# 主从一体
# 串口透传
# 教程

**www.AmoMcu.com**

**阿莫单片机 出品**

**2014-08-22 v2.4**

目录

1，要实现的功能......4
2，开发环境......5
2.1 硬件......5
2.2 软件......5
3，源码位置......5
4，源码分析......5
4.1 工程来源......5
4.2 工程修改要点......6
4.2.1 实现主从一体代码......7
4.2.2 实现数据掉电保存......10
4.2.3 实现串口代码......11
4.2.4 实现 AT 命令......13
4.2.5 实现主机连接......14
4.2.6 增加特征值 CHAR6......15
4.2.7 主机发送数据......19
4.2.8 主机接收数据......19
4.2.9 从机发送数据......20
4.2.10 从机接收数据......20
4.3 主从一体公共文件主要函数分析......20
4.3.1 主从一体公共头文件 simpleBLE.h......20
4.3.2 主从一体公共头文件 simpleBLE.c......23
4.3.3 主设备文件......26
4.3.4 从设备文件......26
5，源码编译......27
6，下载运行......28
7，测试......28
7.1 双机主从一体串口透传......28
7.2 用 ios 的 lightblue 透传......31
7.3 用 AmoMcu.apk 的测试......32
8，联系我们......33
9，附录　AT 命令......33
9.1　AT 测试指令......33
9.2　AT+BAUD 查询、设置串口波特率......33
9.3　AT+PARI 设置串口校验......34
9.4　AT+STOP 设置串口停止位......34

9.5 AT+MODE 设置模块工作模式......35
9.6 AT+NAME 查询、设置设备名称......35
9.7 AT+RENEW 恢复默认设置(Renew)......35
9.8 AT+RESET 模块复位，重启(Reset)......35
9.9 AT+ROLE 查询、设置主从模式......35
9.10 AT+PASS 查询、设置配对密码......36
9.11 AT+TYPE 设置模块鉴权工作类型......36
9.12 AT+ADDR 查询本机 MAC 地址......36
9.13 AT+CONNL 连接最后一次连接成功的从设备......36
9.14 AT+CON 连接指定蓝牙地址的从设备......36
9.15 AT+CLEAR 清除主设备配对信息......37
9.16 AT+RADD 查询成功连接过的从机地址......37
9.17 AT+VERS 查询软件版本......37
9.18 AT+TCON 设置主模式下尝试连接时间......37
9.19 AT+RSSI 读取 RSSI 信号值......37
9.20 AT+TXPW 改变模块发射信号强度......38
9.21 AT+TIBE 改变模块作为 ibeacon 基站广播时间间隔......38
9.22 AT+IMME 设置工作类型......38

# 1，要实现的功能

【1】实现一份代码，编译一个固件，下载到芯片后， 通过 AT 命令或者按键切换来实现主机或从机功能（目前仅实现了 AT 命令切换），目前，市场上商用的模块绝大部分均为主从一体的， 既方便生产，也方便客户使用。

【2】LED 状态灯显示， 用 P0.6 实现。

LED 状态灯已实现。

状态灯修改如下：

连接前：

主机，未记录从机地址时，每秒亮 100ms；

主机，记录从机地址时，每秒亮 900ms；

从机，每 2 秒亮 1 秒。

连线后：

主机与从机均为每 5 秒亮 100 毫秒。（闪亮， 以便省电）

【3】 用 iphone4s+上的 lightblue（第三方 app，我们无源码可在 app store 免费下载免费使用）和 AmoMcu 我们出品的 AmoMcu.apk（有源码）均可测试。

【4】 透传期间每个数据包不宜超过 120 字节，波特率越高，发包间隔要求越长。无线蓝牙透数据传均存在丢包率的问题，所以用户朋友们一定要做好应用层的数据校验和丢包重传。

【5】 所有 AT 命令 都是以 “\r\n”结尾。大部分 AT 命令都是在未连接前有效，连接后所有数据均为透传（有若干条 AT 指令在透传时也有效， 例如 AT+RSSI\r\n 查询 RSSI 信号强度的）。

【6】 iBeacon 简单已集成。 通过 AT 命令 “AT+MODE2\r\n” 来设置， 只有在从机下才可以。

工作模式 0:透传 ， 1: 直驱（保留）， 2: iBeacon

iBeacon 可苹果应用商店上的免费应用 Locate iBeacons 来测试，目前显示距离，使用方法可见我们以前的 blog ： http://blog.csdn.net/mzy202/article/details/20365691

【7】 有数据掉电保存功能， AT 命令的设置绝大部分为调电保存的。

【8】 按住按键开机可恢复出厂设置。按键的 io 口为 P0.7， P0.7 已启用内部上来功能， 所以按键应设计成通过 1K 电阻接地。

【9】 可设置连接密码并绑定。

【10】 其他的一些列参数也可设置， 详见附录的 AT 命令。

# 2，开发环境

## 2.1 硬件

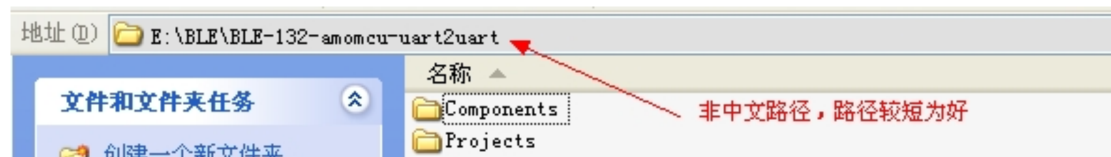
1、　SmartRF 系列开发板 2 块，核心板 CC2540 （或者 CC2541）
2、　CC-Debugger 仿真器
3、　MiniUSB 线

## 2.2 软件

1、　ble 协议栈，版本：1.3.2
2、　IAR for 8051 开发环境，版本：8.10
3、　Flash Programmer 固件烧写软件。
4、　串口调试助手，我们使用的是 SSCOM3.2 ， 打开两个。

# 3，源码位置

AmoMcu 提供的这一份源码是一个独立工程，不依赖与其他任何文件，理论上可放在任何地方进行编译和下载，但作为中国程序员需要有一个好习惯， 最好把这个工程放到不带中文且比较短的路径，否则有可能编译出错， 产生不必要的麻烦，例如我们推荐的路径如下所示：

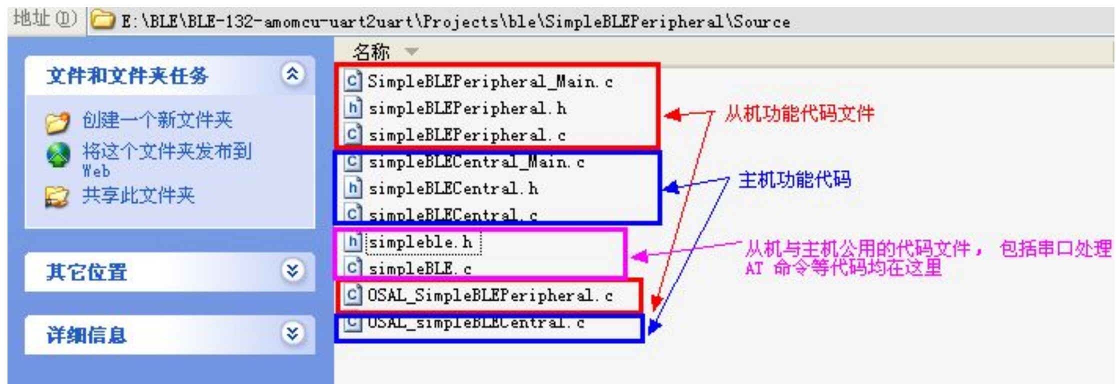



# 4，源码分析

## 4.1 工程来源

该工程修改自 TI BLE 1.3.2 的 peripheral 从机工程， 在基础上， 我们进行添加了主机功能， 并实现启动时选择启动从机或主机程序功能。

主要的目录结构如下：




## 4.2 工程修改要点

注： 如果想了解下这个工程是如何来的， 可以看一下以下的这一段， 不想看的话可以跳过。

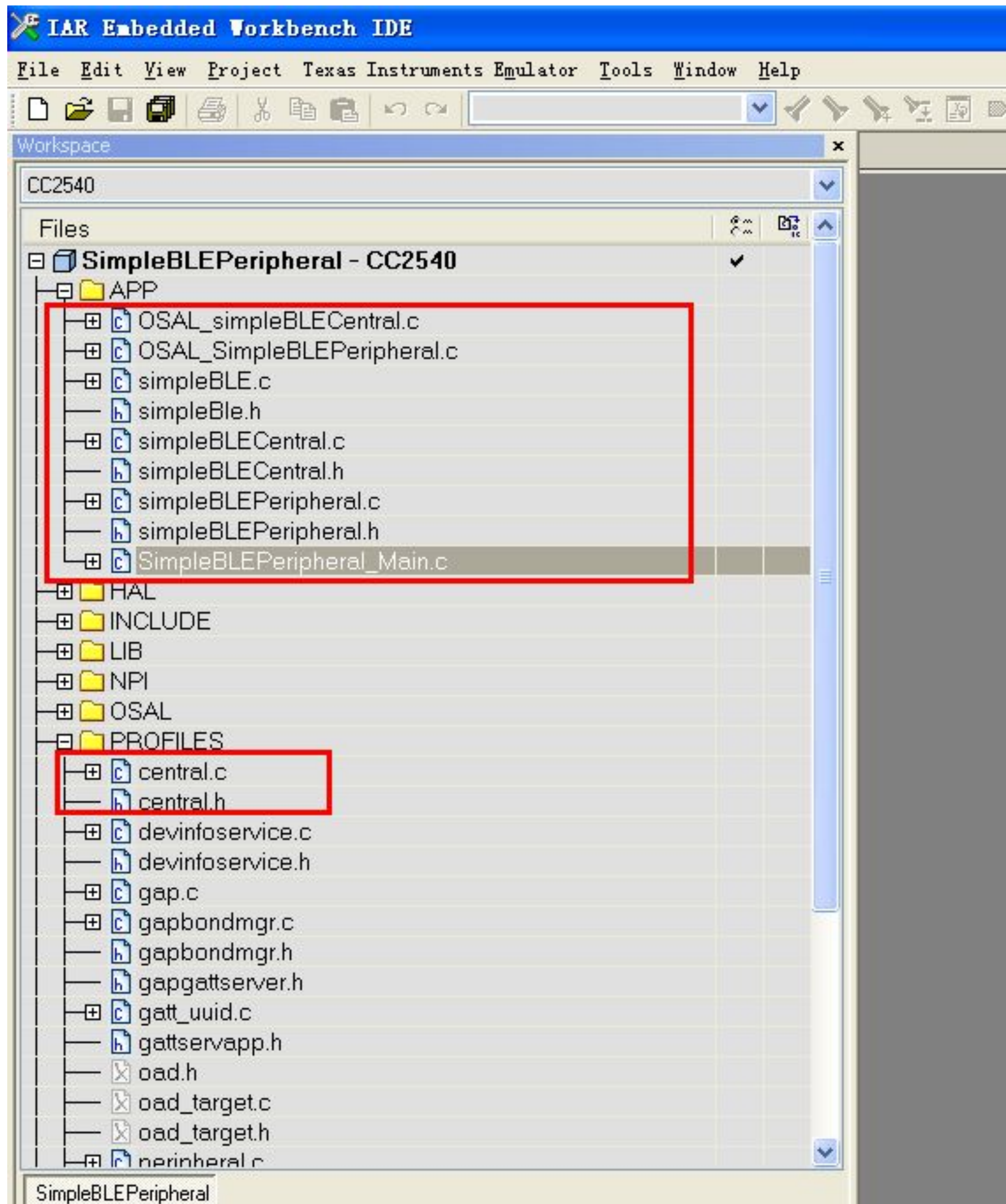
【1】 在 peripheral 从机工程 （默认安装 ble 1.3.2 协议栈的话， 其原路径在：C:\Texas Instruments\BLE-CC254x-1.3.2\Projects\ble\SimpleBLEPeripheral）的基础上增加主机工程， 其中主机工程的源码也是现成的（默认安装 ble 1.3.2 协议栈的话， 其原路径在：C:\Texas Instruments\BLE-CC254x-1.3.2\Projects\ble\SimpleBLECentral ） ， 把 C:\Texas Instruments\BLE-CC254x-1.3.2\Projects\ble\SimpleBLECentral\Source 目录下的以下三个文件复制到从机工程 的 SimpleBLEPeripheral\Source 目录中。

