**OTA 使用说明**

目录

[1. DFU与OTA的关系及简介 3](#_Toc43820240)

[2. OTA基础知识介绍 3](#_Toc43820241)

[2.1 BootLoader与APP介绍 3](#_Toc43820242)

[2.2 FLASH分区 3](#_Toc43820243)

[2.3 程序执行流程 5](#_Toc43820244)

[3. 代码实现 6](#_Toc43820245)

[3.1 BootLoader的 编写 6](#_Toc43820246)

[3.2 APP代码实现 11](#_Toc43820247)

[3.2.1 在蓝牙工程中添加以下几个函数： 11](#_Toc43820248)

[3.2.2 给自己的蓝牙添加相应的特征服务值与读写函数 14](#_Toc43820249)

[3.2.3 OTA与UUID128匹配 15](#_Toc43820250)

[3.2.4 给蓝牙添加相应的读写操作 15](#_Toc43820251)

[3.2.5 修改中断向量表 17](#_Toc43820252)

[3.2.6 修改FLASH的起始地址 17](#_Toc43820253)

[4. OTA升级操作 18](#_Toc43820254)

[4.1 MM32W系列蓝牙代码构架 18](#_Toc43820255)

[4.2 编译bootloader与APP代码 18](#_Toc43820256)

[4.3 Hex文件下载 20](#_Toc43820257)

[4.4 生成ota升级文件 21](#_Toc43820258)

[4.5 OTA升级 22](#_Toc43820259)

# DFU与OTA的关系及简介

如何实现BLE OTA？什么叫DFU？什么叫OTA？DFU与OTA有什么联系呢？怎么保证升级的安全性？本文主要给大家介绍MM32W0xx\_n系列的蓝牙OTA升级的基础知识，主要针对OTA升级的原理，分区的划分，代码的编写以及实验验证这几个方面给大家讲解。

所谓DFU（Device Firmware Update），就是设备固件升级的意思，而OTA（Over The Air）空中下载技术是实现DFU的一种方式而已，准确说，OTA的全称应该是OTA DFU，即通过空中无线方式实现设备固件升级。只不过大家为了方便起见，直接用OTA来指代固件空中升级（有时候大家也将OTA称为FOTA，即Firmware OTA，这种称呼意思更明了一些）。所谓的DFU分为后台式DFU与非后台式DFU。所谓的手机与电脑的升级就是后台式DFU，后台式DFU又称为静默式DFU,在升级的时候新固件会在后台悄悄的下载，系统的软件在下载新固件的时候还是可以正常使用的，当固件下载完成以后系统会进入BootLoader模式，然后BootLoader会将新的固件写入到老固件的位置，将老的固件覆盖掉，非后台式的DFU在升级的时候，系统必须先进入BootLoader模式来下载固件，下载完成以后BootLoader来完成新固件的更新操作，在整个升级的过程中系统是不能正常使用的。

DFU也分为双区DFU与单区DFU,双区与单区式新固件与老固件覆盖的两种方式，后台是必须采用双分区的形式，因为如果采用单区的形式会影响系统在固件下载过程中的正常使用，一般老的固件与新的固件各占一半，固件下载完成以后会进行软件的校验，校验完成以后进入BootLoader模式，然后将Bank\_B的代码搬运到Bank\_A区，单区DFU就是系统判断进入升级模式的时候进入BootLoader模式，然后BootLoader会擦出Bank\_A的代码，当下载完成以后会进行校验，当校验完成以后启动应用程序，否则升级不成功需要重新升级。单分区DFU适用于flash比较紧张的情况。如果升级不成功很容易遇到变砖的情况。双区的DFU模式虽然会浪费内存空间，但是不会遇到变砖的情况，会给客户带来更好的用户体验。

# OTA基础知识介绍

## BootLoader与APP介绍

BootLoader：所谓BootLoader就是烧录引导程序，他的作用就是在进行升级的时候对APP程序进行引导存放，通俗来讲就是把APP程序存放到指定的位置，然后来启动APP程序。

APP：用户需要执行的程序。使用BootLoader进行引导烧录到指定的位置并运行，让系统执行用户所需要的操作。

## FLASH分区

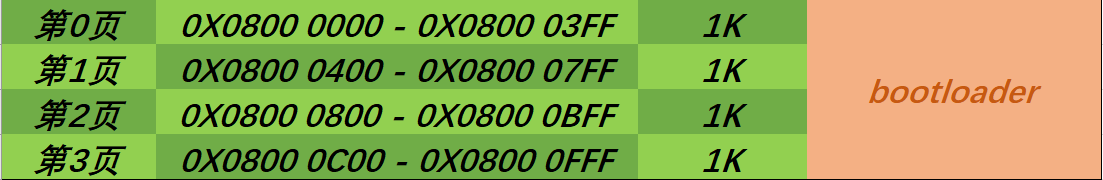
MM32W0xx\_n系列蓝牙的内存空间由64位宽的存储单元组成，既可存放代码又可以存放数据。主闪存块按128页（每页1K字节）或者32个扇区（每个扇区4K字节）分块，并在进行写保护的时候以扇区为单位进行设置。FLASH地址从0x08000000开始。



