx < end\_addr) // 扫清一切障碍.(对非FFFFFFFF的地方,先擦除)  {  if(\*(vu32\*)addrx!=0XFFFFFFFF) // 有非0XFFFFFFFF的地方,要擦除这个扇区,同时进行校验  {  uint16\_t Sector\_num = STMFLASH\_GetFlashSector(addrx);  if(FLASH\_EraseSector(Sector\_num,VoltageRange\_3)!=FLASH\_COMPLETE){ //VCC=2.7~3.6V之间!!  /\* End session \*/  Send\_Byte(CA);  Send\_Byte(CA);  return -1;  }  }else addrx+=4; // 当前可写终止地址  }  FLASH\_DataCacheCmd(ENABLE); // FLASH擦除结束,开启数据缓存  FLASH\_Lock(); // 上锁    Send\_Byte(ACK);  Send\_Byte(CRC16);  }  /\* Filename packet is empty, end session \*/  else  {  Send\_Byte(ACK);  file\_done = 1;  session\_done = 1;  break;  }  }  /\* Data packet \*/  else  {  memcpy(buf\_ptr, packet\_data + PACKET\_HEADER, packet\_length);  RamSource = (uint32\_t)buf;    FLASH\_Unlock(); // 解锁  FLASH\_DataCacheCmd(DISABLE); // FLASH擦除期间,必须禁止数据缓存  for (j = 0;(j < packet\_length) && (FlashDestination < ApplicationAddress + size);j += 4)  {  /\* Program the data received into STM32F10x Flash \*/  FLASH\_ProgramWord(FlashDestination, \*(uint32\_t\*)RamSource);  if (\*(uint32\_t\*)FlashDestination != \*(uint32\_t\*)RamSource)  {  /\* End session \*/  Send\_Byte(CA);  Send\_Byte(CA);  return -2;  }  FlashDestination += 4;  RamSource += 4;  }  Send\_Byte(ACK);  }  FLASH\_DataCacheCmd(ENABLE); // FLASH擦除结束,开启数据缓存  FLASH\_Lock(); // 上锁    packets\_received ++;  session\_begin = 1;  }  }  break;  case 1:  Send\_Byte(CA);  Send\_Byte(CA);  return -3;  default:  if (session\_begin > 0)  {  errors ++;  }  if (errors > MAX\_ERRORS)  {  Send\_Byte(CA);  Send\_Byte(CA);  return 0;  }  Send\_Byte(CRC16);  break;  }  if (file\_done != 0)  {  break;  }  }  if (session\_done != 0)  {  break;  }  }  return (int32\_t)size;  } |

