4.0

蓝牙4.0BLE开发完全手册

物联网开发技术实战

欧阳骏　陈子龙　黄宁淋　编著
粟思科　审校

化学工业出版社
·北京·

前言 Foreword

蓝牙无线技术是使用范围最广泛的全球短距离无线标准之一，全新的蓝牙 4.0 版本将三种蓝牙技术（即传统蓝牙、高速蓝牙和低功耗蓝牙技术）合而为一。它继承了蓝牙技术在无线连接上的固有优势，同时增加了高速蓝牙和低功耗蓝牙的特点。这三个规格可以组合使用也可以单独使用。蓝牙 4.0 规范的核心是低功耗技术（Low Energy），即蓝牙 4.0 BLE。该技术最大特点是拥有超低的运行功耗和待机功耗，蓝牙低功耗设备使用一粒纽扣电池可以连续工作数年之久，可应用于对成本和功耗都有严格要求的无线方案，广泛用于医疗保健、体育健身、家庭娱乐、传感器物联网等众多领域。

大多数读者，在学习新知识的时候很难静下心来去阅读大篇幅的概念性的内容，至少笔者当初在学习的时候就是这种心理，总以为那些话是写给明白人看的，因此为了让读者更轻松、更容易地理解蓝牙技术，本书按照如下主线编写：以蓝牙 4.0 BLE 协议栈为基础，结合智造者科技股份有限公司的 CC2540 蓝牙 4.0 BLE 多功能开发系统，通过大量实例分析向读者展示了构建一个符合蓝牙 4.0 BLE 协议的无线网络的总体过程，带领读者从实践的角度去掌握蓝牙 4.0 BLE 无线网络构建和开发的基础知识以及编程技巧。

本书的特点

▲ 理论与实践相结合。本书以实例为基础，详细阐述了进行蓝牙 4.0 BLE 应用开发所需要的基础知识和编程技巧，同时恰当地摒弃了一些对于初学者而言暂时不用或者很少用到的知识点，目的在于突出学习重点。

▲ 讲解简明，步骤详细。每个实例都是以通俗易懂的语言，并穿插图片和表格进行阐述。

▲ 模块化设计与系统设计相结合。

本书的编写原则

▲ 尽量向读者展示技术细节

书中有些地方可能看似很简单，显得有点啰嗦，但是为了给初学者展现出蓝牙 4.0 BLE 无线网络构建的全貌，笔者选择了这种通俗易懂的编写风格，因为笔者在以

往的学习过程中遇到很多问题，到技术论坛发帖求助，查阅资料，经过很长时间才解决问题，因此为了向读者展现一个完整的开发过程，宁可啰嗦一点，也不漏掉细节问题。

▲ 代码注重的是可读性，没有拘泥于效率和编程规范

本书代码力求通俗易懂，并没有过多的考虑程序执行的效率和编程风格。如果读者对基本的编程都没有大概了解的话，又谈何编程规范呢。因此，尽快掌握编程才是硬道理，其他问题可以后续解决。

▲ 尽量用朴实的语言去描述看似深奥的理论

笔者努力使本书作为一本蓝牙 4.0 BLE 无线网络构建和应用开发的实战手册，想展现出开发过程中的问题及其解决方法，尽量给读者提供一个参考，使读者少走弯路，因此，笔者选择用尽量通俗的语言来叙述，并不想用晦涩难懂的语言来迷惑读者。

虽然蓝牙 4.0 BLE 技术涉及电子、通信、射频等多学科的知识，但是，本书将给读者提供了一个崭新的学习思路，即从应用的角度去学习、理解进而掌握蓝牙无线网络构建的基本原理。

本书内容概述

▲ 第 1 章——简要介绍蓝牙 4.0 BLE 技术的基础知识和应用前景，同时给出智造者科技股份有限公司的 CC2540 开发板的硬件组成，这也是本书中采用的硬件平台。