【2】 我们以 cc2540 为例（cc2541 同样的操作即可）

打开 ...\SimpleBLEPeripheral\CC2540DB\SimpleBLEPeripheral.eww 工程，

然后增加刚才从主机工程那里复制进来的三个文件（以及我们编写的 simpleBLE.c 与 simpleBLE.h）， 并且在 profile 中再增加 central.c 与 central.h ，以便我们使用主机功能。

增加完成后， 如图所示：




## 4.2.1 实现主从一体代码

我们直接来分析一下整体的执行流程。

在 SimpleBLEPeripheral_Main.c 的 main 函数中，

```
00086:    /* Initialize NV system */
00087:    osal_snv_init();
00088:
00089: #if 1
00090: //这一段代码和说明是 amomcu 增加的
00091: /*
00092:   从设置中读出以保存的数据， 以便决定现在应该是跑主机还是从机
00093:   注意， 这里用到了 osal_snv_xxx ， 数据是存在flash里边的， 大家可以找找相关
00094:   代码和说明,需要注意的是 osal_snv_read 和 osal_snv_write ,
00095:   第一个 参数 osalSnvId_t id, 这个id， 我们编程可用的是从 0x80 至 0xff,
00096:   其中目前程序中可用的空间是 2048 字节
00097:   这个大小定义于 osal_snv.c 中， 即以下宏定义， 跟踪代码进去就能看到
00098: */
00099: /*
00100: // NV page configuration
00101: #define OSAL_NV_PAGE_SIZE       HAL_FLASH_PAGE_SIZE
00102: #define OSAL_NV_PAGES_USED      HAL_NV_PAGE_CNT
00103: #define OSAL_NV_PAGE_BEG        HAL_NV_PAGE_BEG
00104: #define OSAL_NV_PAGE_END       (OSAL_NV_PAGE_BEG + OSAL_NV_PAGES_USED - 1)
00105: */
00106:
00107:     {
00108:         int8 ret8 = osal_snv_read(0x80, sizeof(SYS_CONFIG), &sys_config);
00109:         // 如果该段内存未曾写入过数据， 直接读，会返回 NV_OPER_FAILED ,
00110:         // 我们利用这个特点作为第一次烧录后的运行， 从而设置参数的出厂设置
00111:         if(NV_OPER_FAILED == ret8)
00112:         {
00113:             simpleBLE_SetAllParaDefault(PARA_ALL_FACTORY);
00114:             simpleBLE_WriteAllDataToFlash();
00115:         }
00116:
00117:         // 执行  串口初始化
00118:         simpleBLE_NPI_init();
00119:     }
00120: #endif//这一段代码和说明是 amomcu 增加的
00121:
```

108 行 至 115 行， 是增加的用于先读取存贮到 nv flash 中的数据，如果该段内存未曾写入过数据， 直接读，会返回 NV_OPER_FAILED，我们利用这个特点作为第一次烧录后的运行， 从而设置参数的出厂设置。

118 行 ， 执行 串口初始化。

```
00124:    /* Initialize the operating system */
00125:    osal_init_system(sys_config.role);
00126:
00127:    /* Enable interrupts */
00128:    HAL_ENABLE_INTERRUPTS();
00129:
00130:    // Final board initialization
00131:    InitBoard( OB_READY );
00132:
00133:    #if defined ( POWER_SAVING )
00134:      osal_pwrmgr_device( PWRMGR_BATTERY );
00135:    #endif
00136:
00137:    /* Start OSAL */
00138:    osal_start_system(); // No Return from here
00139:
00140:    return 0;
00141: } ? end main ?
```

125 行， 是在原来的 osal_init_system 函数中增加了一个参数 sys_config.role， 这个 sys_config.role 的值 为 ：

BLE_ROLE_PERIPHERAL = 0,　　　　//从机角色

或

BLE_ROLE_CENTRAL = 1,　　　　//主机角色

我们深入到 OSAL.c 的 uint8 osal_init_system( uint8 Role )

函数中， 有下面图中这样的调用：

```
01041: //参数 Role //1: 主机,  0: 从机
01042: uint8 osal_init_system( uint8 Role )
01043: {
01044:   // Initialize the Memory Allocation System
01045:   osal_mem_init();
01046:
01047:   // Initialize the message queue
01048:   osal_qHead = NULL;
01049:
01050:   // Initialize the timers
01051:   osalTimerInit();
01052:
01053:   // Initialize the Power Management System
01054:   osal_pwrmgr_init();
01055:
01056:   // Initialize the system tasks.
01057: #if 0
01058:     osalInitTasks();
01059: #else
01060:   if(Role == 1)
01061:   {
01062:     osalInitCentralTasks();
01063:   }
01064:   else
01065:   {
01066:     osalInitPeripheralTasks();
01067:   }
01068: #endif
01069:
01070:   // Setup efficient search for the first free block of heap.
01071:   osal_mem_kick();
01072:
01073:   return ( SUCCESS );
```

1060 行至 1067 行，就是根据传入的参数来决定执行主机代码或从机代码。以上就是我们主从提一体的主要修改点。

另外， 在上图的红色方框内的两个函数基本内容其实是一样的， 我们下面来分析一下从机流程：

```
00088: const pTaskEventHandlerFn tasksArr_peripheral[] =
00089: {
00090:   LL_ProcessEvent,                                               // task 0
00091:   Hal_ProcessEvent,                                              // task 1
00092:   HCI_ProcessEvent,                                              // task 2
00093: #if defined ( OSAL_CBTIMER_NUM_TASKS )
00094:   OSAL_CBTIMER_PROCESS_EVENT( osal_CbTimerProcessEvent ),           // task 3
00095: #endif
00096:   L2CAP_ProcessEvent,                                            // task 4
00097:   GAP_ProcessEvent,                                              // task 5
00098:   GATT_ProcessEvent,                                             // task 6
00099:   SM_ProcessEvent,                                               // task 7
00100:   GAPRole_ProcessEvent,                                          // task 8
00101:   GAPBondMgr_ProcessEvent,                                       // task 9
00102:   GATTServApp_ProcessEvent,                                      // task 10
00103:   SimpleBLEPeripheral_ProcessEvent                               // task 11
00104: };
00105:
00106: uint8 tasksCnt;
00107: uint16 *tasksEvents;
00108: pTaskEventHandlerFn tasksArr[12];
```

88 行至 104 行， 定义了各个人物的处理函数， 这些任何函数， 就是 osal 所谓的“多任务”处理函数，其中 103 行的 SimpleBLEPeripheral_ProcessEvent 就是我们的从机的应用任务处理函数。

106 行至 108 行，如果我们对照一下 TI 官方分出的从机参考代码的话， 就会发现，TI 定义成了 const 的常量， 意思是不可改变的，我们主机相对应的代码中把它重用了， 这样的话， 我们就不许要再定义一次吗节省内存空间， 详见下图：

```
00084: const pTaskEventHandlerFn tasksArr_central[] =
00085: {
00086:   LL_ProcessEvent,
00087:   Hal_ProcessEvent,
00088:   HCI_ProcessEvent,
00089: #if defined ( OSAL_CBTIMER_NUM_TASKS )
00090:   OSAL_CBTIMER_PROCESS_EVENT( osal_CbTimerProcessEvent ),
00091: #endif
00092:   L2CAP_ProcessEvent,
00093:   GAP_ProcessEvent,
00094:   GATT_ProcessEvent,
00095:   SM_ProcessEvent,
00096:   GAPCentralRole_ProcessEvent,
00097:   GAPBondMgr_ProcessEvent,
00098:   GATTServApp_ProcessEvent,
00099:   SimpleBLECentral_ProcessEvent
00100: };
00101:
00102: extern uint8 tasksCnt;
00103: extern uint16 *tasksEvents;
00104: extern pTaskEventHandlerFn tasksArr[];
```

下面来看一下 osalInitPeripheralTasks 这个函数：

```
00123: void osalInitPeripheralTasks( void )
00124: {
00125:   uint8 taskID = 0;
00126:
00127:   tasksCnt = sizeof( tasksArr_peripheral ) / sizeof( tasksArr_peripheral[0
00128:   osal_memcpy(tasksArr, tasksArr_peripheral, sizeof( tasksArr_peripheral)
00129:
00130:   tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
00131:   osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));
00132:
00133:   /* LL Task */
00134:   LL_Init( taskID++ );
00135:
00136:   /* Hal Task */
00137:   Hal_Init( taskID++ );
00138:
00139:   /* HCI Task */
00140:   HCI_Init( taskID++ );
```