基本上就黄色区域需要修改

###### ymodem.h

|  |
| --- |
|  |

###### ymodem.c中代码分析

主要函数就是Ymodem\_Receive函数，这个函数用于处理Ymodem协议的数据包

|  |
| --- |
| int32\_t Ymodem\_Receive (uint8\_t \*buf)  {  uint8\_t packet\_data[PACKET\_1K\_SIZE + PACKET\_OVERHEAD], file\_size[FILE\_SIZE\_LENGTH], \*file\_ptr, \*buf\_ptr;  int32\_t i, j, packet\_length, session\_done, file\_done, packets\_received, errors, session\_begin, size = 0;  /\* 初始化写flash的地址 \*/  FlashDestination = ApplicationAddress;  for (session\_done = 0, errors = 0, session\_begin = 0; ;)  {  for (packets\_received = 0, file\_done = 0, buf\_ptr = buf; ;)  {  /\* 开始接收数据包 \*/  switch (Receive\_Packet(packet\_data, &packet\_length, NAK\_TIMEOUT))  {  /\* 正常处理数据包 \*/  case 0:  errors = 0;  switch (packet\_length)  {  /\* Abort by sender \*/  case - 1:  Send\_Byte(ACK);  return 0;  /\* End of transmission \*/  case 0:  Send\_Byte(ACK);  file\_done = 1;  break;  /\* 处理正常的数据包 \*/  default:  if ((packet\_data[PACKET\_SEQNO\_INDEX] & 0xff) != (packets\_received & 0xff))  {  Send\_Byte(NAK);  }  else  {  if (packets\_received == 0)  {  /\* Filename packet \*/  if (packet\_data[PACKET\_HEADER] != 0)  {  /\* Filename packet has valid data \*/  for (i = 0, file\_ptr = packet\_data + PACKET\_HEADER; (\*file\_ptr != 0) && (i < FILE\_NAME\_LENGTH);)  {  file\_name[i++] = \*file\_ptr++;  }  file\_name[i++] = '\0';  for (i = 0, file\_ptr ++; (\*file\_ptr != ' ') && (i < FILE\_SIZE\_LENGTH);)  {  file\_size[i++] = \*file\_ptr++;  }  file\_size[i++] = '\0';  Str2Int(file\_size, &size); // 这里接收的就是 数据包的 字节数，就是发送文件的大小      /\* Test the size of the image to be sent \*/  /\* Image size is greater than Flash size \*/  u32 addrx=FlashDestination; // 获取当前要写入的地址  u32 end\_addr=FlashDestination+size;    /\* 地址检查 \*/  if (size > (STM32\_FLASH\_SIZE - 1) || addrx < STM32\_FLASH\_BASE ||  addrx >= STM32\_FLASH\_END || // 起始地址检查  end\_addr >= STM32\_FLASH\_END || // 终止地址检查  addrx % 4 )  {  /\* End session \*/  Send\_Byte(CA);  Send\_Byte(CA);  return -1;  }  /\* Erase the needed pages where the user application will be loaded \*/  /\* Define the number of page to be erased \*/  FLASH\_Unlock(); // 解锁  FLASH\_DataCacheCmd(DISABLE); // FLASH擦除期间,必须禁止数据缓存    /\* 先擦除所在的扇区 FLASH \*/  while(addrx < end\_addr) // 扫清一切障碍.(对非FFFFFFFF的地方,先擦除)  {  if(\*(vu32\*)addrx!=0XFFFFFFFF) // 有非0XFFFFFFFF的地方,要擦除这个扇区,同时进行校验  {  uint16\_t Sector\_num = STMFLASH\_GetFlashSector(addrx);  if(FLASH\_EraseSector(Sector\_num,VoltageRange\_3)!=FLASH\_COMPLETE){ //VCC=2.7~3.6V之间!!  /\* End session \*/  Send\_Byte(CA);  Send\_Byte(CA);  return -1;  }  }else addrx+=4; // 当前可写终止地址  }  FLASH\_DataCacheCmd(ENABLE); // FLASH擦除结束,开启数据缓存  FLASH\_Lock(); // 上锁    Send\_Byte(ACK);  Send\_Byte(CRC16);  }  /\* Filename packet is empty, end session \*/  else  {  Send\_Byte(ACK);  file\_done = 1;  session\_done = 1;  break;  }  }  /\* Data packet \*/  else  {  memcpy(buf\_ptr, packet\_data + PACKET\_HEADER, packet\_length);  RamSource = (uint32\_t)buf;    FLASH\_Unlock(); // 解锁  FLASH\_DataCacheCmd(DISABLE); // FLASH擦除期间,必须禁止数据缓存  for (j = 0;(j < packet\_length) && (FlashDestination < ApplicationAddress + size);j += 4)  {  /\* Program the data received into STM32F10x Flash \*/  FLASH\_ProgramWord(FlashDestination, \*(uint32\_t\*)RamSource);  if (\*(uint32\_t\*)FlashDestination != \*(uint32\_t\*)RamSource)  {  /\* End session \*/  Send\_Byte(CA);  Send\_Byte(CA);  return -2;  }  FlashDestination += 4;  RamSource += 4;  }  Send\_Byte(ACK);  }  FLASH\_DataCacheCmd(ENABLE); // FLASH擦除结束,开启数据缓存  FLASH\_Lock(); // 上锁    packets\_received ++;  session\_begin = 1;  }  }  break;  case 1:  Send\_Byte(CA);  Send\_Byte(CA);  return -3;  default:  if (session\_begin > 0)  {  errors ++;  }  if (errors > MAX\_ERRORS)  {  Send\_Byte(CA);  Send\_Byte(CA);  return 0;  }  Send\_Byte(CRC16);  break;  }  if (file\_done != 0)  {  break;  }  }  if (session\_done != 0)  {  break;  }  }  return (int32\_t)size;  } |

##### 适配SecureCRTPortable.exe的YMODEM

参考STM32官方的即可。

##### 适配超级终端的YMODEM

<https://www.cnblogs.com/ycpkbql/p/9117129.html>

首先超级终端中Ymodem协议是这样的，定义单片机位接受者，超级终端为发送者：

SOH开头： 128字节的数据，总包大小128+5，SOH即0x01

STX开头： 1024字节的数据，总包大小1024+5 ，STX即0x02

EOT开头： 单个字节，传输完成，总包大小1 ，EOT即0x04

/\*YModem standard CMD\*/

#define YMODEM\_SOH (0x01)