▲ 第2章——对IAR集成开发环境进行讲解，摒弃部分初学者暂时用不到的功能，突出重点。

▲ 第 3 章——对 CC2540 开发板硬件资源的设计进行讲解，并介绍如何使用仿真调试下载器进行程序的仿真调试和下载。

▲ 第 4 章——给出一系列基础实验，帮助读者快速掌握 CC2540 射频单片机硬件资源的使用。

▲ 第 5 章——深入讲解蓝牙 4.0 BLE 协议栈中各层的机制，同时给出一个在协议栈中使用串口进行通信的实验。

▲ 第 6 章——对蓝牙 4.0 BLE 节点设备和集中器设备编程进行详细阐述。

▲ 第 7 章——结合实例详细阐述 PC 端调试开发工具 BToll 的使用方法。

▲ 第 8 章——讲述如何搭建蓝牙 4.0 BLE 协议分析仪，并结合协议分析仪捕获的数据对蓝牙无线数据传输流程进行分析。

▲ 第 9 章——结合实例，对蓝牙 4.0 BLE 无线网络中常用的项目开发经验进行阐述。

本书是蓝牙 4.0 BLE 应用开发入门级的读物，阅读完本书后，读者需要结合自己的项目需求，对相应的源代码进行修改来完成应用开发，只有通过不断的练习，才能真

正掌握蓝牙 4.0 BLE 无线网络应用开发的技术。

适用对象

- 高等院校电子、通信、自动控制等专业学生
- 从事蓝牙 4.0 BLE 应用开发的技术人员
- 向无线网络开发转型的工程师

编者与致谢

本书由欧阳骏、陈子龙、黄宁淋编著，粟思科审校，参与本书编写工作的人员还有王小强、张凯之、李岩、王治国、钟晓林、王娟、胡静、杨龙、张成林、方明、王波、雷晓、李军华、陈晓云、方鹏、龙帆、刘亚航、凌云鹏、陈龙、曹淑明、徐伟、杨阳、张宇、刘挺 、单琳、吴川、李鹏、朱榕、陈思涛和孙浩，另外还要感谢唐鑫、张剑、陈小军、王博、谢德华、童凯、田野、王丹、严林祥为本书编写所提供的帮助。

配套服务

我们为蓝牙读者和用户尽心服务，围绕蓝牙 4.0 BLE 技术、产品和项目市场，探讨应用与发展，发掘热点与重点；开辟了本书的讨论专区并提供技术支持，俱乐部 QQ：183090495，电子邮箱：hwhpc@163.com。欢迎读者讨论交流。

由于编者水平有限，加之时间仓促，书中难免有不恰当的地方，恳请广大读者及同行专家批评指正。

编者著

目录 Contents

第 1 章　蓝牙 4.0 BLE 简介

1.1　无线网络数据传输协议对比　2

1.2　短距离无线网络的分类　4

　1.2.1　什么是蓝牙 4.0 BLE　4

　1.2.2　蓝牙 4.0 BLE 的特点　4

1.3　蓝牙 4.0 BLE 协议简介　6

1.4　蓝牙 4.0 BLE 无线网络通信信道分析　6

1.5　蓝牙 4.0 BLE 无线网络拓扑结构　8

1.6　蓝牙 4.0 BLE 技术在物联网的应用　9

1.7　CC2540 开发板硬件资源概述　11

1.8　本章小结　13

［扩展阅读］　蓝牙技术发展历程　13

第 2 章　IAR 集成开发环境

2.1　IAR 集成开发环境简介　15

2.2　工程的编辑与修改　16

　2.2.1　建立一个新工程　16

2.2.2 建立一个源文件 19
2.2.3 添加源文件到工程 20
2.2.4 工程设置 24
2.2.5 源文件的编译 28
2.3 本章小结 29
[扩展阅读] 模块化编程技巧 29

第 3 章 CC2540 开发板硬件资源

3.1 CC2540 核心板硬件资源 32
3.1.1 CC2540 简介 32
3.1.2 天线及巴伦匹配电路设计 33
3.1.3 晶振电路设计 33
3.1.4 电源电路设计 34
3.1.5 LED 电路设计 34
3.1.6 AD 转换电路设计 35
3.1.7 串口电路设计 36
3.2 仿真调试与下载 36
3.2.1 仿真调试下载器驱动的安装 36
3.2.2 程序仿真调试 38
3.2.3 结合 SmartRF Flash Programmer 软件实现程序下载 39
3.2.4 读取 CC2540 蓝牙设备的 IEEE 地址 41
3.3 本章小结 42
[扩展阅读] 天线基础理论 42