127 行至 131 行， 这里对任务个数经行了赋值， 并且开辟了任务的事件内存。
接下来是逐个任务的初始化。

```
00160:   /* Profiles */
00161:   GAPRole_Init( taskID++ );
00162:   GAPBondMgr_Init( taskID++ );
00163:
00164:   GATTServApp_Init( taskID++ );
00165:
00166:   /* Application */
00167:   SimpleBLEPeripheral_Init( taskID );
00168: } ? end osalInitPeripheralTasks ?
```

167 行， 是对应用任务的初始化，

## 4.2.2 实现数据掉电保存

需要保存的数据为一个结构体，详见：

以下函数实现结构体 sys_config 的读数据，我们是在开机阶段进行了读 flash。
osal_snv_read(0x80, sizeof(SYS_CONFIG), &sys_config);

以下函数实现结构体 sys_config 的写数据，我们是在需要保存数据时进行写 flash。在 simpleBLE.c 文件中，有些数据的函数：

```
00110: // 保存所有数据到nv flash
00111: void simpleBLE_WriteAllDataToFlash()
00112: {
00113:     osal_snv_write(0x80, sizeof(SYS_CONFIG), &sys_config);    // 写所有参数
00114: }
```

113 行的 0x80 是可写的 flash 的应用层可用的开始 id，这个开始 id 的依据，可以在文件 bcomdef.h 的下面图中获得：

```
00166: // Device NV Items -    Range 0 - 0x1F
00167: #define BLE_NVID_IRK                     0x02  //!< The Device's IRK
00168: #define BLE_NVID_CSRK                    0x03  //!< The Device's CSRK
00169: #define BLE_NVID_SIGNCOUNTER             0x04  //!< The Device's Sign Counter
00170:
00171: // Bonding NV Items -   Range  0x20 - 0x5F    - This allows for 10 bondings
00172: #define BLE_NVID_GAP_BOND_START           0x20  //!< Start of the GAP Bond Mana
00173: #define BLE_NVID_GAP_BOND_END             0x5f  //!< End of the GAP Bond Manage
00174:
00175: // GATT Configuration NV Items - Range  0x70 - 0x79 - This must match the number
00176: #define BLE_NVID_GATT_CFG_START           0x70  //!< Start of the GATT Configur
00177: #define BLE_NVID_GATT_CFG_END             0x79  //!< End of the GATT Configurat
00178: /** @} End BLE_NV_IDS */
```

172 行至 173 行定义了绑定的数据存储开始和结束 id

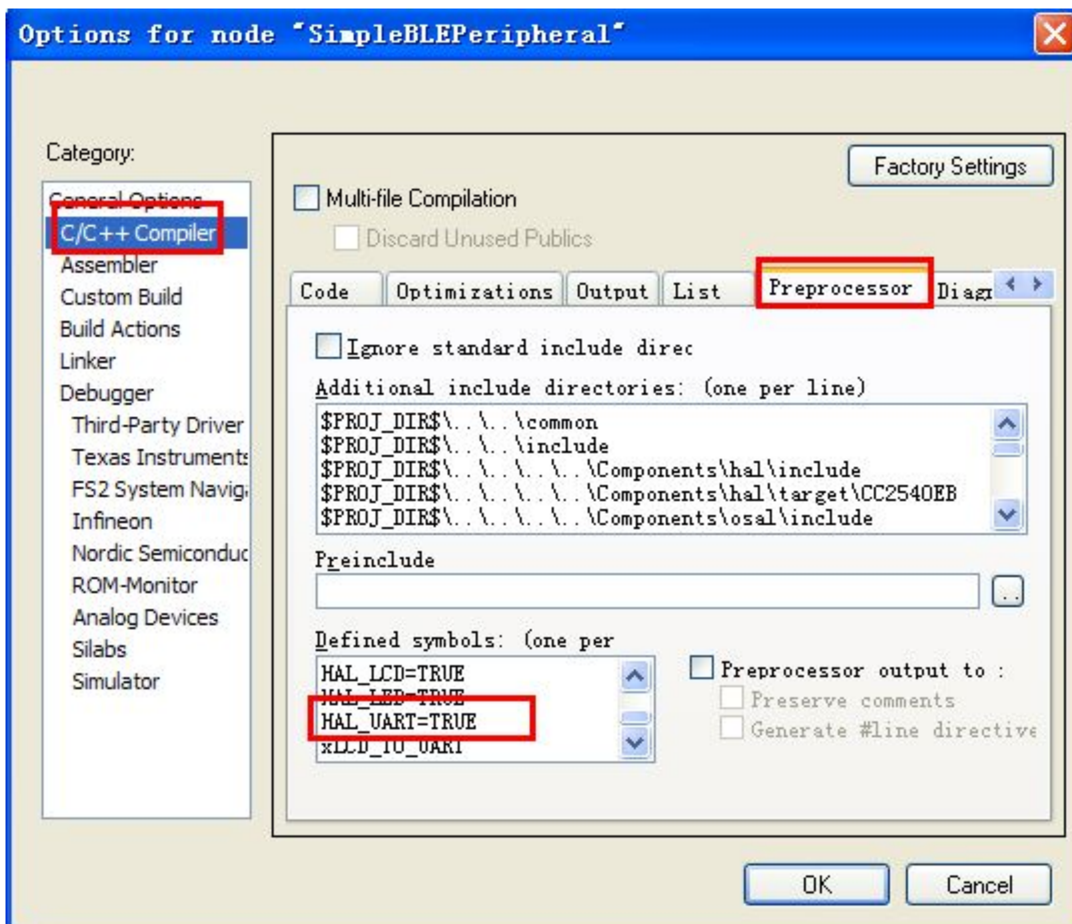
176 行至 177 行，鼎医疗 GATT 配置文件的开始和结束 id。

所以，我们可以用的 id 应该是从 0x7A 开始，我们的代码中以 0x80 开始存贮。

## 4.2.3 实现串口代码

串口的实现，TI 在 BLE1.3.2 协议栈的例程里边，已经封装了串口代码，要在从机中使用串口，我们只需要 2 步

1，　增加宏　HAL_UART=TRUE




2, 初始化串口函数， 并增加回调函数

在我们增加的 simpleBLE.c 文件中，有如下函数进行串口初始化：

```
00399: // 串行口 uart 初始化
00400: void simpleBLE_NPI_init(void)
00401: {
00402: #if 1
00403:     NPI_InitTransportEx(simpleBLE_NpiSerialCallback, sys_config.baudrate,
00404:         sys_config.parity, sys_config.stopbit );
00405: #else
00406:     NPI_InitTransport(simpleBLE_NpiSerialCallback);
00407: #endif
00408:
00409:     // 开机打印主机还是从机
00410:     if(GetBleRole() == BLE_ROLE_CENTRAL)
00411:     {
00412:         NPI_WriteTransport("Hello World Central\r\n",21);
00413:     }
00414:     else
00415:     {
00416:         NPI_WriteTransport("Hello World Peripheral\r\n",24);
00417:     }
00418: }
```

400 行， 我们的串口初始化函数中，有参数：波特率、奇偶校验和停止位的初始化。跟踪到 _hal_uart_dma.c 文件的 static void HalUARTOpenDMA(halUARTCfg_t *config)函数中， 即可发现我们增加的 奇偶校验和停止位 的代码， 如下：

```
00552: #if 1// amomcu.com 修改, 增加 奇偶校验  与 停止位
00553:   /*
00554:     Para 范围 0,1,2
00555:     0: 无校验
00556:     1: EVEN
00557:     2: ODD
00558:     Default: 0
00559:   */
00560:     if(config->parity == 1)
00561:     {
00562:         UxUCR |= UCR_PARITY;
00563:     }
00564:     else if(config->parity == 2)
00565:     {
00566:         UxUCR |= (UCR_PARITY | UCR_D9);
00567:     }
00568:
00569:
00570:   /*
00571:     Para: 0~1
00572:     0: 1 停止位
00573:     1: 2 停止位
00574:     Default: 0
00575:   */
00576:     if(config->stopbit == 1)
00577:     {
00578:         UxUCR |= UCR_SPB;
00579:     }
00580: #endif
```

回到我们增加的 simpleBLE.c 文件中，有如下串口回调函数：

static void simpleBLE_NpiSerialCallback( uint8 port, uint8 events )

下面把该回调函数里实现的功能讲解一下：

当串口收到数据后，就会马上调用以下回调函数，在实际测试中发现，此回调函数调用频繁（如果跟踪到底层驱动中就会发现， 只要收到一个字节， 都会调用一次该回调函数），如果你不执行 NPI_ReadTransport 函数进行读取， 那么这个回调函数就会频繁地被执行，但是，你通过串口发送一段数据， 你本意是想处理这一完整一段的数据，所以，我们在下面引入了时间的处理方法， 也即接收的数据够多或者超时，就读取一次数据， 然后根据当前的状态决定执行，如果没有连接上，就把所有数据当做 AT 命令处理， 如果连接上了，就把数据送到对端。

## 4.2.4 实现 AT 命令

为了便于外部 MCU 的控制， 我们增加了 AT 命令， AT 命令通过串口进行发送，同时，我们参考在工作中使用过各种模块，大对数的 AT 命令，使用“\r\n”即十六进制的 0x0D 0x0A 来作为 AT 命令的结束参考，这样比较方便。