以上图的128KFLASH为例，将FLASH分为三个区：BootLoader区，Bank\_A区，Bank\_B区（程序备份区）具体的划分如下：

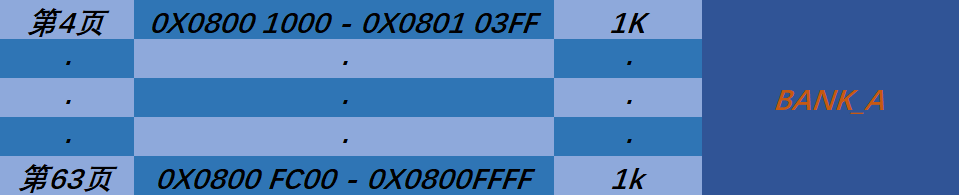
Bootloader区

08000000------08000FFF 4kB



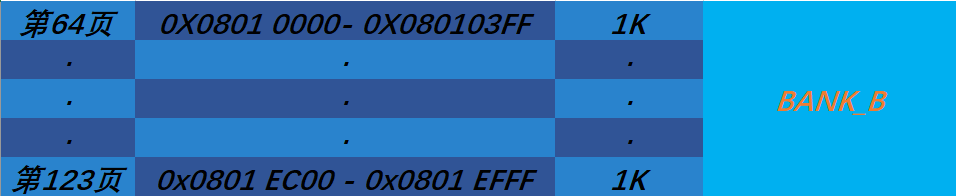
Bank\_A区

08001000------0800FFFF 60KB



Bank\_B区

08010000------0801EFFF 60KB

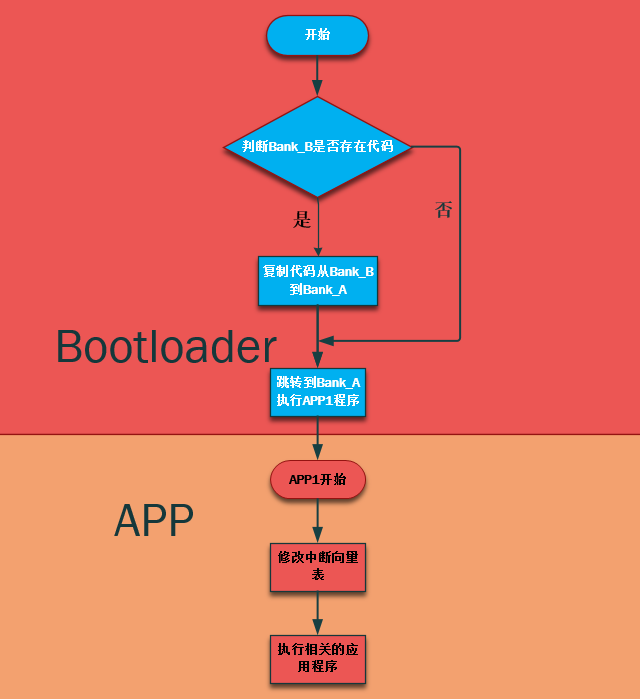


## 程序执行流程

程序会先执行BootLoader程序，程序会去检查Bank\_B是否存在代码，如果不存在则直接跳转到Bank\_A的APP程序执行。

在执行Bank\_A区的程序过程中，因为APP与BootLoader这两个代码的中断向量表不同，所以在跳转到Bank\_A区的APP的程序的时候，第一步就是需要更改程序的中断向量表，然后再去执行其他的应用程序。

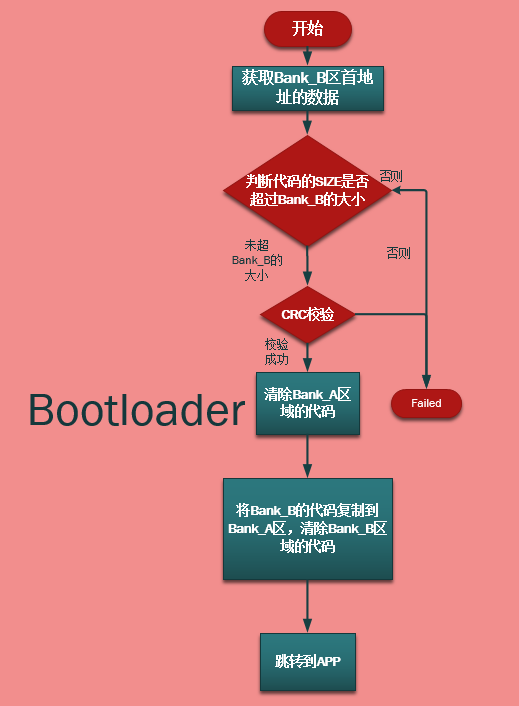
在程序中加入OTA升级的部分，然后进行OTA升级，手机端文件传输完毕以后会对系统进行重启然后进入BootLoader模式，在此模式下会对Bank\_B区的代码进行检测，如果没有代码就跳转到Bank\_A区执行APP程序，如果Bank\_B区存在代码则对Bank\_B区的代码进行相应的校验，校验成功就进行代码区的操作，将Bank\_B区的代码复制到Bank\_A，操作成功以后跳转到Bank\_A执行相应的APP程序。



# 代码实现

## BootLoader的 编写

MM32W0xx\_n的第一个地址的前两个字节代表需要升级的代码的大小，第三第四个字节为CRC校验的数值，升级的过程中，手机端先与蓝牙数据通信，数据通信完毕以后蓝牙会重新启动然后进入BootLoader模式，在进入BootLoader模式以后会判断B区是否存在代码，如果存在代码会对代码的大小进行判断并进行CRC校验，校验完成以后会将B区的代码复制到A区，然后再将B区的代码清除并跳转到APP区。