第 4 章　CC2540 基础实验

4.1　流水灯基础实验　46

4.2　定时器 / 计数器基础实验　49

4.3　定时器 / 计数器高级实验——定时器中断　53

4.4　UART 串口基础实验　57

4.5　UART 串口高级实验　61

4.6　UART 串口编程高级技巧　68

4.7　AD 转换实验之测量 VDD 实验　71

4.8　AD 转换实验之温度采集实验　75

4.9　本章小结　78

[扩展阅读]　可变参数函数的使用技巧　78

第 5 章　蓝牙 4.0 BLE 技术入门

5.1　蓝牙 4.0 BLE 协议栈　81

5.1.1　什么是蓝牙 4.0 BLE 协议栈　81

5.1.2　如何使用蓝牙 4.0 BLE 协议栈　82

5.1.3　蓝牙 4.0 BLE 协议栈的安装、编译与下载　82

5.2　深入理解蓝牙 4.0 BLE 协议栈的构成　84

5.2.1　BLE 协议栈基础　84

5.2.2　BLE 协议栈开发平台配置　85

5.2.3　BLE 协议栈执行流程浅析　87

5.3　操作系统抽象层　88

5.3.1　OSAL 常用术语　88

5.3.2　OSAL 运行机理　90

5.3.3 OSAL 消息队列 97
5.3.4 OSAL 添加新任务 98
5.3.5 OSAL 应用编程接口 99
5.4 硬件抽象层 102
5.4.1 硬件抽象层文件目录 102
5.4.2 硬件抽象层驱动编译 104
5.4.3 用户自定义驱动 105
5.5 配置文件层 105
5.5.1 通用访问配置文件（GAP） 106
5.5.2 通用属性配置文件（GATT） 108
5.5.3 使用 GAP 和 GATT 的堆栈 API 110
5.6 蓝牙 4.0 BLE 协议栈中串口应用详解 110
5.6.1 串口收发实验 111
5.6.2 实例测试 113
5.6.3 串口工作原理剖析 118
5.7 本章小结 124
[扩展阅读] OSI 七层模型及协议栈分层思想 124

第 6 章 蓝牙 4.0 BLE 无线网络技术

6.1 蓝牙无线网络节点设备示例工程 128
6.1.1 工程概述 128
6.1.2 初始化 129
6.1.3 周期事件 132
6.1.4 外设状态通知回调 134
6.1.5 按键服务 136
6.1.6 LCD 显示 138

6.2 蓝牙无线网络集中器示例工程 140
6.2.1 工程概述 140
6.2.2 初始化 141
6.2.3 事件处理 143
6.2.4 回调函数 153
6.2.5 服务发现 157
6.3 网络处理器工程简介 167
6.3.1 工程概述 167
6.3.2 BLE 协议栈的外部设备控制 168
6.4 本章小结 168
[扩展阅读] 探究接收数据的存放位置 169

第 7 章 PC 端调试开发工具

7.1 启动 BToll 工具 173
7.2 创建 BToll 与蓝牙节点设备的连接 175
7.2.1 设置节点设备的可发现状态 175
7.2.2 扫描节点设备 177
7.2.3 连接参数设置 178
7.2.4 建立连接 178
7.3 GATT 数据服务 180
7.3.1 通过 UUID 读取特性 180
7.3.2 通过句柄写入特性值 183
7.3.3 通过句柄读取特性值 184
7.3.4 通过 UUID 发现特性 185
7.3.5 通过句柄读取多个特性值 186

7.3.6 特性值通知功能 188
7.4 自定义 GATT 数据服务 191
7.4.1 节点设备编程 191
7.4.2 实例测试 202
7.5 本章小结 207
[扩展阅读] GATT 数据服务的通知与指示 207

第 8 章 蓝牙 4.0 BLE 无线数据传输分析

8.1 蓝牙 4.0 BLE 数据包的捕获 210
8.1.1 构建蓝牙 4.0 BLE 协议分析仪 210
8.1.2 蓝牙 4.0 BLE 数据包的结构 212
8.2 蓝牙 4.0 BLE 数据传输实验剖析 213
8.2.1 实验原理及流程图 213
8.2.2 节点设备发送的广告数据 214
8.2.3 节点设备扫描回应的数据 215
8.2.4 蓝牙 4.0 BLE 应用数据传输流程分析 217
8.2.5 数据收发实验回顾 225
8.3 本章小结 226
[扩展阅读] 蓝牙 4.0 BLE 协议栈数据包格式 226

第 9 章 蓝牙 4.0 BLE 无线网络技术拓展

9.1 BLE 协议栈的中断处理机制 230
9.1.1 中断处理编程 230
9.1.2 实例测试 237
9.2 获取广告设备地址 238

9.2.1 广告设备编程 239
9.2.2 集中器设备编程 244
9.2.3 实例测试 249
9.3 无线温度采集系统设计 250
9.3.1 实验原理及流程图 250
9.3.2 节点设备编程 250
9.3.3 集中器设备编程 254
9.4 蓝牙 4.0 BLE 无线传感器网络组建 256
9.4.1 集中器与多个节点设备建立连接 257
9.4.2 实例测试 263
9.5 本章小结 264
[扩展阅读] 倒 F 天线设计 264

参考文献

第1章

蓝牙4.0 BLE简介

近年来，无线网络得到了快速的发展，在此过程中也出现了各种无线网络数据传输标准，诸如WiFi™、Wireless USB、ZigBee，不同的协议标准对应不同的应用领域，例如，WiFi™主要用于大量数据的传输；Wireless USB主要用于视频数据的传输；ZigBee主要用于短距离无线网络控制系统，传输少量控制信息等。

蓝牙无线技术是使用范围最广泛的全球短距离无线标准之一，蓝牙技术联盟（SIG）于2010年7月发布了蓝牙4.0版本核心规范，以低功耗（BLE）作为新版本的主要技术特点，蓝牙迈入4.0时代。随着iPhone 4S的发布，世界上首款支持蓝牙4.0标准的手机面世。各大厂商紧跟潮流，纷纷推出支持蓝牙4.0的智能设备。TI公司已经推出了完全兼容蓝牙4.0 BLE协议的SoC芯片CC2540，同时也开发出了相应的软件协议栈，开发者可以使用上述硬件和软件资源，搭建自己的应用开发平台。