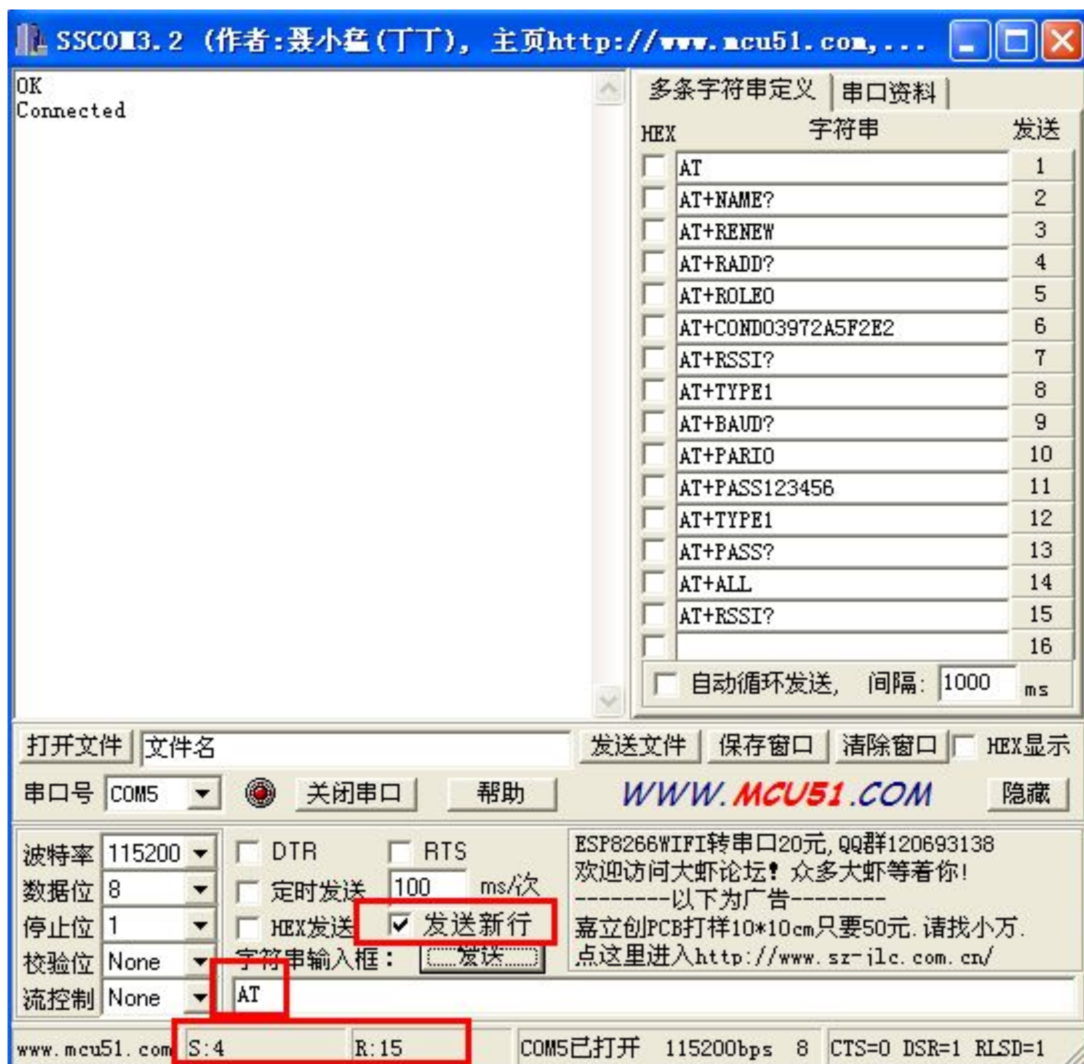
在我们的代码中， 总共实现了超过 20 条的 AT 命令，涵盖串口参数设置、设备名称修改、蓝牙广播参数设置、蓝牙角色设置以及蓝牙连接设置等。这些 AT 命令详见附录 9。下面我们来看一下代码， 在我们增加的 simpleBLE.c 文件中，有如下函数进行 AT 命令的处理：

// AT 命令处理

bool simpleBLE_AT_CMD_Handle(uint8 *pBuffer, uint16 length)

如果 AT 命令正确被执行， 那么会返回 附录 9 中的定义值， 否则一律返回 “ERROR\r\n”，特别需要注意的是每一条 AT 命令后面都必须带着“\r\n”， 下面我们以串口助手示例看看

如何发送一个 “AT\r\n” 指令：




如图红色框所示， 点击“发送”按键后， 将会发送“AT\r\n”测试 AT 命令的指令， 正确执行， 会返回 “OK\r\n”， 我们也可以看到， 是发送了 4 个字节。

## 4.2.5 实现主机连接

在主机代码文件 simpleBLECentral.c 中，我们添加了自动连接从机的代码，当然，这个自动连接是有条件的， 这个条件就是 AT 命令的设置， 我们来先看代码位置， 如下：

在函数 static void simpleBLECentralEventCB( gapCentralRoleEvent_t *pEvent ) 中：

case GAP_DEVICE_DISCOVERY_EVENT:

执行的是从机发现事件， 这里能够得到每一个发现了得从机的地址，

地址存放于 simpleBLEDevList[i].addr ，

simpleBLEScanRes 则是扫面到的从机个数。

```
00869:             else if(g_Central_connect_cmd  == BLE_CENTRAL_CONNECT_CMD_CONNL
00870:             || g_Central_connect_cmd  == BLE_CENTRAL_CONNECT_CMD_CONN )
00871:             {//连接指定最后成功的从机
00872:               // 连接从机
00873:               if(simpleBLE_GetToConnectFlag(Addr))
00874:               {
00875:                 uint8 tempstr[13] = {0};
00876:                 osal_memcpy(tempstr, Addr, 12);
00877:                 LCD_WRITE_STRING_VALUE( "ConAddr ", 0, 10, HAL_LCD_LINE_2 )
00878:                 LCD_WRITE_STRING( (char *)tempstr, HAL_LCD_LINE_2 );
00879:
00880:                 // 连接指定的mac 地址的从机设备
00881:                 //simpleBLEScanIdx = 0;
00882:                 for(i = 0; i < simpleBLEScanRes; i++)
00883:                 {
00884:                     LCD_WRITE_STRING_VALUE( "peerAddr ", i, 10, HAL_LCD_LIN
00885:                     LCD_WRITE_STRING( bdAddr2Str( simpleBLEDevList[i].add
00886:
00887:                     // connect to current device in scan result
00888:                     peerAddr = simpleBLEDevList[i].addr;
00889:                     addrType = simpleBLEDevList[i].addrType;
00890:
00891:                     if(osal_memcmp(Addr, bdAddr2Str( simpleBLEDevList[i].a
00892:                     {
00893:                       //simpleBLEScanIdx = i;
00894:                       ifDoConnect = TRUE;
00895:                       break;
00896:                     }
00897:                 }
00898:               } ? end if simpleBLE_GetToConnec... ?
00899:             } ? end if g_Central_connect_cmd... ?
```

873 行， 先判断一下， 是否可以去连接， 如果可以的话， 需要判断需要去连接的从机地址是否在已被发现的从机列表中， 如果在列表中， 则使 ifDoConnect = TRUE;

```
00931:               if(ifDoConnect)
00932:               {
00933:                   simpleBLEState = BLE_STATE_CONNECTING;
00934:
00935:                  LCD_WRITE_STRING( "Connecting", HAL_LCD_LINE_1 );
00936:                  LCD_WRITE_STRING( bdAddr2Str( pEvent->linkCmpl.devAddr ), H
00937:
00938:                  GAPCentralRole_EstablishLink( DEFAULT_LINK_HIGH_DUTY_CYCLE,
00939:                                                DEFAULT_LINK_WHITE_LIST,
00940:                                                addrType, peerAddr );
00941:               }
00942:               else
00943:               {
00944:                 LCD_WRITE_STRING( "Continue scanning", HAL_LCD_LINE_1 );
00945:
00946:                 // 继续扫描
00947:                 simpleBLEScanning  = FALSE;
00948:                 //simpleBLEScanRes = 0;
00949:                 //simpleBLEStartScan();
00950:               }
00951:             } ? end if simpleBLEState==BLE_S... ?
```

938 行， 是执行对从机的连接。

这里只是讲一下代码流程，如果要真机实操，请看后面的测试篇章。

## 4.2.6 增加特征值 CHAR6

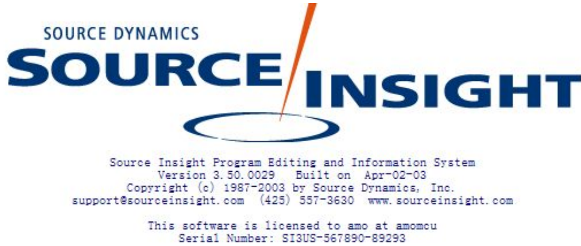
我们最重要的还是要实现透传，要实现透传， 我们有个前提， 就是需要建立连接后，需要对某个特征值进行写和通知。目前我们是在 BLE1.3.2 的协议栈的 SimpleProfile 基础上增加：

#define SIMPLEPROFILE_CHAR6_UUID 0xFFF6

其位于： \BLE-CC254x-1.3.2\Projects\ble\Profiles\SimpleProfile\simpleGATTprofile.h

下面来梳理一下我们增加 CHAR6 后的代码：
注意： 我们为了便于对比， 使用了 以下的源码对比软件，请同学们使用这个哈

这里来电题外话，对初学者应该是有用的。磨刀不误砍柴工， 要是问我干了 10 年技术，有什么感触，我感触颇多哦，驱动软件工程师装机必备：
一个是 Source Insight：




另一个就是上面所提到的 Beyond Compare 2。
左手刀右手枪，小米加齐上阵哈， 源码分析、源码修改、以及源码修改后的 bug 查找对比，那是相当的方便的。言归正传。

【1】头文件 simpleGATTprofile.h

这个 simpleGATTprofile.h 实现的是 TI 自定义的 5 个 profile，我们在此基础上增加了 CHAR6 和 CHAR7， 但是我们透传只用到 CHAR6，CHAR7 大家可不必理会。

```
56 // Profile Parameters
57 #define SIMPLEPROFILE_CHAR1                0  // RW uint8 - Profile Characteristic 1 value
58 #define SIMPLEPROFILE_CHAR2                1  // RW uint8 - Profile Characteristic 2 value
59 #define SIMPLEPROFILE_CHAR3                2  // RW uint8 - Profile Characteristic 3 value
60 #define SIMPLEPROFILE_CHAR4                3  // RW uint8 - Profile Characteristic 4 value
61 #define SIMPLEPROFILE_CHAR5                4  // RW uint8 - Profile Characteristic 4 value
62 #define SIMPLEPROFILE_CHAR6                5  // RW uint8 - Profile Characteristic 5 value
63 #define SIMPLEPROFILE_CHAR7                6  // RW uint8 - Profile Characteristic 5 value
64
65 // Simple Profile Service UUID
66 #define SIMPLEPROFILE_SERV_UUID            0xFFF0
67
68 // Key Pressed UUID
69 #define SIMPLEPROFILE_CHAR1_UUID          0xFFF1
70 #define SIMPLEPROFILE_CHAR2_UUID          0xFFF2
71 #define SIMPLEPROFILE_CHAR3_UUID          0xFFF3
72 #define SIMPLEPROFILE_CHAR4_UUID          0xFFF4
73 #define SIMPLEPROFILE_CHAR5_UUID          0xFFF5
74 #define SIMPLEPROFILE_CHAR6_UUID          0xFFF6
75 #define SIMPLEPROFILE_CHAR7_UUID          0xFFF7
76
77 // Simple Keys Profile Services bit fields
78 #define SIMPLEPROFILE_SERVICE             0x00000001
79
80 // Length of Characteristic 5 in bytes
81 #define SIMPLEPROFILE_CHAR5_LEN         5
82 #define SIMPLEPROFILE_CHAR6_LEN         19    //主机读写 (理应最大可设成20， 但不知为何目前只能最大设成19 --amomcu)
83 #define SIMPLEPROFILE_CHAR7_LEN         SIMPLEPROFILE_CHAR6_LEN    //从机通知，这里不需要用到
84
85 /*********************************************************************
86  * TYPEDEFS
87  */
```