#define YMODEM\_STX (0x02)

#define YMODEM\_EOT (0x04)

#define YMODEM\_ACK (0x06)

#define YMODEM\_NAK (0x15)

#define YMODEM\_CAN (0x18)

#define YMODEM\_C (0x43)

以下为交流顺序：

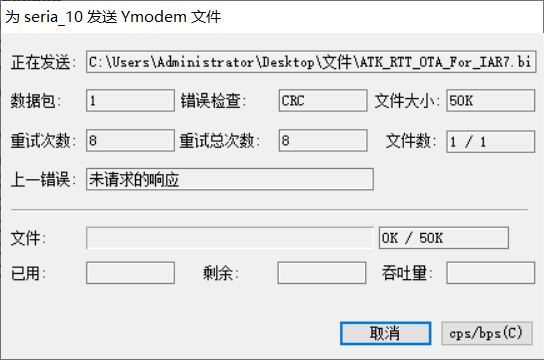
|  |
| --- |
| 接受者：发送大写字母C，等待数据  发送者：发送ASCII中的SOH也就是01，后面是包序号00，包序号反码FF，文件名YG06\_V0\_1.bin，NULL，文件大小11212 bytes ，补齐111字节至128字节，CRC CRC是数据段的16位CRC校验码。总共此包是128+5=133字节。SOH  我的第一个包收到如下： 01 00 FF "YG06\_V0\_1.bin" NULL "11212" NULL[111] CRC CRC  接受者：发送大写字母ACK，发送大写字母C，等待数据  发送者：发送ASCII中的STX也就是02，后面是包序号01，包序号反码FE，1024字节数据，CRC CRC.总共此包是1024+5=1029字节。STX  我的第一个包收到如下： 02 01 FE DATA[1024] CRC CRC  接受者：ACK  发送者:STX包  接受者：ACK  发送者:STX包  .......  接受者：ACK  发送者:EOT  重复发送1024字节直到收到EOT开头数据，网上的错误点就在这里，搞了我很久，网上说的是不够128凑齐发SOH包，其实不是，是凑齐1024发STX包直到发EOT。  接受者：NAK ( 第一次收到EOT发NAK )  发送者:EOT  接受者：ACK C( 第二次收到EOT发ACK 和大写 C )  发送者:SOH 00 FF NUL[128] CRC CRC （发送128字节空包）  接受者：ACK ( 传输结束 )、  简明如下：  发送端 接收端  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< C  SOH 00 FF "YG06\_V0\_1.bin" NULL "11212" NULL[111] CRC CRC>>>>>>>>>>>>>  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ACK  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< C  STX 01 FE data[1024] CRC CRC>>>>>>>>>>>>>>>>>>>>>>>>  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ACK  STX 02 FD data[1024] CRC CRC>>>>>>>>>>>>>>>>>>>>>>>  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ACK  STX 03 FC data[1024] CRC CRC>>>>>>>>>>>>>>>>>>>>>>>  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ACK  STX 04 FB data[1024] CRC CRC>>>>>>>>>>>>>>>>>>>>>>>  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ACK  STX xx xx data[972] 1A[52] CRC CRC>>>>>>>>>>>>>>>>>> 补1A凑齐1024字节  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ACK  EOT >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< NAK  EOT>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ACK  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< C  SOH 00 FF NUL[128] CRC CRC >>>>>>>>>>>>>>>>>>>>>>>  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ACK 结束 |

##### 测试超级终端和区别

##### 常见错误

###### 1、超级终端出现错误，但是使用secureCRT又是正常的

超级终端中不知道是使用了小的数据包还是怎么回事，超级终端使用DMA的话，这个缓存区要很大。



解决

串口的环形缓存区不能够太小，否则丢失数据，通常情况下一个1K的数据包，至少需要1024+5个字节

#define USARTx\_RINGBUF\_SIZE (1024+6) 但是测试时候这样可以

|  |
| --- |
| #define USARTx\_RINGBUF\_SIZE 1050 |

###### 2、支持Ymodem协议的软件踩坑

搜过很多文章，大家都是用超级终端来传bin文件更新自己的stm32程序，也有用secureCRT软件的。

我在一开始是选择使用CRT，我刚开始接触Ymodem协议以为这两软件都可以用来升级，这是没错的，但其实还得看你的代码支持不支持。我浏览的很多文章大多协议代码都是匹配超级终端的。