本章主要讨论蓝牙4.0 BLE的产生、发展过程，向读者展示蓝牙4.0 BLE的特点以及相关应用领域，帮助初学者快速入门。

1.1 无线网络数据传输协议对比

现在比较流行的无线网络数据传输协议有WiFi™、Wireless USB、Bluetooth™、Cellular等，不同的协议都有各自的应用领域，因此，选择网络协议时，要根据不同的应用来选择某一种特定的协议。

各种无线数据传输协议对比图如图1-1所示。

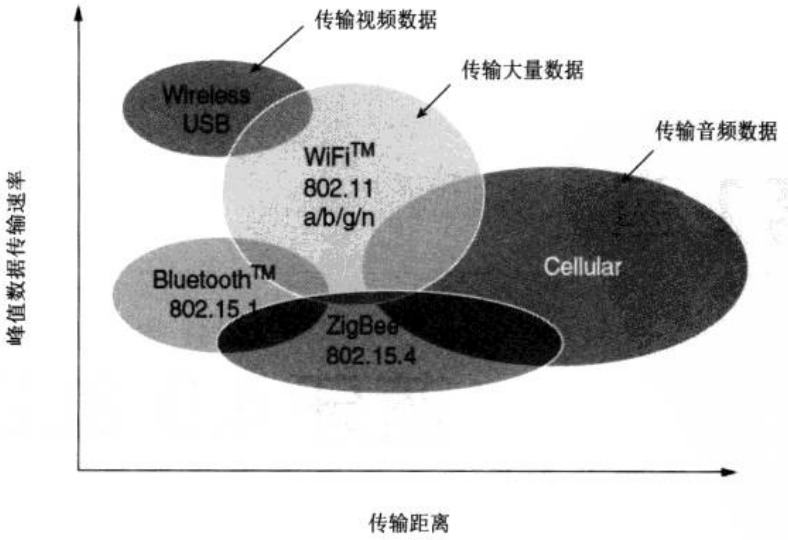


图1-1 各种无线数据传输协议对比图

从图 1-1 中可以看到不同的无线数据传输协议在数据传输速率和传输距离有各自的使用范围。

传统蓝牙、ZigBee 以及 IEEE 802.11b 标准都是工作在 2.4GHz 频段的无线通信标准，下面将传统蓝牙与 ZigBee、IEEE 802.11b 标准进行简要的比较，帮助读者快速地了解几种无线传输标准的应用领域。

▲ 传统蓝牙数据传输速率小于 3Mbps，典型数据传输距离为 2 ～ 10 m，典型应用是在两台蓝牙设备之间进行小量数据的传输。

▲ IEEE 802.11b 最高数据传输速率可达 11Mbps，典型数据传输距离在 30 ～ 100 m，IEEE 802.11b 技术提供了一种 Internet 的无线接入技术，如很多笔记本电脑可以使用自带的 WiFi 功能实现上网。

▲ ZigBee 协议可以理解为一种短距离无线传感器网络与控制协议，传输速率低，主要用于传输控制信息，数据量相对来说比较小，适用于电池供电的系统。此外，相对于上述两种标准，ZigBee 协议实现成本较低。

传统蓝牙、ZigBee 以及 IEEE 802.11b 标准对比情况如表 1-1、图 1-2 所示。

表 1-1　传统蓝牙、ZigBee 以及 IEEE 802.11b 标准对比

数据传输协议	数据速率	数据传输距离 /m	典型应用领域
传统蓝牙	1-3Mbps	2 ~ 10	无线手持设备、无线鼠标
ZigBee	20-250Kbps	10 ~ 100	无线传感器网络
IEEE 802.11b	1-11Mbps	30 ~ 100	无线 Internet 接入