检测相应的代码区是否存在代码以及对代码的大小与code区进行判断，并进行CRC校验。

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

\*\*function: CheckCodeBank

\*\*@brief This function is Check if there is code in CODE area and verify

\*\*

\*\*@param BANK\_ID\_Flag : Code area selection

\*\*

\*\*@return 0/1 : CRC check returns 1 correctly, otherwise returns 0

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

int CheckCodeBank(int BANK\_ID\_Flag)

{

uint16\_t Size,Crc = 0;

uint8\_t\* BankAddr = (uint8\_t\*)(BANK\_BASE + BANK\_A\_OFFSET); //BANKA

if(BANK\_ID\_Flag == BANK\_ID\_B)

{

BankAddr = (uint8\_t\*)(BANK\_BASE + BANK\_B\_OFFSET);

}

Size = BankAddr[1]; Size <<= 8; Size |= BankAddr[0];

if(Size > BANK\_B\_SIZE\_KB\_MAX \* 1024 - 16)

{

Size = BANK\_B\_SIZE\_KB\_MAX \* 1024 - 16;

}

Crc = CRC16(BankAddr+16,(int)Size,Crc);

if(((Crc & 0xFF) != BankAddr[2]) || (((Crc>>8) & 0xFF) != BankAddr[3]))

{

return 0; //CRC failed

}

return 1; //OK

}

将B区的代码复制到A区。

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

\*\*function: CopyCode\_B\_to\_A

\*\*@brief This function is Copy the code of area B to area A

\*\*

\*\*@param None.

\*\*

\*\*@return None.

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

void CopyCode\_B\_to\_A(void)

{

uint16\_t i,Size;

uint32\_t BankAddrA = BANK\_BASE + BANK\_A\_OFFSET,Data;

unsigned char\* addr = (unsigned char\*)(BANK\_BASE + BANK\_B\_OFFSET);

Size = addr[1]; Size <<= 8; Size |= addr[0];

Size >>= 2;

Size += 4;//header 16 bytes

if(Size > BANK\_B\_SIZE\_KB\_MAX \* 256)

{

Size = BANK\_B\_SIZE\_KB\_MAX \* 256;

}

ClearBank(BANK\_ID\_A);

FLASH\_Unlock();

for(i = 0 ; i < Size ; i ++)

{

Data = addr[3]; Data <<= 8;

Data |= addr[2]; Data <<= 8;

Data |= addr[1]; Data <<= 8;

Data |= addr[0];

FLASH\_ProgramWord(BankAddrA, Data);

BankAddrA += 4;

addr += 4;

}

FLASH\_Lock();

if(!CheckCodeBank(BANK\_ID\_A))

{

}

}

APP跳转函数

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

\*\*function: Jump\_App

\*\*@brief This function is a jump function, jump from BOOT to APP.

\*\*

\*\*@param None.

\*\*

\*\*@return None.

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

void Jump\_App()

{

JumpAddress = \*(\_\_IO uint32\_t\*) (APPLICATION\_ADDRESS + 4);

Jump\_To\_Application = (pFunction) JumpAddress;

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

\_\_set\_MSP(\*(\_\_IO uint32\_t\*) APPLICATION\_ADDRESS);

/\* Jump to application \*/

Jump\_To\_Application();

}

int main(void)

{

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_SYSCFG, ENABLE);

if(CheckCodeBank(BANK\_ID\_B))

{

CopyCode\_B\_to\_A();

}

ClearBank(BANK\_ID\_B);

Jump\_App();

while (1)

{

}

}

## APP代码实现

给自己的蓝牙工程添加OTA功能具体的操作如下：

### 在蓝牙工程中添加以下几个函数：

中断向量表重映射函数

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

\*\*function: CodeNvcRemap

\*\*@brief APPLICATION ADDRESS remapping function, adjust the starting address of APP

\*\*

\*\*@param None.

\*\*

\*\*@return None.

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

void CodeNvcRemap(void)

{

uint32\_t i = 0;

for(i = 0; i < 48; i++) {

\*((uint32\_t\*)(0x20000000 + (i << 2))) = \*(\_\_IO uint32\_t\*)(APPLICATION\_ADDRESS + (i << 2));

}

RCC->APB2ENR |= 0x00000001;

SYSCFG->CFGR |= 0x03;

}

u32 GetOtaAddr(void)

{

return FLASH\_E2PROM\_ADDR\_OTA;

}

获取数据存放FLASH的地址

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

\*\*function: GetCodeAddr

\*\*@brief Get the address where the code is written to flash

\*\*

\*\*@param None.

\*\*

\*\*@return None.

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

u32 GetCodeAddr(void)

{

return (FLASH\_E2PROM\_ADDR\_BASE + FLASH\_BOOT\_ROM\_SIZE);

}

写数据到FLASH操作函数

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

\*\*function: WriteFlashE2PROM

\*\*@brief MM32 write flash function, write data to the corresponding flash area

\*\*

\*\*@param data : Storage address of data written to flash

\*\*

\*\*@param len : Storage length of data written to flash

\*\*

\*\*@param pos : flash storage data address

\*\*

\*\*@param flag : flash unlock flag

\*\*

\*\*@return None.

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

void WriteFlashE2PROM(u8\* data, u16 len, u32 pos, u8 flag) //4 bytes aligned

{

u32 t;

if(flag)FLASH\_Unlock();

while(len >= 4) {

t = data[3]; t <<= 8;

t |= data[2]; t <<= 8;

t |= data[1]; t <<= 8;

t |= data[0];

FLASH\_ProgramWord(pos, t);

pos += 4;

len -= 4;

data += 4;

}

if(flag)FLASH\_Lock();

}

单片机重启函数

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

\*\*function: OtaSystemReboot

\*\*@brief MM32 interrupt reset function.