当时以为这两者没区别，导致后面遇到的问题让我很迷惑。

CRT软件下的第一包帧头其实为STX。

而且有文章指出：

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

### 适配OTA升级部分

#### 云平台

阿里云：SDK可以下载，但是必须是RTOS系统

#### 阿里云

##### 获取C LINK SDK

<https://help.aliyun.com/document_detail/163755.html?spm=a2c4g.141434.0.0.4c109e44zTzHo8#topic2747>

##### 环境说明

C Link SDK可实现跨平台的移植，对开发环境要求灵活，只需支持以下条件：

开发语言：C99标准的C语言。

开发工具：不限。支持C语言编译即可。

编译框架：Makefile编译。

|  |
| --- |
|  |

##### 移植示例说明

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# 常见flash字节数字

**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

## stm32f407zgt6

ROM ： 起始地址0x08000000 0x080FFFFF

/\* 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 \*/

# OTA常见工具使用

## 超级终端



1、保证 LORA中带有boot程序

2、

## nodetools

192.168.100.253

# 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：



# 公司的IAP程序修改适配部分工作

## 公司的IAP 的BootLoader程序

#### wifi的boot程序修改过程

#### cat1的boot程序修改

##### flash\_if.h 对应

#define PARAMETER\_ADDRESS (uint32\_t)0x0801FC00 这个参数地址要和APP对应

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

##### command.h

typedef struct user\_data {

uint16\_t boot\_mode;

uint16\_t app\_en;

uint32\_t flag;

char mac[24]; //模块mac

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

}user\_data\_t;

##### 开启测试 记得改回去

###### main.h

#define USB\_TEST 1

#### NB的boot程序修改

#define PARAMETER\_ADDRESS (uint32\_t) 0x0800FE00

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

boot程序大小 16KB

NB-Serial-IAP.bin 31KB

参数区 末尾512B

完全足够

##### flash\_if.h 对应

#define PARAMETER\_ADDRESS (uint32\_t)0x0800FE00

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

##### command.h

typedef struct user\_data {

uint16\_t boot\_mode;

uint16\_t app\_en;

uint32\_t flag;

char id[40]; //ID

char key[120]; //KEY

char ip[64]; //

unsigned char nbandFlag;

int port;

int mode;

}user\_data\_t;

##### 开启测试 记得改回去

###### main.h

#define USB\_TEST 1

## IAP F103的APP程序

#### wifi的F103 APP程序修改过程

#### cat1的F103 APP程序修改过程

##### flash.h 对应

#define FLASH\_PARAM\_START\_ADDR 0x0801FC00 /\* Start @ of user Flash area \*/

#define FLASH\_PARAM\_END\_ADDR 0x0801FFFF /\* End @ of user Flash area \*/

##### rf\_if\_app.c 记录

t\_rf\_info rf\_info; //通讯模块信息

##### rf\_if\_app.h 更改

typedef struct {

uint16\_t boot\_mode;

uint16\_t app\_en;

uint32\_t flag;

char mac[24]; //模块mac

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

} t\_rf\_info;

##### board.c

#define USER\_APP\_BEGIN (uint32\_t)0x08004000

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

##### flash.h

#define BOOT\_IAP 0xA2

#define BOOT\_APP 0x3C

#define APP\_OK 0x4D

#define APP\_ERR 0xFF

##### at\_app.c

#ifdef CONFIG\_AT\_IAP

else if (strncmp(pat, "AT+IAP=", 7)==0){

ATOK();

if(strncmp(pat+7, "IAP", 3)==0) {

rf\_info.boot\_mode = BOOT\_IAP;

} else if(strncmp(pat+7, "APP", 3)==0) {

rf\_info.boot\_mode = BOOT\_APP;

}

rf\_info.app\_en = APP\_OK;

flash\_write(FLASH\_PARAM\_START\_ADDR, (uint16\_t \*)(&rf\_info), sizeof(t\_rf\_info) / 2);

rt\_thread\_mdelay(100);

//关闭所有中断

rt\_base\_t level;

level=rt\_hw\_interrupt\_disable();

SysTick->CTRL = 0;

SysTick->VAL = 0;

rt\_hw\_interrupt\_enable(level);