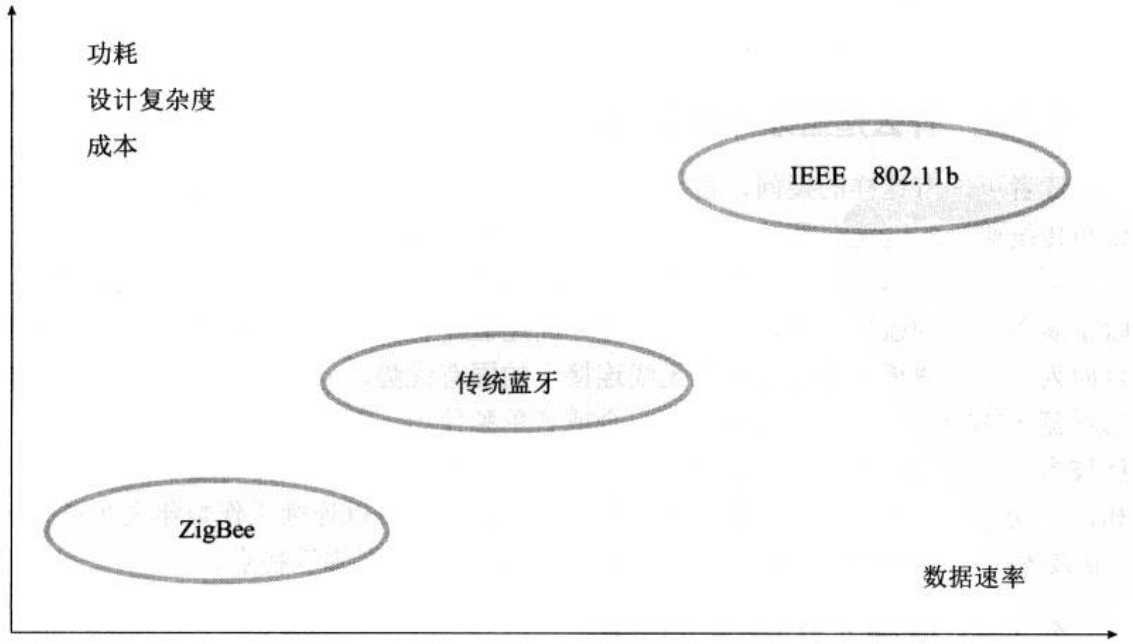


图 1-2　传统蓝牙、ZigBee 以及 IEEE 802.11b 标准对比

1.2 短距离无线网络的分类

短距离无线网络主要分为两类：

▲ 无线局域网（WLANs，Wireless Local Area Networks）；

▲ 无线个域网（WPANs，Wireless Personal Area Networks）。

无线局域网是有线局域网（LANs，Wired Local Area Networks）的扩展，一个无线局域网设备可以很容易地接入到有线局域网。

无线个域网是为了在POS（Personal Operating Space）范围内提供一种高效、节能的无线通信方法，其中POS是指以无线设备为中心半径10m（33ft）内的球形区域。

按照数据传输速率的不同，无线个域网又分为三种：

▲ HR-WPLANS——High-Rate WPLANS；

▲ MR-WPLANS——Medium-Rate WPLANS；

▲ LR-WPLANS——Low-Rate WPLANS。

上述三类个域网所对应的协议如表1-2所示。

表1-2 个域网所对应的通信协议

无线个域网	通信协议
HR-WPANS	802.15.3
MR-WPANS	BlueTooth
LR-WPANS	802.15.4

1.2.1 什么是蓝牙4.0 BLE

读者可能有这样的疑问，蓝牙4.0协议与上述协议有什么关系呢，蓝牙4.0协议与传统蓝牙的区别在哪里呢，蓝牙4.0协议主要用在哪些应用领域呢？

蓝牙无线技术是使用范围最广泛的全球短距离无线标准之一，全新的蓝牙4.0版本涵盖了三种蓝牙技术，即传统蓝牙、高速蓝牙和低功耗蓝牙技术，将三种规范合而为一。它继承了蓝牙技术在无线连接上的固有优势，同时增加了高速蓝牙和低功耗蓝牙的特点。这三个规格可以组合或者单独使用。蓝牙4.0规范的核心是低功耗技术（Low Energy），即蓝牙4.0 BLE。该技术最大特点是拥有超低的运行功耗和待机功耗，蓝牙低功耗设备使用一粒纽扣电池甚至可以连续工作数年之久；蓝牙4.0技术同时还拥有低成本、向下兼容、跨厂商互操作性强等特点。

1.2.2 蓝牙4.0 BLE的特点

总体而言，蓝牙4.0 BLE技术具有如下特点。

（1）高可靠性

对于无线通信而言，由于电磁波在传输过程中容易受很多因素的干扰，例如，障碍物的阻挡、天气状况等，因此，无线通信系统在数据传输过程中，具有内在的不可靠性。

蓝牙技术联盟（SIG）在制定蓝牙 4.0 规范时已经考虑到了这种数据传输过程中的内在的不确定性，在射频、基带协议、链路管理协议（LMP）中采用可靠性措施，包括：差错检测和校正、进行数据编解码、差错控制、数据加噪等，极大地提高了蓝牙无线数据传输的可靠性。另外，使用自适应跳频技术，最大程度地减少和其他 2.4GHz ISM 频段无线电波的串扰。

（2）低成本、低功耗

低功耗蓝牙支持两种部署方式：双模式和单模式[●]。双模式中，低功耗蓝牙功能集成在现有的经典蓝牙控制器中，或在现有经典蓝牙技术（2.1+EDR/3.0+HS）芯片上增加低功耗堆栈，整体架构基本不变，因此成本增加有限。