\*\*

\*\*@param None.

\*\*

\*\*@return None.

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

void OtaSystemReboot(void)//porting api

{

NVIC\_SystemReset();

}

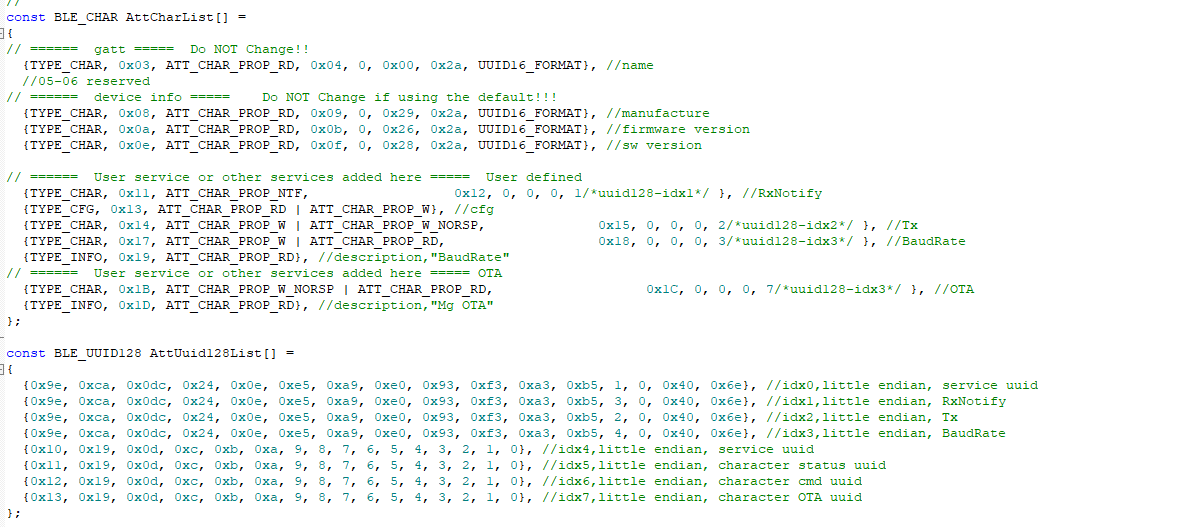
### 给自己的蓝牙添加相应的特征服务值与读写函数

{TYPE\_CHAR, 0x1B, ATT\_CHAR\_PROP\_W\_NORSP | ATT\_CHAR\_PROP\_RD, 0x1C,0,0,0,7/\*uuid128-idx3\*/ }, //OTA

{TYPE\_INFO,0x1D,ATT\_CHAR\_PROP\_RD}, //description,"Mg OTA"

{0x10,0x19,0x0d,0xc,0xb,0xa, 9, 8,7,6,5,4,3,2,1,0}, //idx0,little endian, service uuid

{0x13,0x19,0x0d,0xc,0xb,0xa,9,8,7,6,5,4,3,2,1,0},//idx3,little endian,character OTA uuid



给OTA配置相应的UUID，这两个的UUID是固定的，否则将会导致OTA升级不成功。

### OTA与UUID128匹配

在void att\_server\_rdByGrType(u8 pdu\_type, u8 attOpcode, u16 st\_hd, u16 end\_hd, u16 att\_type)函数中对OTA的特征值与UUID进行匹配：

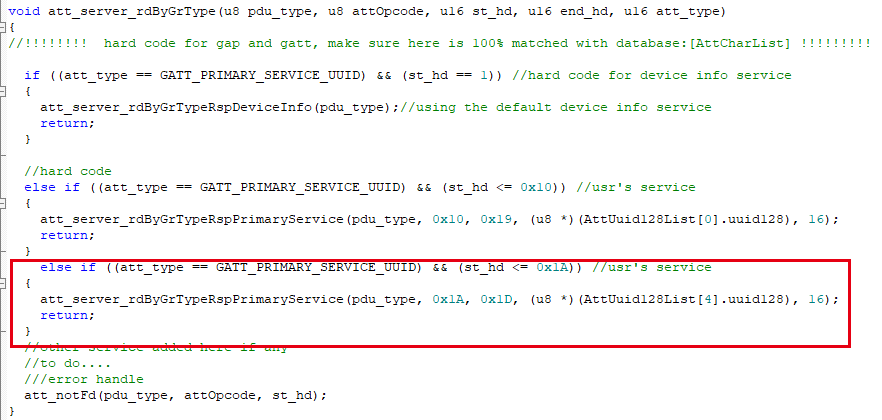
else if ((att\_type == GATT\_PRIMARY\_SERVICE\_UUID) && (st\_hd <= 0x1A)) //usr's service

{

att\_server\_rdByGrTypeRspPrimaryService(pdu\_type, 0x1A, 0x1D, (u8 \*)(AttUuid128List[4].uuid128), 16);

return;