NVIC\_DisableIRQ(EXTI3\_IRQn);

NVIC\_DisableIRQ(EXTI4\_IRQn);

NVIC\_DisableIRQ(EXTI9\_5\_IRQn);

NVIC\_DisableIRQ(USART1\_IRQn);

NVIC\_DisableIRQ(USART2\_IRQn);

NVIC\_DisableIRQ(USART3\_IRQn);

//复位所有初始化过的外设

EXTI\_DeInit();

USART\_DeInit(USART1);

USART\_DeInit(USART2);

USART\_DeInit(USART3);

GPIO\_DeInit(GPIOA);

GPIO\_DeInit(GPIOB);

GPIO\_AFIODeInit();

//复位时钟

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_USART1, DISABLE);

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_USART2, DISABLE);

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_USART3, DISABLE);

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA, DISABLE);

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB, DISABLE);

RCC\_DeInit();

//系统复位

typedef void (\*pFunction)(void);

pFunction Jump\_To\_Application;

unsigned JumpAddress;

uint32\_t programe\_addr = 0x08000000;

JumpAddress = \*(volatile unsigned\*) (programe\_addr + 4);

Jump\_To\_Application = (pFunction) JumpAddress;

/\* Initialize user application's Stack Pointer \*/

\_\_set\_PSP(\*(volatile unsigned\*) programe\_addr);

\_\_set\_CONTROL(0);

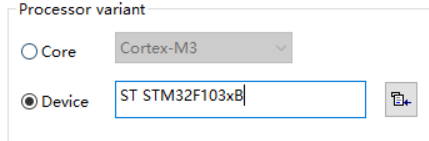
\_\_set\_MSP(\*(volatile unsigned\*) programe\_addr);

Jump\_To\_Application();

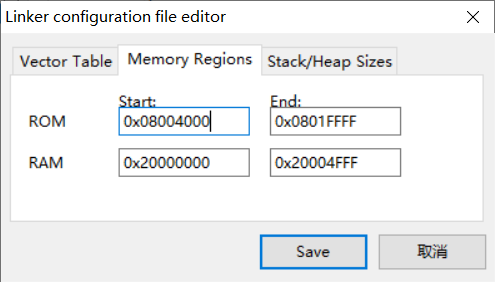
}

##### 工程配置

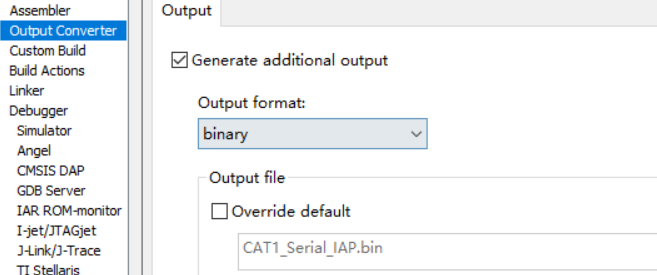
###### ST STM32F103xB



###### 配置起始地址



###### 配置bin生成

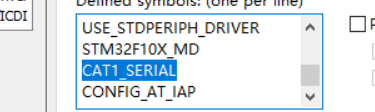


##### 开启测试 记得还原

###### 1、rt\_config.h

// #define RT\_USING\_UART3

###### 2、删除



删除

CAT1\_SERIAL

###### 3、rf\_if\_app.c 43行

#ifndef CAT1\_SERIAL /\*终端节点\*/

if(recv[0]=='{' && recv[length-1]=='}') { //control类数据

recv[length] = 0;

// zxbee\_onrecv\_fun(recv, length);

}

#endif

###### 4、rf\_if\_app.c 241行

// #ifndef CAT1\_SERIAL /\*终端节点\*/

// sensorLinkOn();

// #endif

#### NB的F103 APP程序修改过程

##### 参数地址对应

###### config.h

#define PARAMETER\_ADDRESS (uint32\_t)0x0800FE00

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

#define BOOT\_IAP 0xA2

#define BOOT\_APP 0x3C

#define APP\_OK 0x4D

#define APP\_ERR 0xFF

下面和boot程序对应

typedef struct {

uint16\_t boot\_mode;

uint16\_t app\_en;

uint32\_t flag;

char id[40]; //ID

char key[120]; //KEY

char ip[64]; //

unsigned char nbandFlag;

int port;

int mode;

}nb\_config\_t;

##### 修改函数

###### config.c

int config\_init(void)

{

nb\_config\_t temp\_param;