单模式面向高度集成、紧凑的设备，使用一个轻量级链接层（Link Layer）提供超低功耗的待机模式操作。蓝牙 4.0 BLE 技术可以应用于 8-bit MCU，目前 TI 公司推出的兼容蓝牙 4.0 BLE 协议的 SoC 芯片 CC2540，外接 PCB 天线和几个阻容器件构成的滤波电路即可实现蓝牙网络节点的构建。

低功耗设计：蓝牙 4.0 版本强化了蓝牙在数据传输上的低功耗性能，功耗较传统蓝牙降低了百分之九十。

▲ 传统蓝牙设备的待机耗电量大一直是其缺陷之一，这与传统蓝牙技术采用 16 ～ 32 个频道进行广播不无关系，而低功耗蓝牙仅使用了 3 个广播通道，且每次广播时射频的开启时间也由传统的 22.5ms 减少到 0.6 ～ 1.2ms，这两个协议规范的改变，大幅降低了因为广播数据导致的待机功耗。

▲ 低功耗蓝牙设计了用深度睡眠状态来替换传统蓝牙的空闲状态，在深度睡眠状态下，主机（Host）长时间处于超低的负载循环（Duty Cycle）状态，只在需要运作时由控制器来启动，由于主机较控制器消耗的能源更多，因此这样的设计也节省了最多的能源。

（3）快速启动，瞬间连接

此前蓝牙版本为人诟病的地方就在于启动速度方面，蓝牙 2.1 版本的启动连接需要 6s 时间，而蓝牙 4.0 版本仅仅需要 3ms 即可完成，几乎是瞬间连接。

● 双模方式可同时支持传统蓝牙（基本速率及增强速率）和蓝牙低功耗技术，目前使用蓝牙技术的所有产品都可以采用该模式，如手机等；单模方式只支持蓝牙低功耗技术，主要应用于蓝牙新型产品，如蓝牙血压计、心率监控仪等。

（4）传输距离极大提高

传统蓝牙传输距离为2～10m，而蓝牙4.0的有效传输距离可达到60～100m，传输距离提升了十倍，极大开拓了蓝牙技术的应用前景。

（5）高安全性

为了保证数据传输的安全性，使用AES-128 CCM加密算法进行数据包加密和认证，对于初学阶段，安全性问题可以暂不考虑。

1.3 蓝牙4.0 BLE协议简介

蓝牙4.0 BLE规范中定义了GAP(Generic Access Profile)和GATT（Generic Attribute）两个基本配置文件。

▲ 协议栈中的GAP层负责设备访问模式和进程，包括设备发现、建立连接、终止连接、初始化安全特性和设备配置。

▲ 协议栈中的GATT层用于已连接的蓝牙设备之间的数据通信。

> **注意 ▶▶▶**
>
> 关于蓝牙4.0 BLE协议的分层模型和技术细节，请读者参见本书后续章节以及实验部分，笔者力图给读者展现一个循序渐进的学习过程，相信很多读者对于无线网络软件架构的分层思想还没有基本的认识，因此，在此没有过多地对蓝牙4.0 BLE协议栈的知识进行讲解，笔者会选择合适的实验环境，将这部分知识展示给读者。

1.4 蓝牙4.0 BLE无线网络通信信道分析

天线对于无线通信系统来说至关重要，在日常生活中可以看到各式各样的天线，如手机天线、电视接收天线等，天线的主要功能可以概括为：完成无线电波的发射与接收。发射时，把高频电流转换为电磁波发射出去；接收时，将电磁波转换为高频电流。

如何区分不同的电波呢？

一般情况，不同的电波具有不用的频谱，无线通信系统的频谱有几十兆赫兹到

几千兆赫兹，包括了收音机、手机、卫星电视等使用的波段，这些电波都使用空气作为传输介质来传播，为了防止不同的应用之间相互干扰，就需要对无线通信系统的通信信道进行必要的管理。

各个国家都有自己的无线电管理结构，如美国的联邦通信委员会（FCC）、欧洲的典型标准委员会（ETSI），我国的无线电管理机构称为中国无线电管理委员会，其主要职责是负责无线电频率的划分、分配与指配、负责卫星轨道位置协调和管理、负责无线电监测、检测、干扰查处、协调处理电磁干扰事宜和维护空中电波秩序等。

一般情况，使用某一特定的频段需要得到无线电管理部门的许可，当然，各国的无线电管理部门也规定了一部分频段是对公众开放的，不需要许可即可使用，以满足不同的应用需求，这些频段包括 ISM（Industrial,Scientific and Medical——工业、科学和医疗）频带。

在 IEEE 802.15.4 中总共规定了 27 个信道：

- ▲ 在 2.4GHz 频段，共有 16 个信道，信道通信速率为 250kbps；
- ▲ 在 915MHz 频段，共有 10 个信道，信道通信速率为 40kbps；
- ▲ 在 896MHz 频段，有 1 个信道，信道通信速率为 20kbps。