}



### 给蓝牙添加相应的读写操作

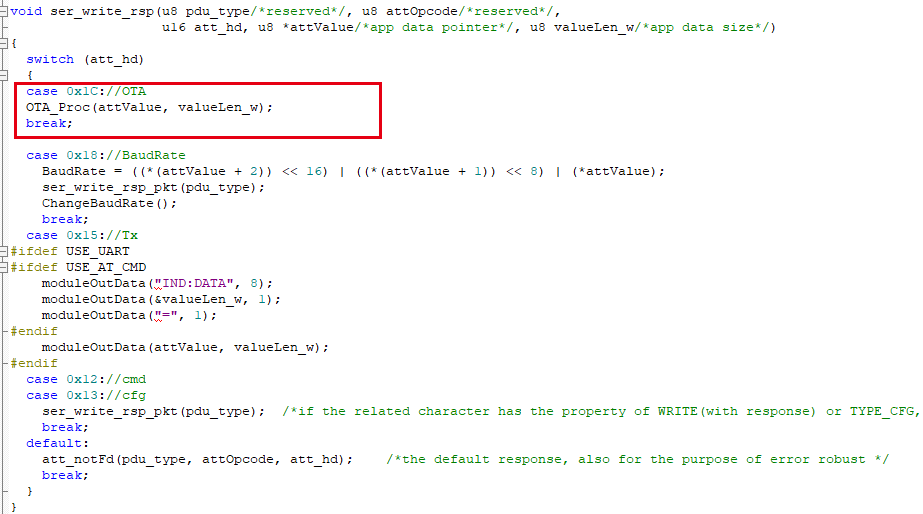
在void ser\_write\_rsp(u8 pdu\_type/\*reserved\*/, u8 attOpcode/\*reserved\*/,

u16 att\_hd, u8 \*attValue/\*app data pointer\*/, u8 valueLen\_w/\*app data size\*/)函数中添加以下内容：

case 0x1C://OTA

OTA\_Proc(attValue, valueLen\_w);

break;



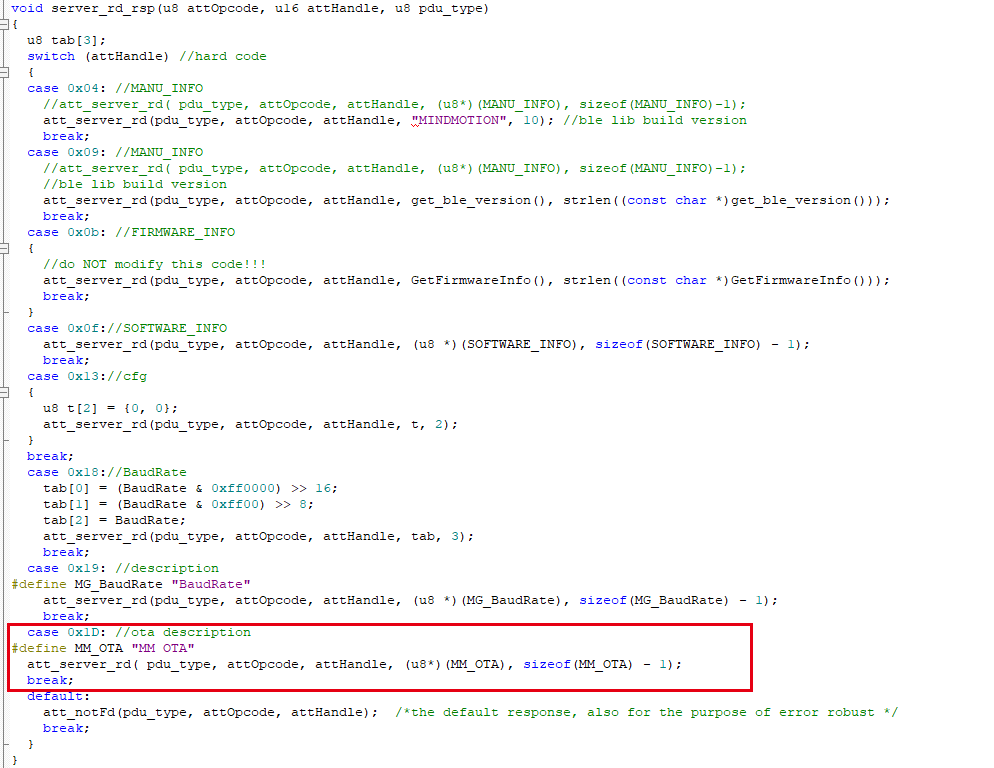
在void server\_rd\_rsp(u8 attOpcode, u16 attHandle, u8 pdu\_type)函数中添加如下操作函数。

case 0x1D: //ota description

#define MM\_OTA "MM OTA"

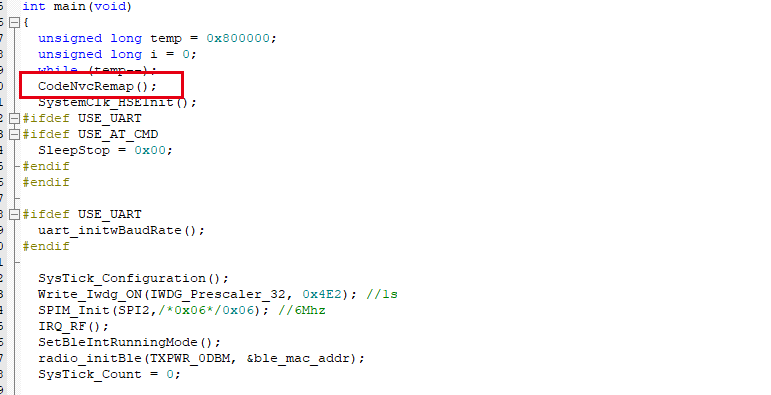
att\_server\_rd(pdu\_type,attOpcode,attHandle,(u8\*)(MM\_OTA),sizeof(MM\_OTA) - 1);

break;