62 行，增加 CHAR6 的 profile 参数。

74 行，增加该特征值的 UUID。

82 行，增加该特征值的长度，这个是参考 CHAR5 来实现的，但是原版的 CHAR5 有个缺陷，就是读写都只能是 5 个字节，但是我们既然是串口透传，意味着我们传输的数据长度是随意的，有可能是 1 个字节，也有可能是 120 个字节，所以在 82 行这里我们设置为 19

个字节（理应最大可设成 20， 但不知为何目前只能最大设成 19， 后面有时间再找找原因，哪位朋友发现问题所在，请告诉我哈，谢谢了)。

```
146  *    param - Profile parameter ID
147  *    value - pointer to data to write.  This is dependent on
148  *            the parameter ID and WILL be cast to the appropriate
149  *            data type (example: data type of uint16 will be cast to
150  *            uint16 pointer).
151  */
152 extern bStatus_t SimpleProfile_GetParameter( uint8 param, void *value, uint8 *returnBytes);
153
```

152 行，原版的 SimpleProfile_GetParameter 是不带第二个参数 uint8 *returnBytes 的，我们把他增加上，见名知意，returnBytes 就是返回多少个字节的意思，这里正好对应到上面所说到的 “我们传输的数据长度是随意的”。

【2】C 文件 simpleGATTprofile.c

文件位置：\ble\Profiles\SimpleProfile\simpleGATTprofile.c

simpleGATTprofile.c 是我们的自定义的 profile 的实现文件，包含了可悲应用软件调用的 GATT profile 和 GATT 服务（0xfff0）。下面我们来分析一下我们增加的代码：

```
110 // Characteristic 6 UUID: 0xFFF6
111 CONST uint8 simpleProfilechar6UUID[ATT_BT_UUID_SIZE] =
112 {
113   LO_UINT16(SIMPLEPROFILE_CHAR6_UUID), HI_UINT16(SIMPLEPROFILE_CHAR6_UUID)
114 };
115
116 // Characteristic 7 UUID: 0xFFF7
117 CONST uint8 simpleProfilechar7UUID[ATT_BT_UUID_SIZE] =
118 {
119   LO_UINT16(SIMPLEPROFILE_CHAR7_UUID), HI_UINT16(SIMPLEPROFILE_CHAR7_UUID)
120 };
```

111 行至 114 行，增加 16BIT 特征值的 UUID 定义。

```
200 // Simple Profile Characteristic 6 Properties
201 static uint8 simpleProfileChar6Props = GATT_PROP_READ | GATT_PROP_WRITE_NO_RSP | GATT_PROP_NOTIFY;
202
203 // Characteristic 6 Value
204 static uint8 simpleProfileChar6[SIMPLEPROFILE_CHAR6_LEN] = { 0, 0, 0, 0, 0 };
205 static uint8 simpleProfileChar6Len = 0;
206
207 // Simple Profile Characteristic 6 Configuration Each client has its own
208 // instantiation of the Client Characteristic Configuration. Reads of the
209 // Client Characteristic Configuration only shows the configuration for
210 // that client and writes only affect the configuration of that client.
211 static gattCharCfg_t simpleProfileChar6Config[GATT_MAX_NUM_CONN];
212
213 // Simple Profile Characteristic 6 User Description
214 static uint8 simpleProfileChar6UserDesp[17] = "Characteristic 6\0";
215
216
217 // Simple Profile Characteristic 7 Properties
218 //static uint8 simpleProfileChar7Props = GATT_PROP_NOTIFY;
219
220 // Characteristic 7 Value
221 static uint8 simpleProfileChar7[SIMPLEPROFILE_CHAR7_LEN] = { 0, 0, 0, 0, 0 };
222 static uint8 simpleProfileChar7Len = 0;
223
224 // Simple Profile Characteristic 7 Configuration Each client has its own
225 // instantiation of the Client Characteristic Configuration. Reads of the
226 // Client Characteristic Configuration only shows the configuration for
227 // that client and writes only affect the configuration of that client.
228 static gattCharCfg_t simpleProfileChar7Config[GATT_MAX_NUM_CONN];
229
230 // Simple Profile Characteristic 7 User Description
231 //static uint8 simpleProfileChar7UserDesp[17] = "Characteristic 7\0";
```

201 行使我们增加的 CHAR6 特征值 profile 的特征描述，意思是这个主设备可对该特征

值读（目前不需要的，如果读，则返回的数据不正确）、写、和通知。实际在透传过程中，我们只需用到写和通知即可， 具体来说就是，主机对从机是写，而从机到主机是是通知。

下面我们来讲一讲 237 行的表格：

static gattAttribute_t simpleProfileAttrTbl[SERVAPP_NUM_ATTR_SUPPORTED]

该 simpleProfileAttrTbl 是 GATT 的属性表，提到属性，我们可以理解成他的特点，也就是他都有什么特点，这个是在增加服务的时候用到的， 具体可见 SimpleProfile_AddService 的函数体。

```
375     // Characteristic 6 Declaration
376     {
377       { ATT_BT_UUID_SIZE, characterUUID },
378       GATT_PERMIT_READ,
379       0,
380       &simpleProfileChar6Props
381     },
382
383       // Characteristic Value 6
384       {
385         { ATT_BT_UUID_SIZE, simpleProfilechar6UUID },
386         GATT_PERMIT_READ | GATT_PERMIT_WRITE,
387         0,
388         simpleProfileChar6
389       },
390
391       // Characteristic 6 configuration
392       {
393         { ATT_BT_UUID_SIZE, clientCharCfgUUID },
394         GATT_PERMIT_READ | GATT_PERMIT_WRITE,
395         0,
396         (uint8 *)simpleProfileChar6Config
397       },
398
399       // Characteristic 6 User Description
400       {
401         { ATT_BT_UUID_SIZE, charUserDescUUID },
402         GATT_PERMIT_READ,
403         0,
404         simpleProfileChar6UserDesp
405       },
```

依照 TI 原本给出的 CHAR4， 我们同样增加了上面这一段。

```
479 bStatus_t SimpleProfile_AddService( uint32 services )
480 {
481   uint8 status = SUCCESS;
482
483   // Initialize Client Characteristic Configuration attributes
484   GATTServApp_InitCharCfg( INVALID_CONNHANDLE, simpleProfileChar4Config );
485   GATTServApp_InitCharCfg( INVALID_CONNHANDLE, simpleProfileChar6Config );
486   //GATTServApp_InitCharCfg( INVALID_CONNHANDLE, simpleProfileChar7Config );
487
488   // Register with Link DB to receive link status change callback
489   VOID linkDB_Register( simpleProfile_HandleConnStatusCB );
490
491   if ( services & SIMPLEPROFILE_SERVICE )
492   {
493     // Register GATT attribute list and CBs with GATT Server App
494     status = GATTServApp_RegisterService( simpleProfileAttrTbl,
495                                           GATT_NUM_ATTRS( simpleProfileAttrTbl ),
496                                           &simpleProfileCBs );
497   }
498
499   return ( status );
500 }
```

485 行， 依照 CHAR4 增加了这一行，大家可能会问为什么是参照 CHAR4 而不是 CHAR1 或 CHAR2， 这主要是原本 CHAR4 就是只具备 notify 的属性，所以，我们既然有 notify 的属性， 必然需要它所需要的代码和流程，只不过 CHAR4 处理的是 1 个字节，而我们增加的 CHAR6 每次可处理的字节数可达 19 至 20 个字节， 这才能应用到数据透传上。

下面我们来看一下从机对主机的通知：
bStatus_t SimpleProfile_SetParameter( uint8 param, uint8 len, void *value )

## 4.2.7 主机发送数据

这个函数是主机往从机发送透传数据， 这个速度非常的快。

```
00783: // 处理: 串口发送过来的数据， 通过无线发送出去
00784: void simpleBLE_UartDataMain(uint8 *buf, uint8 numBytes)
00785: {
00786:     if(FALSE == simpleBLE_CheckIfUse_Uart2Uart())     //未使用透传模式时不透传
00787:         return;
00788:
00789:     if(GetBleRole() == BLE_ROLE_CENTRAL )//主机
00790:     {
00791:         if(simpleBLEChar6DoWrite
00792:         && ( simpleBLECharHd6 != 0)
00793:         && simpleBLECentralCanSend)                //写入成功后再写入
00794:         {
00795:             attWriteReq_t AttReq;
00796:
00797:             LCD_WRITE_STRING_VALUE( "s=", numBytes, 10, HAL_LCD_LINE_1 );
00798:
00799:             simpleBLEChar6DoWrite = FALSE;
00800:
00801:             AttReq.handle = simpleBLECharHd6;
00802:             AttReq.len = numBytes;
00803:             AttReq.sig = 0;
00804:             AttReq.cmd = 0;
00805:             osal_memcpy(AttReq.value,buf,numBytes);
00806:             GATT_WriteCharValue( 0, &AttReq, simpleBLETaskId );
00807:         }
```

## 4.2.8 主机接收数据

在以下函数中 有串口透传的接收数据入口：
static void simpleBLECentralProcessGATTMsg( gattMsgEvent_t *pMsg )

```
00716:   else if ( ( pMsg->method == ATT_HANDLE_VALUE_NOTI ) )   //通知
00717:   {
00718:     if( pMsg->msg.handleValueNoti.handle == simpleBLECharHd6/*0x0035*/)   //CH
00719:     {
00720:         if(simpleBLE_CheckIfUse_Uart2Uart())     //使用透传模式时才透传
00721:         {
00722:             NPI_WriteTransport(pMsg->msg.handleValueNoti.value, pMsg->msg.han
00723:         }
00724:     }
00725:   }
00726: } ? end simpleBLECentralProcessGATTMsg ?
```