注意 ▶▶▶

2.4GHz 是全球通用的 ISM 频段，915MHz 是北美的 ISM 频段，896MHz 是欧洲认可的 ISM 频段。bps 是指每秒钟传输的二进制位数，即比特每秒。

ISM 频段信道分布图如图 1-3 所示。

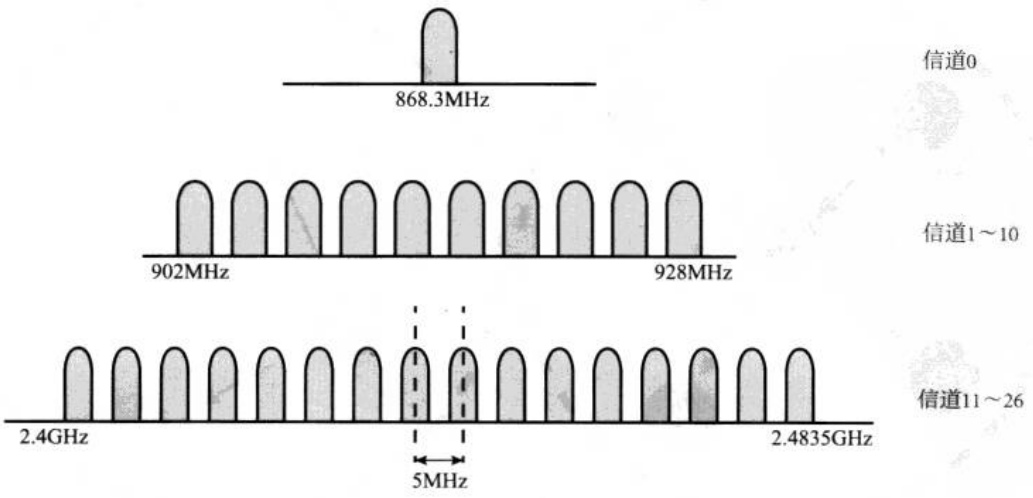


图 1-3　信道分布图

各国对无线频谱的管理不仅规定了 ISM 频带的频率，同时也规定了在这些频带上所使用的发射功率，在项目开发过程中，需要查阅相关的手册，如我国信息产业部发布的《微功率（短距离）无线电设备管理规定》。

蓝牙 4.0 BLE 工作在 2.4GHz ISM 频段，仅使用了 3 个广播通道，使用所有蓝牙规范版本通用的自适应跳频技术，最大程度地减少与其他 2.4GHz ISM 频段无线电波的串扰。

自适应跳频技术是建立在自动信道质量分析基础上的一种频率自适应和功率自适应控制相结合的技术。它能使跳频通信过程自动避开被干扰的跳频频点，并以最小的发射功率、最低的被截获概率，达到在无干扰的跳频信道上长时间保持优质通信的目的。所谓频率自适应控制是在跳频通信过程中，拒绝使用那些曾经用过但是传输不成功的跳频频率集中的频点，即实时去除跳频频率集中被干扰的频点，使跳频通信在无干扰的可使用的频点上进行，从而大大提高跳频通信中接收信号的质量。

1.5 蓝牙 4.0 BLE 无线网络拓扑结构

蓝牙 4.0 BLE 网络拓扑结构分为星型拓扑和广播组拓扑。不同的网络拓扑对应不同的应用领域，在蓝牙 4.0 BLE 无线网络中，不同的网络拓扑结构对网络节点的配置有不同的要求（蓝牙网络节点的类型可以分为主机、从机，也可以分为服务器、客户端，具体配置需要根据配置文件决定），在本书后面章节将进行讲解，在此，读者只需要对蓝牙 4.0 BLE 网络拓扑结构有个概念性的认识即可。