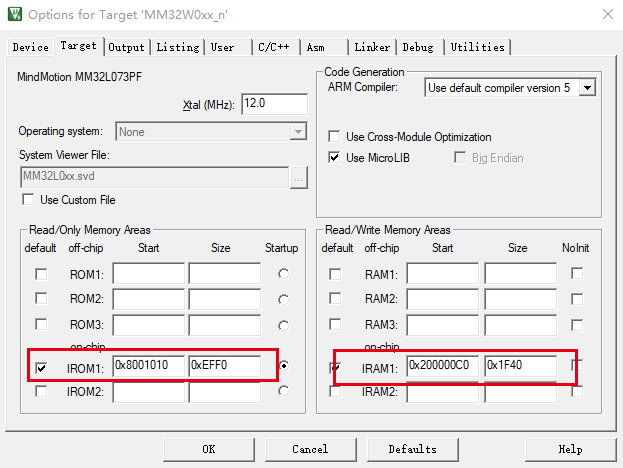
### 修改中断向量表

在主函数刚开始的时候调用CodeNvcRemap()函数来进行中断向量表地址的映射。



### 修改FLASH的起始地址

具体操作如下图：

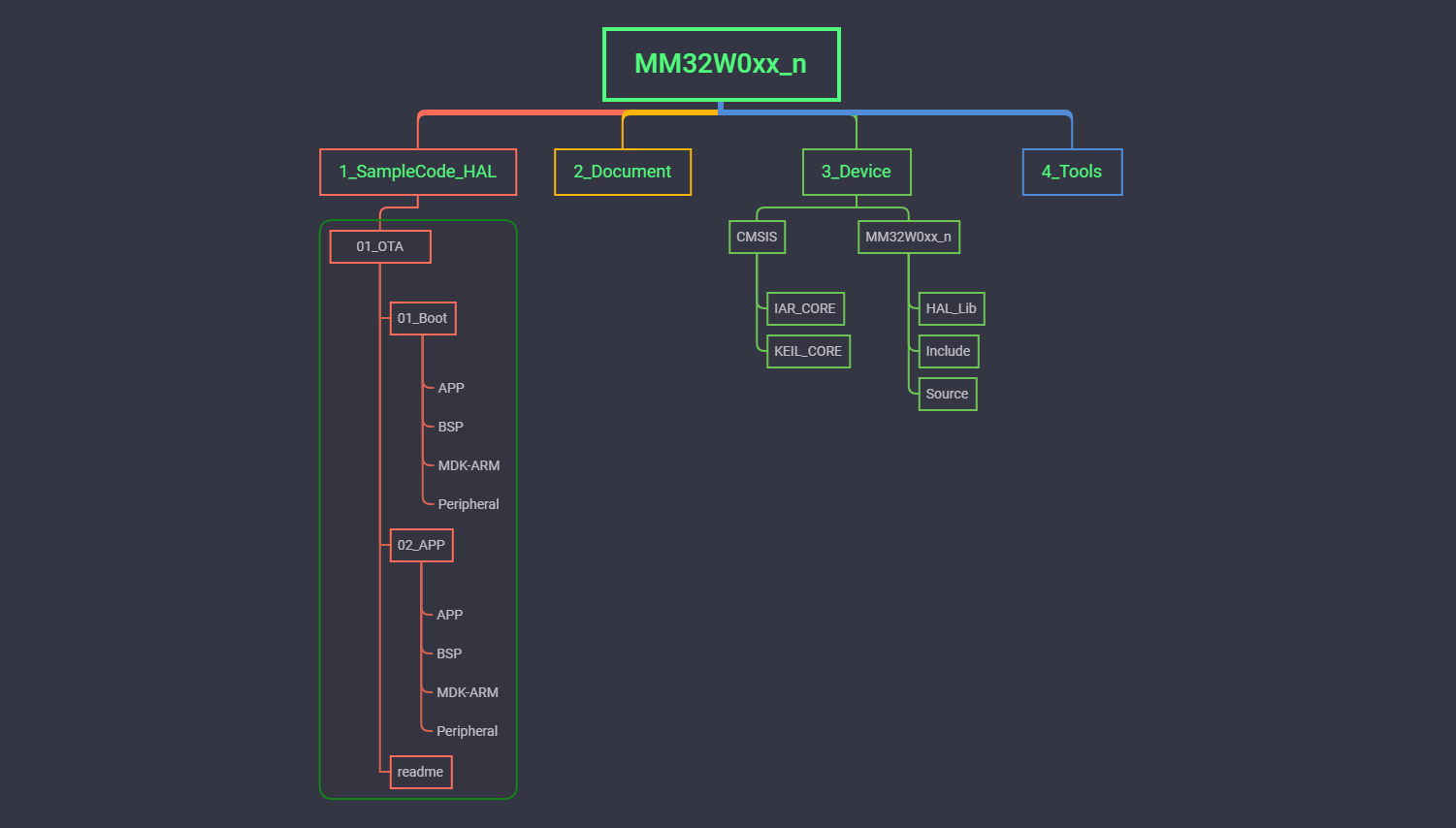


到此处为止我们的蓝牙代码已经成功添加了OTA功能。

# OTA升级操作

具体的OTA升级操作的步骤以官方例程MM32W0xx\_n为例。

## MM32W系列蓝牙代码构架



1\_SampleCode\_HAL存放的是基于MM32W0xx\_n的相关的例程，也包括OTA升级。

2\_Document文件夹下面存放的是相关的技术文档。

3\_Device文件夹下面存放的是相关的HAL库文件以及相关的启动文件。

4\_Tools文件夹下面存放的是相关的工具。

## 编译bootloader与APP代码

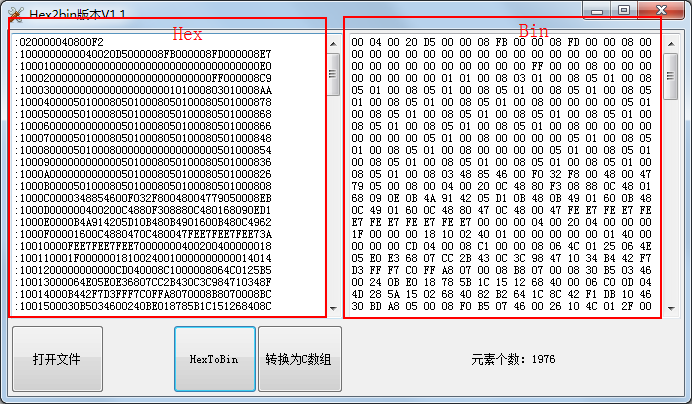