## 4.2.9 从机发送数据

// 处理：串口发送过来的数据， 通过无线发送出去
void simpleBLE_UartDataMain(uint8 *buf, uint8 numBytes)

```
00816:          if(simpleBLEChar6DoWrite2)              //写入成功后再写入
00817:          {
00818: #if 0 // 这种速度慢 SimpleProfile_SetParameter
00819:             simpleBLEChar6DoWrite2 = FALSE;
00820:             SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR6,numBytes, buf );
00821: #else // 这种速度快 GATT_Notification
00822:             static attHandleValueNoti_t pReport;
00823:             pReport.len = numBytes;
00824:             pReport.handle = 0x0035;
00825:             osal_memcpy(pReport.value, buf, numBytes);
00826:             GATT_Notification( 0, &pReport, FALSE );
00827: #endif
00828:          }
00829:          else
00830:          {
00831:             LCD_WRITE_STRING_VALUE( "simpleBLEChar6DoWrite=", simpleBLEChar6D
00832:          }
```

## 4.2.10 从机接收数据

在以下函数中，有串口透传的接收数据入口：
static void simpleProfileChangeCB( uint8 paramID )

```
00888:       case SIMPLEPROFILE_CHAR6:
00889:         SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR6, newChar6Value, &returnB
00890:         if(returnBytes > 0)
00891:         {
00892:           if(simpleBLE_CheckIfUse_Uart2Uart())      //使用透传模式时才透传
00893:           {
00894:             NPI_WriteTransport(newChar6Value,returnBytes);
00895:           }
00896:         }
00897:
00898:         break;
```

# 4.3 主从一体公共文件主要函数分析

## 4.3.1 主从一体公共头文件 simpleBLE.h

// 系统定时器间隔时间
#define SBP_PERIODIC_EVT_PERIOD 100//必须是 100ms

//最大记录的从机地址
#define MAX_PERIPHERAL_MAC_ADDR 10//最大记录的从机地址

//mac 地址的字符长度 (一个字节等于两个字符)
#define MAC_ADDR_CHAR_LEN 12//mac 地址的字符长度 (一个字节

等于两个字符)

```
// 出厂设置或清除配对信息与从机信息
typedef enum
{
    PARA_ALL_FACTORY = 0,           //全部恢复出厂设置
    PARA_PARI_FACTORY = 1,          //配对信息恢复出厂设置-相当于清除配对信息
与从机信息
}PARA_SET_FACTORY;

// 当前单片机运行的角色
typedef enum
{
    BLE_ROLE_PERIPHERAL = 0,        //从机角色
    BLE_ROLE_CENTRAL = 1,           //主机角色
}BLE_ROLE;

// 应用程序状态
enum
{
  BLE_STATE_IDLE,                   //无连接-空闲状态
  BLE_STATE_CONNECTING,             //连接中...
  BLE_STATE_CONNECTED,              //已连接上
  BLE_STATE_DISCONNECTING,          //断开连接中
  BLE_STATE_ADVERTISING             //从机广播中
};

// 系统裕兴模式定义
enum
{
  BLE_MODE_SERIAL,                  // 串口透传模式 【默认】
  BLE_MODE_DRIVER,                  // 直驱模式
  BLE_MODE_iBeacon,                 // iBeacon 广播模式
  BLE_MODE_MAX,
};

// 连接模式指示
typedef enum
{
  CONNECT_MODE_FIX_ADDR_CONNECTED,       // 指定 mac 地址进行连接
  CONNECT_MODE_LAST_ADDR_CONNECTED,      // 连接最后成功连接过的 mac 地址
  CONNECT_MODE_MAX,
}CONNECT_MODE;
```

```
// 应用程序状态
typedef enum
{
   BLE_CENTRAL_CONNECT_CMD_NULL,   //主机 AT 连接命令  空
   BLE_CENTRAL_CONNECT_CMD_CONNL, //主机 AT 连接命令连接最近成功过的地址
   BLE_CENTRAL_CONNECT_CMD_CON,   //主机 AT 连接命令  连接指定地址
   BLE_CENTRAL_CONNECT_CMD_DISC,  //主机 AT 扫描从机命令
   BLE_CENTRAL_CONNECT_CMD_CONN, //主机 AT 连接命令  连接扫描到的地址的下
标号对应的地址
}BLE_CENTRAL_CONNECT_CMD;
extern BLE_CENTRAL_CONNECT_CMD g_Central_connect_cmd ;


// 定于系统结构变量， 该结构会在开机时从 nv flash 中读取，
// 数据有修改时， 需要写入 nv flash
// 这样， 就实现了系统重启后数据还是上一次设置的
typedef struct
{
    /*
    波特率
    0---------9600
    1---------19200
    2---------38400
    3---------57600
    4---------115200
    */
    uint8 baudrate;              //波特率 ， 目前支持的列表如上
    uint8 parity;                //校验位
    uint8 stopbit;               //停止位

    uint8 mode;                  //工作模式 0:透传 ， 1: 直驱 , 2: iBeacon

    // 设备名称，最长 11 位数字或字母，含中划线和下划线，不建议用其它字符
    uint8 name[12];

    BLE_ROLE role;               //主从模式  0: 从机   1: 主机

    uint8 pass[7];               //密码， 最大 6 位 000000~999999

    /*
    Para: 0 ~ 1
    0: 连接不需要密码
```

```
    1: 连接需要密码
    */
    uint8 type;                              //鉴权模式

    uint8 mac_addr[MAC_ADDR_CHAR_LEN+1];       //本机 mac 地址 最大 12 位 字符表示
    uint8 connect_mac_addr[MAC_ADDR_CHAR_LEN+1];   //指定去连接的 mac 地址

    //曾经成功连接过的从机个数
    uint8 ever_connect_peripheral_mac_addr_conut;
    //曾经成功连接过的从机个数,当前 index， 用于增加从机地址时快速插入或读取
    uint8 ever_connect_peripheral_mac_addr_index;
    //最新一次成功连接过的从机地址 index， 用于针对 AT+CONNL 这个指令
    uint8 last_connect_peripheral_mac_addr_index;
    //曾经成功连接过的从机地址
    uint8
ever_connect_mac_status[MAX_PERIPHERAL_MAC_ADDR][MAC_ADDR_CHAR_LEN];

    /*
    Para: 000000～009999
    000000 代表持续连接，其
    余代表尝试的毫秒数
    Default:001000
    */

    uint16 try_connect_time_ms;               // 尝试连接时间---目前无效
    int8 rssi;                                  //  RSSI 信号值
    uint8 rxGain;                               //  接收增益强度
    uint8 txPower;                              //  发射信号强度
    uint16 ibeacon_adver_time_ms;             // 广播间隔

    //  模块工作类型  0: 立即工作， 1: 等待 AT+CON 或 AT+CONNL 命令
    uint8 workMode;
}SYS_CONFIG;
```

## 4.3.2 主从一体公共头文件 simpleBLE.c

```
// 该函数延时时间为 1ms， 用示波器测量过， 稍有误差， 但误差很小
void simpleBLE_Delay_1ms(int times);

// 字符串对比
static uint8 str_cmp(uint8 *p1,uint8 *p2,uint8 len);
```

```
// 字符串转数字
uint32 str2Num(uint8* numStr, uint8 iLength);

char *bdAddr2Str( uint8 *pAddr );

// 保存所有数据到 nv flash
void simpleBLE_WriteAllDataToFlash();

// 读取自定义的 nv flash 数据 -------未使用到
void simpleBLE_ReadAllDataToFlash();

//flag: PARA_ALL_FACTORY: 全部恢复出厂设置
//flag: PARA_PARI_FACTORY: 清除配对信息
void simpleBLE_SetAllParaDefault(PARA_SET_FACTORY flag);

// 打印所有存储到 nv flash 的数据， 方便调试代码
void PrintAllPara(void);

// 返回设备角色
//主从模式 0: 从机 1: 主机
BLE_ROLE GetBleRole();

// 判断蓝牙是否连接上
// 0: 未连接上
// 1: 已连接上
bool simpleBLE_IfConnected();

// 增加从机地址， 注意， 需要连接成功后， 再增加该地址
void simpleBLE_SetPeripheralMacAddr(uint8 *pAddr);

// 读取从机地址, index < MAX_PERIPHERAL_MAC_ADDR
// 用于判断是否系统中已存有该 Mac 地址
/*
index: 应该是 < MAX_PERIPHERAL_MAC_ADDR,
*/
bool simpleBLE_GetPeripheralMacAddr(uint8 index, uint8 *pAddr);

//开机时判断到按键按下 3 秒， 恢复出厂设置
//按键定义为 p0.7
void CheckKeyForSetAllParaDefault(void);

// 串行口 uart 初始化
void simpleBLE_NPI_init(void);
```

```
// 设置接收增益
void UpdateRxGain(void);

// 设置发射功率
void UpdateTxPower(void);

// 设置 led 灯的状态
void simpleBle_LedSetState(uint8 onoff);

// 保存 RSSI 到系统变量
void simpleBle_SetRssi(int8 rssi);

// 串口打印密码 -----测试用----
void simpleBle_PrintPassword();

// 获取设备名称
uint8* simpleBle_GetAttDeviceName();

// 主机是否记录了从机地址
bool simpleBle_IFfHavePeripheralMacAddr( void );

// 定时器任务定时执行函数， 用于设置 led 的状态----也可以增加一个定时器来做
void simpleBLE_performPeriodicTask( void );

// 获取鉴权要求, 0: 连接不需要密码， 1: 连接需要密码
bool simpleBle_GetIfNeedPassword();

// 获取连接密码
uint32 simpleBle_GetPassword();

// 判断是否是 iBeacon 广播模式
bool simpleBLE_CheckIfUse_iBeacon();

// 判断是否使能串口透传
bool simpleBLE_CheckIfUse_Uart2Uart();

// 判断是输入的形参-地址是否是需要去连接的地址，如果是， 返回真， 否则返回假
bool simpleBLE_GetToConnectFlag(uint8 *Addr);

// 设置 iBeacon 的广播间隔
uint32 simpleBLE_GetiBeaconAdvertisingInterral();

// 串口回调函数， 下面把该回调函数里实现的功能讲解一下
/*
```