STMFLASH\_Read(PARAMETER\_ADDRESS, (u16\*)&temp\_param, (sizeof(nb\_config\_t)+1)>>1);

if (temp\_param.flag != 0xa5a5a5a5) { //未初始化

nbConfig.boot\_mode = BOOT\_APP;

nbConfig.app\_en = APP\_OK;

nbConfig.flag = 0xa5a5a5a5;

return 0;

} else {

nbConfig = temp\_param;

return 1;

}

}

void config\_save(void)

{

nbConfig.flag = 0xa5a5a5a5;

STMFLASH\_Write(PARAMETER\_ADDRESS, (u16\*)&nbConfig, (sizeof(nbConfig)+1)>>1);

}

###### contiki-main.c

#include "misc.h"

main函数添加

#ifdef USER\_APP\_BEGIN

NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH,(USER\_APP\_BEGIN - NVIC\_VectTab\_FLASH));

#endif

###### at.c

extern void clock\_delay\_ms(unsigned int ms);

else if (memcmp(p\_msg, "AT+IAP=", 7)==0){

at\_response(ATOK);

if(memcmp(p\_msg+7, "IAP", 3)==0) {

nbConfig.boot\_mode = BOOT\_IAP;

} else if(memcmp(p\_msg+7, "APP", 3)==0) {

nbConfig.boot\_mode = BOOT\_APP;

}

nbConfig.app\_en = APP\_OK;

config\_save();

clock\_delay\_ms(100);

//关闭所有中断

NVIC\_DisableIRQ(SysTick\_IRQn);

NVIC\_DisableIRQ(EXTI3\_IRQn);

NVIC\_DisableIRQ(EXTI4\_IRQn);

NVIC\_DisableIRQ(EXTI9\_5\_IRQn);

NVIC\_DisableIRQ(USART1\_IRQn);

NVIC\_DisableIRQ(USART2\_IRQn);

NVIC\_DisableIRQ(USART3\_IRQn);

SysTick->CTRL = 0;

SysTick->VAL = 0;

//复位所有初始化过的外设

EXTI\_DeInit();

USART\_DeInit(USART1);

USART\_DeInit(USART2);

USART\_DeInit(USART3);

GPIO\_DeInit(GPIOA);

GPIO\_DeInit(GPIOB);

GPIO\_AFIODeInit();

//复位时钟

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_USART1, DISABLE);

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_USART2, DISABLE);

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_USART3, DISABLE);

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA, DISABLE);

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB, DISABLE);

RCC\_DeInit();

//系统复位

typedef void (\*pFunction)(void);

pFunction Jump\_To\_Application;

unsigned JumpAddress;

uint32\_t programe\_addr = 0x08000000;

JumpAddress = \*(volatile unsigned\*) (programe\_addr + 4);

Jump\_To\_Application = (pFunction) JumpAddress;

/\* Initialize user application's Stack Pointer \*/

\_\_set\_PSP(\*(volatile unsigned\*) programe\_addr);

\_\_set\_CONTROL(0);

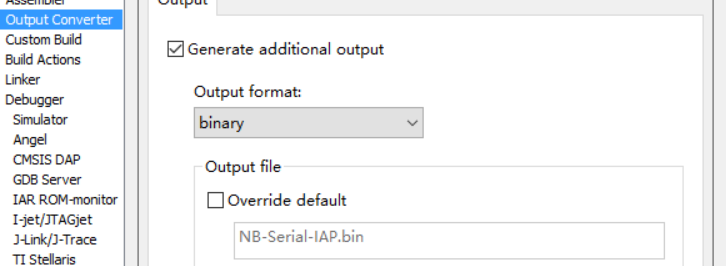
\_\_set\_MSP(\*(volatile unsigned\*) programe\_addr);

Jump\_To\_Application();

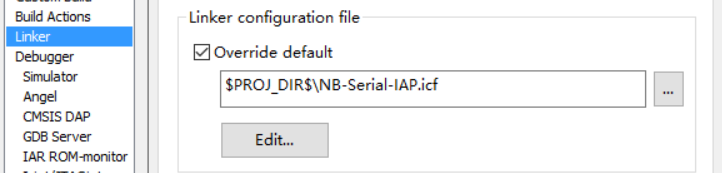
}

##### 工程配置

###### 配置bin生成



###### 配置起始地址



###### 添加宏定义



USER\_APP\_BEGIN=0x08004000

## STM32F407工程

## 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