打开1\_SampleCode\_HAL 文件夹，然后进入OTA工程，我们会发现OTA分为两套代码，一套为Bootloader一套为APP代码, 编译两套代码生成.hex文件与.bin文件，具体的操作方法有两种：

1. 使用keil自带的FORMORT生成.bin文件，具体的配置如下：

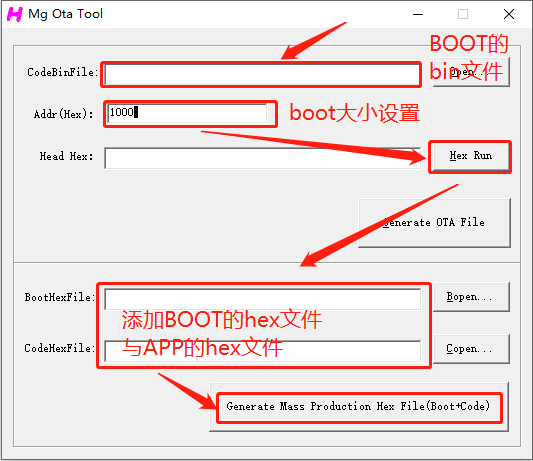
点击User栏，将Run#1打勾，并在后面的方框中填入生成.bin的路径。我的路径为：fromelf --bin --output .\Objects\OTA\_APP.bin .\Objects\OTA\_APP.axf我们在进行配置的时候需要注意空格，fromelf(一个空格) --bin(一个空格) --output(一个空格) .\Objects\OTA\_APP.bin(一个空格) .\Objects\OTA\_APP.axf

1. 使用hex文件转bin文件工具。

在完成APP 和BOOT 的程序编译后，需要将Code生成的Hex 文件转换成对应的Bin 文件，方便将两部分Code下载到两段不同的Flash存储空间中。Hex转Bin 可通过Keil MDK 来生成，也可以通过第三方软件来生成。