1，思路： 当串口收到数据后，就会马上调用以下回调函数，在实际测试中发现，此回调函数调用频繁， 如果你不执行 NPI_ReadTransport 函数进行读取， 那么这个回调函数就会频繁地被执行，但是，你通过串口发送一段数据， 你本意是想处理这一完整一段的数据，所以，
我们在下面引入了时间的处理方法， 也即接收的数据够多或者超时，就读取一次数据，
然后根据当前的状态决定执行，如果没有连接上，就把所有数据当做 AT 命令处理， 如果连接
上了，就把数据送到对端。
*/

//uart 回调函数
static void simpleBLE_NpiSerialCallback( uint8 port, uint8 events );

// 处理：串口发送过来的数据， 通过无线发送出去
void simpleBLE_UartDataMain(uint8 *buf, uint8 numBytes);

// AT 命令处理 函数
bool simpleBLE_AT_CMD_Handle(uint8 *pBuffer, uint16 length);

/*
很多朋友问我们， 如何实现把主机或从机上的传感器数据直接发送到对端并通过主机的串口透传出去， 下面我们就能实现这个功能， 不过到底需要什么样的传感器， 以及什么样的数据就需要你自己来组织了， 下面这个函数每 100ms 执行一次:都可以把数据发送到对端， 对端通过串口透传出去。下面给出一个样例：实现把字符串发送到对方
*/
void simpleBLE_SendMyData_ForTest();

### 4.3.3 主设备文件

【1】simpleBLECentral.h
【2】simpleBLECentral.c
【3】OSAL_simpleBLECentral.c
【4】SimpleBLEPeripheral_Main.c （这个是主设备与从设备的公用文件）

### 4.3.4 从设备文件

【1】simpleBLEPeripheral.h
【2】simpleBLEPeripheral.c
【3】OSAL_simpleBLEPeripheral.c
【4】SimpleBLEPeripheral_Main.c （这个是主设备与从设备的公用文件）

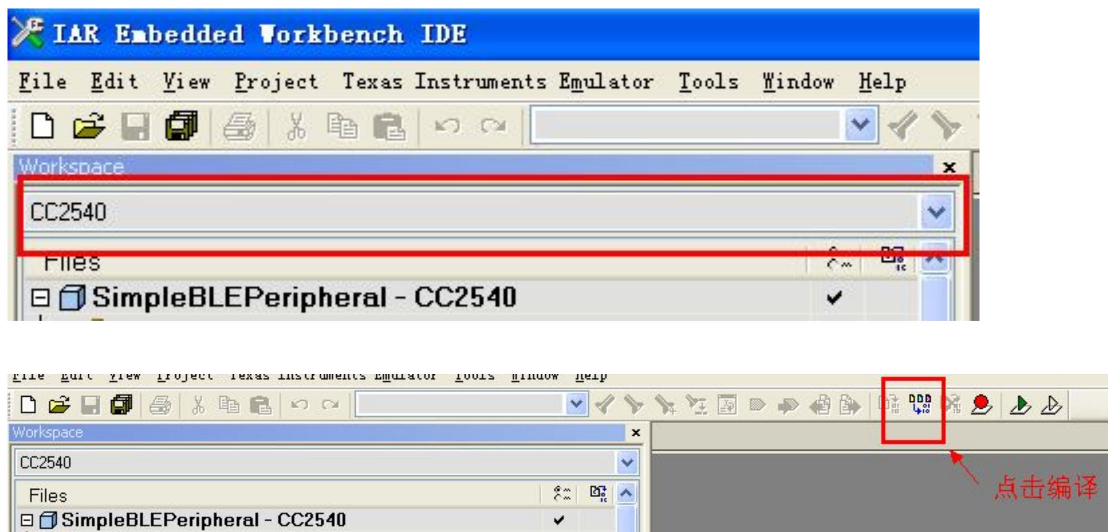
# 5，源码编译

（必须安装了 IAR 8.10.3 或 IAR 8.10.4 版本。如果没有安装，请先按照我们的《1.BLE 入门与提高教程》来先安装好 IAR）

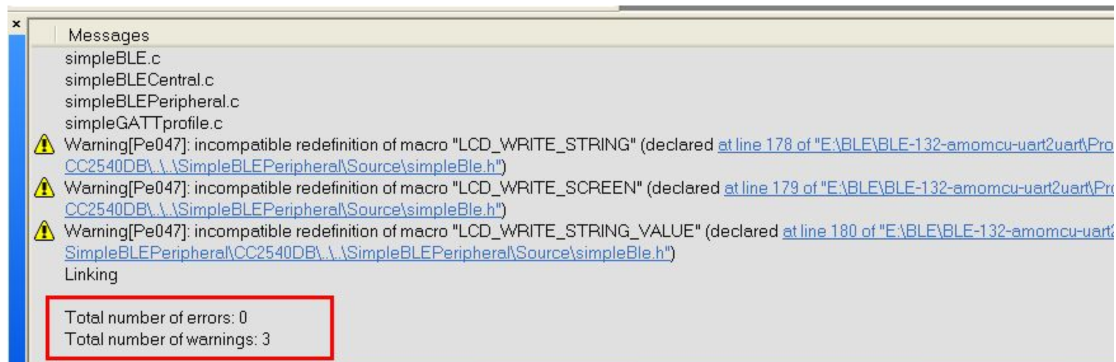
如果是 cc2540， 打开 \BLE-132-ZCYT\Projects\ble\SimpleBLEPeripheral\CC2540DB\SimpleBLEPeripheral.eww 工程，

如果是 cc2541， 打开 \BLE-132-ZCYT\Projects\ble\SimpleBLEPeripheral\CC2541DB\SimpleBLEPeripheral.eww 工程，

然后一定必须记得， 编译选项选 CC2540 或 CC2541， 否则编译会出错的。




编译将没有任何错误， 可能有若干个 wanning，可忽略之。

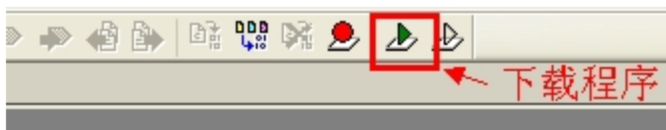


# 6，下载运行

两种下载方法：

下载之前需要先连接好我们开发板， 如何连接， 请参考我们的 《1.BLE 入门与提高教程》， 还需提醒注意的是， 我们的 cc-debugger 一定是需要显示绿色才是表明正确连接了开发板的。

【1】 IAR 直接下载：




【2】用 SmartRF Flash Programmer 下载固件：

路径如下：（文件名可能经过升级后稍有不同）

如果是 CC2540：
\SimpleBLEPeripheral\CC2540DB\CC2540\Exe\BLE_MasteSlave_V1.2_cc2540.hex

如果是 CC2541：
\SimpleBLEPeripheral\CC2541DB\CC2541\Exe\BLE_MasteSlave_V1.2_cc2541.hex

# 7，测试

## 7.1 双机主从一体串口透传

测试前， 需要打开两个 PC 上的 SSCOM.exe 串口助手， 并都如下图设置：
波特率 9600，数据位 8，停止位 1 校验 none， 一定要选中发送新行(也就是 AT 命令均以新行结尾， 十六进制是 0x0D 0x0A， 字符串表示就是 “\r\n”)
我们的 AT 命令较多， 这里只介绍两条 AT 指令， 实现 AT 命令和主机从机切换即可。