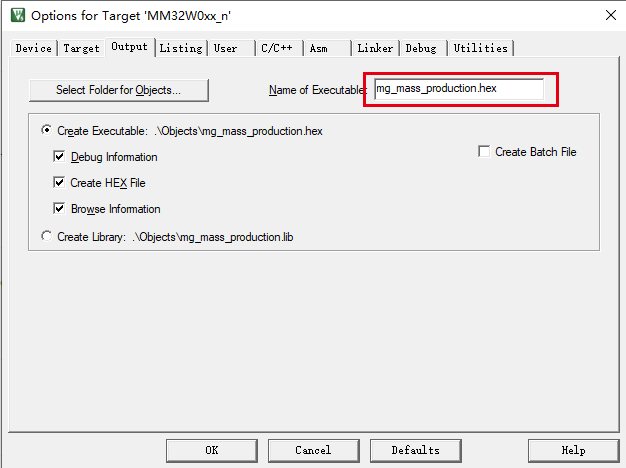
将BOOT的hex与APP的hex进行合并生成可执行的hex文件。



我们会看到文件夹下面会有一个mg\_mass\_production.hex文件。

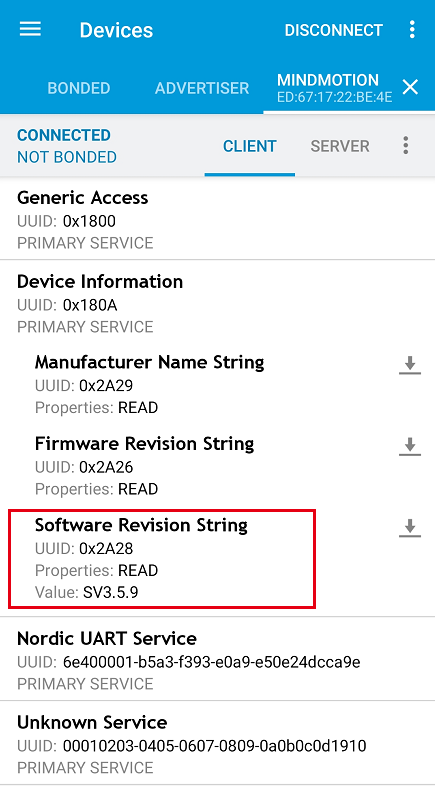
## Hex文件下载

我们将这个文件存放到APP代码的生成hex文件的路径下面，然后修改keil的配置（注意需要添加.hex尾缀）



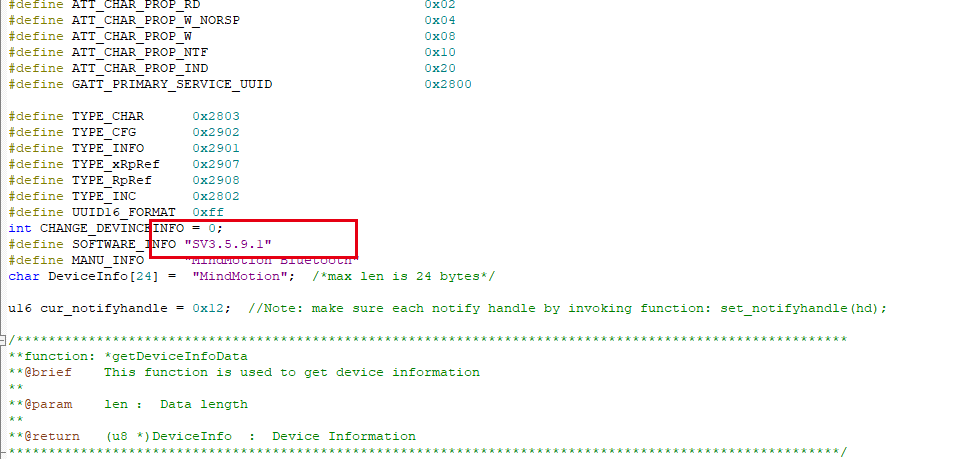
然后直接点击下载按钮，注意不要编译直接下载。

芯片重新上电用手机BLE软件搜索广播“MINDMOTION”并进行连接，我们可以看到我们目前的蓝牙的软件版本为3.5.9。

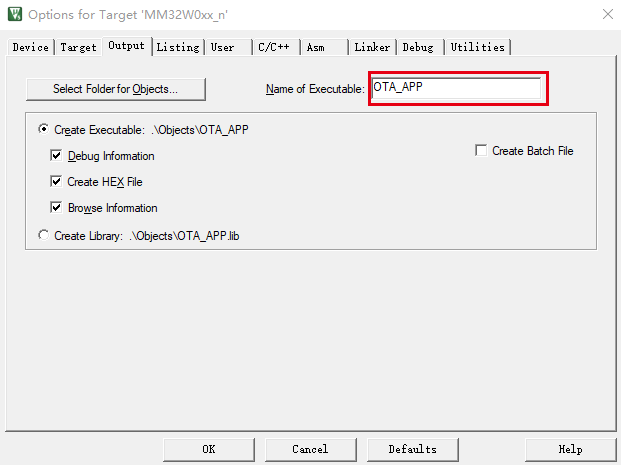


## 生成ota升级文件

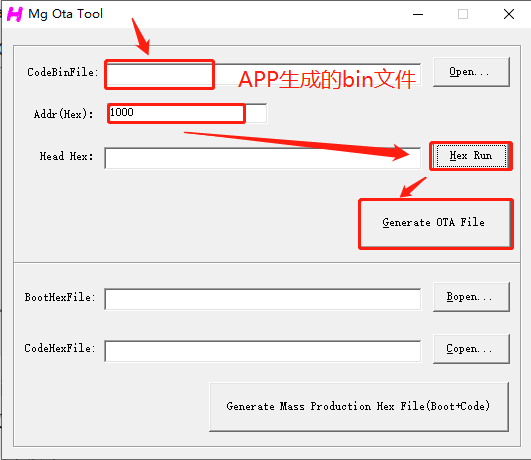
我们修改蓝牙的软件版本为3.5.9.1。



修改keil生成文件的配置



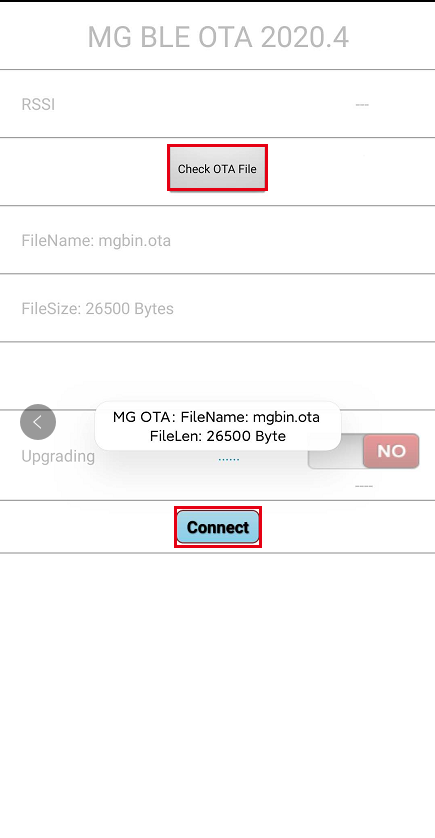
然后编译生成Bin文件，并将生成的Bin文件转化生成为ota文件，生成的文件的名字为mg\_bin.ota。



我们需要将名字修改为我们需要的mgbin.ota。

## OTA升级

将修改以后的ota升级文件保存到手机的根目录下面，下面就可以打开手机的OTA软件进行OTA升级了。



点击Chack OTA Files软件自动搜索手机根目录出的ota升级文件，点击Connect按钮来搜索并选择我们需要进行升级的蓝牙的名字进行连接。然后点击开关来进行自动升级。升级完成以后使用NRF来查看版本信息为3.5.9.1，版本正确，说明我们的OTA升级成功。