1， AT\r\n

返回 OK\r\n （注意： “\r\n”在返回时表示为回车换行了， 如果你切换成 hex 显示，即可看到是 0x0D 0x0A 结尾 ）

2， AT+ROLE1\r\n

这是切换成主机的意思。

我们的源码编译出来， 默认会跑从机的， 既然要做主从透传， 必然有一个主机，对吧。

增加连接方法：

主机：

按以下顺序发指令：

【1】 切换成主机
发 AT+ROLE1
回 OK+Set:1
Hello World Central

【2】设置连接模式
发 AT+IMME1
回 OK+Set:1

**(v1.4-20140821 版本增加)**
如果在这一步：
发 AT+IMME0
回 OK+Set:0
那么后面的主机重启后， 就会自动连接最近连接成功的从机

【3】扫描送机
发 AT+DISC?
回
OK+DISCS
OK+DISC:78A50450354B
OK+DISC:78A504502635
OK+DISCE

【4】连接从机
发 AT+CONN0 连接 78A50450354B 从机
或
发 AT+CONN1 连接 78A504502635 从机

从机立即会输出 Connected， 而主机会在更新参数后， 大约 2~3 秒后才返回 Connected

此时进入 了透传模式 。。。。。。。。。。。。。。。。。。。。。。。。。。。

【5】如果主机重启

**(v1.4-20140821 版本增加)**
如果之前设置过下面的指令：

发 AT+IMME0
回 OK+Set:0
那么后面的主机重启后，　就会自动连接最近连接成功的从机

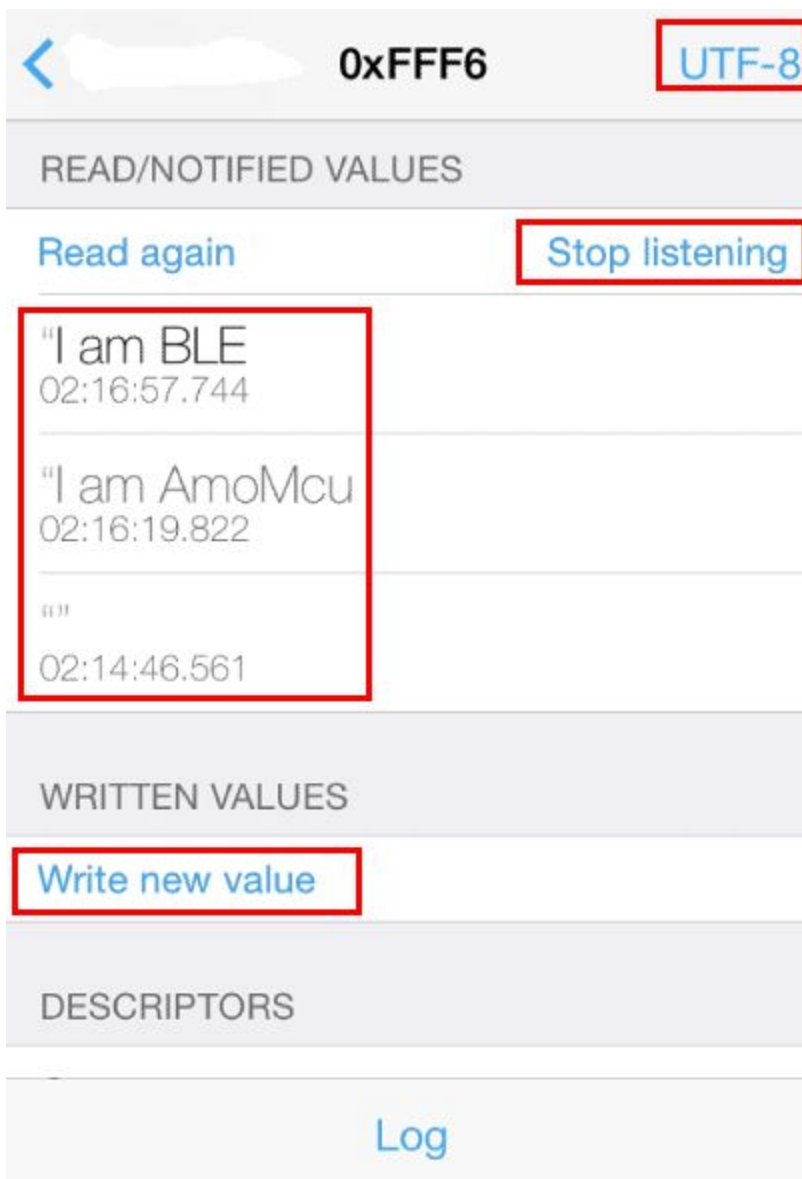
否则会如果之前是设置　AT+IMME1，　那么：
可以 以下三条命令中的一条来连接从机
A,　AT+CONNL　　　　　连接最近连接成功的从机
　发 AT+CONNL，　从机立即会输出 Connected，　而主机会在更新参数后，　大约 2~3 秒后才返回 Connected
此时进入 了透传模式。。。。。。。。。。。。。。。。。。。。。。。。。。。

B,　AT+CON78A50450354B　　连接指定从机
　发 AT+CON78A50450354B，　1.5 秒后从机立即会输出 Connected，　而主机会在更新参数后，　大约 2~3 秒后才返回 Connected
此时进入 了透传模式。。。。。。。。。。。。。。。。。。。。。。。。。。。

注意一下 D1 这个 led 灯的状态：
　LED 状态灯显示
　　LED 状态灯已实现。状态灯修改如下：
　　连接前：
　　　主机，未记录从机地址时，每秒亮 100ms；
　　　主机，记录从机地址时，每秒亮 900ms；
　　　从机，每 2 秒亮 1 秒。
　　连线后：
　　　主机与从机均为每 5 秒亮 100 毫秒。（闪亮，　以便省电）

　另外，　我们这个固件具有掉线自动重连的功能，　任何一方掉线，　都可以重新进行连接。

## 7.2 用 ios 的 lightblue 透传




该教程， 还需要时间完善， 谢谢

## 7.3 用 AmoMcu.apk 的测试

该教程， 还需要时间完善， 谢谢。
这个 apk 的源码目前可能叫做 “Ex039BLE”， 所以叫做 Ex039BLE.apk
2014.08.06

# 8，联系我们

**QQ 群：257318688**
**QQ:　　11940507**
**Tel:　　18588220515　　阿莫**
**网站支持：　www.AmoMcu.com　　阿莫单片机社区网**
**淘宝店铺：　http://amomcu.taobao.com**

# 9，附录　AT 命令

**默认的串口配置为：波特率 9600，无校验，数据位 8，停止位 1，无流控。**

## 9.1　AT 测试指令

| 指令 | 应答 | 参数 |
|---|---|---|
| AT | OK | 无 |

例：发送 AT，返回 OK

## 9.2　AT+BAUD 查询、设置串口波特率

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+BAUD? | OK+Get:[para1] | Para1：0～4<br>0=9600;1=19200;<br>2=38400;3=57600;<br>4=115200<br>Default：0（9600） |
| 设置：AT+BAUD[para1] | OK+Set:[para1] | |

例：
发送：AT+BAUD2
返回：OK+Set:2
0---------9600
1---------19200
2---------38400
3---------57600
4---------115200

## 9.3　AT+PARI 设置串口校验

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+PARI? | OK+ Get:[para] | 无 |
| 设置：AT+PARI[para] | OK+Set:[para] | Para 范围 0,1,2<br>0: 无校验<br>1: EVEN<br>2: ODD<br>Default: 0 |

## 9.4　AT+STOP 设置串口停止位

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+STOP? | OK+ Get:[para] | |
| 设置：AT+STOP[para] | OK+Set:[para] | Para: 0~1<br>0: 1 停止位<br>1: 2 停止位<br>Default: 0 |

## 9.5　AT+MODE 设置模块工作模式

| 指令 | 应答 | 参数 |
| --- | --- | --- |
| 查询：AT+MODE? | OK+ Get:[para] | |
| 设置：AT+MODE[para] | OK+Set:[para] | Para: 0～1<br>0: 开启串口透传模式<br>1: 关闭串口透传模式<br>Default: 0 |

## 9.6　AT+NAME 查询、设置设备名称

| 指令 | 应答 | 参数 |
| --- | --- | --- |
| 查询：AT+NAME？ | OK+NAME[para1] | Para1：设备名称<br>最长 11 位数字或字母，含中划线和下划线，不建议用其它字符。<br>Default：Microduino |
| 设置：AT+NAME[para1] | OK+Set[para1] | |

例：发送 AT+NAMEbill_gates

返回 OK+Set:bill_gates

这时蓝牙模块名称改为 bill_gates

## 9.7　AT+RENEW 恢复默认设置(Renew)

| 指令 | 应答 | 参数 |
| --- | --- | --- |
| AT+RENEW | OK+RENEW | |

## 9.8　AT+RESET 模块复位，重启(Reset)

| 指令 | 应答 | 参数 |
| --- | --- | --- |
| AT+RESET | OK+RESET | |

## 9.9　AT+ROLE 查询、设置主从模式

| 指令 | 应答 | 参数 |
| --- | --- | --- |
| 查询：AT+ROLE? | OK+ Get:[para1] | Para1: 0～1<br>1: 主设备<br>0: 从设备<br>Default: 0 |
| 设置：AT+ROLE[para1] | OK+Set:[para1] | |

## 9.10 AT+PASS 查询、设置配对密码

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+PASS? | OK+PASS:[para1] | Para1: 000000~999999<br>Default：000000 |
| 设置：AT+PASS[para1] | OK+Set:[para1] | |

例如：

发送 AT+PASS008888、返回 OK+Set:008888

## 9.11 AT+TYPE 设置模块鉴权工作类型

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+TYPE? | OK+ Get:[para] | |
| 设置：AT+TYPE[para] | OK+Set:[para] | Para: 0 ~ 1<br>0: 连接不需要密码<br>1: 连接需要密码<br>Default: 0 |

## 9.12 AT+ADDR 查询本机 MAC 地址

| 指令 | 应答 | 参数 |
|---|---|---|
| 或者：AT+ADDR? | OK+LADD:MAC 地址 | |

## 9.13 AT+CONNL 连接最后一次连接成功的从设备

| 指令 | 应答 | 参数 |
|---|---|---|
| AT+CONNL | OK+CONN[Para] | Para: L, N, E,F<br>L:连接中、N:空地址<br>E:连接错误、F:连接失败 |

## 9.14 AT+CON 连接指定蓝牙地址的从设备

| 指令 | 应答 | 参数 |
|---|---|---|
| AT+CON[para1] | OK+CONN[Para2] | Para1: MAC 地址、<br>如: 0017EA0923AE<br>Para2: A, E, F<br>A: 连接中<br>E: 连接错误<br>F: 连接失败 |

## 9.15 AT+CLEAR 清除主设备配对信息

| 指令 | 应答 | 参数 |
|---|---|---|
| AT+CLEAR | OK+CLEAR | |

清除成功连接过的设备地址码信息

备注：此指令只有在主设备时才有效；从设备时不接受此指令

## 9.16 AT+RADD 查询成功连接过的从机地址

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+RADD? | OK+RADD:MAC 地址<br>... | Para：蓝牙设备 MAC 地址<br>最多返回 10 个设备地址 |

注：只能显示在主模式下成功连接过的地址

## 9.17 AT+VERS 查询软件版本

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+VERS? | 版本信息 | |

## 9.18 AT+TCON 设置主模式下尝试连接时间

(此参数设置没有意义)

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+TCON? | OK+TCON:[para] | |
| 设置：AT+TCON[para] | OK+Set:[para] | Para: 000000～009999<br>000000 代表持续连接，其余代表尝试的毫秒数<br>Default:001000 |

注：该指令只在主模式下有效，当模块记住了上一次成功链接的地址后，再次开机自动尝试连接该地址分钟数由此参数控制，超过该数值，会自动进入搜索状态，000000 为一直尝试连接，该参数值为毫秒，如无必要请不要设置该值太小，会影响模块正常工作。

## 9.19 AT+RSSI 读取 RSSI 信号值

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+RSSI? | OK+ RSSI:[para] | Para:信号强度，单位为 db<br>Para 是一个负数，绝对值越小说明信号强度越大 |

## 9.20　AT+TXPW 改变模块发射信号强度

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+TXPW? | OK+ TXPW:[para] | |
| 设置：AT+TXPW[para] | OK+Set:[para] | Para: 0 ~ 3<br>0: 4dbm、1: 0dbm<br>2: -6dbm、3: -23dbm<br>Default: 0 |

## 9.21　AT+TIBE 改变模块作为 ibeacon 基站广播时间间隔

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+TIBE? | OK+ TIBE:[para] | |
| 设置：AT+TIBE[para] | OK+Set:[para] | Para: 000000～009999<br>000000 代表持续广播，其余代表尝试的毫秒数<br>Default:000500 |

## 9.22　AT+IMME 设置工作类型

| 指令 | 应答 | 参数 |
|---|---|---|
| 查询：AT+IMME? | OK+Get:[para] | |
| 设置：AT+IMME[para] | OK+Set:[para] | Para:0～1<br>0: 立即工作，<br>1: 等待 AT+CON 或 AT+CONNL 命令<br>Default:0 |