星型网络拓扑图如图 1-4 所示。广播组拓扑图如图 1-5 所示。

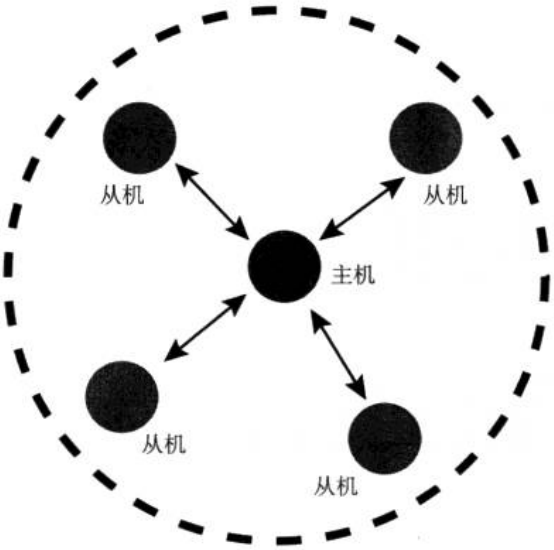


图 1-4　星型网络拓扑图

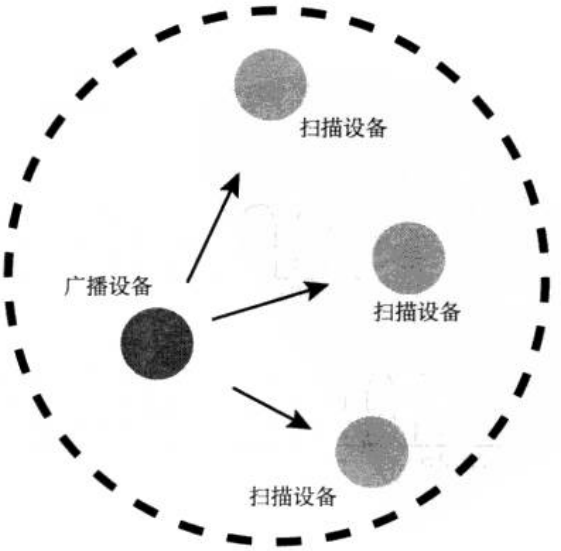


图 1-5　广播组拓扑图

1.6 蓝牙 4.0 BLE 技术在物联网的应用

蓝牙 4.0 的推出使蓝牙技术凭借低功耗特性进入物联网和医疗保健等应用，而且因在手机中的高占有率，使其在那些以手机为核心的物联网应用中或将更具优势。移动医疗设备市场将是首个受益于蓝牙 4.0 低功耗技术的设备市场，而健身行业、无线智能家居、汽车电子行业也将蓄势待发。

下面给读者展示几个当前市场上蓝牙 4.0 低功耗技术应用的例子。

在医学应用领域，可以借助蓝牙 4.0 BLE 技术，准确、有效地监测病人的血压、体温等信息。在病人监控系统中，病人的血压、心率可以通过特定的传感器检测，因此，可以将传感器与蓝牙 4.0 BLE 设备相连，蓝牙 4.0 BLE 设备定期检测病人的血压、心率，并将数据以无线的方式发送到服务器，服务器将数据传输到医生的电脑上，医生就可以实时监控并根据病人的血压、心率数据进行恰当的诊断，如图 1-6 所示。

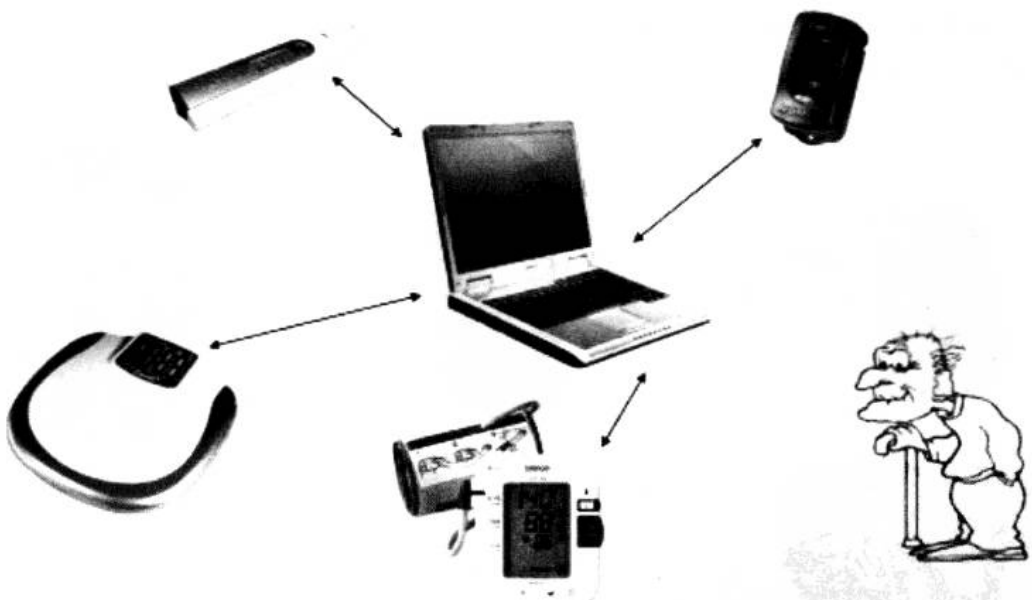


图 1-6　医学领域应用

在智能家居和商业楼宇自动化方面，将空调、电视、DVD、窗帘控制器等设备通过蓝牙 4.0 BLE 技术来组成一个无线网络，通过手机上的应用软件就可以实现对各种家电设备的控制，这种应用相比现行的每个家电配备一个遥控器要方便的多。智能手机、平板电脑装置也可通过蓝牙技术，来控制家电、瓦斯系统和空调系统的开关，并可透过手机、平板屏幕或电视屏幕了解用电状况、环境温湿度等信息，如图 1-7 所示。

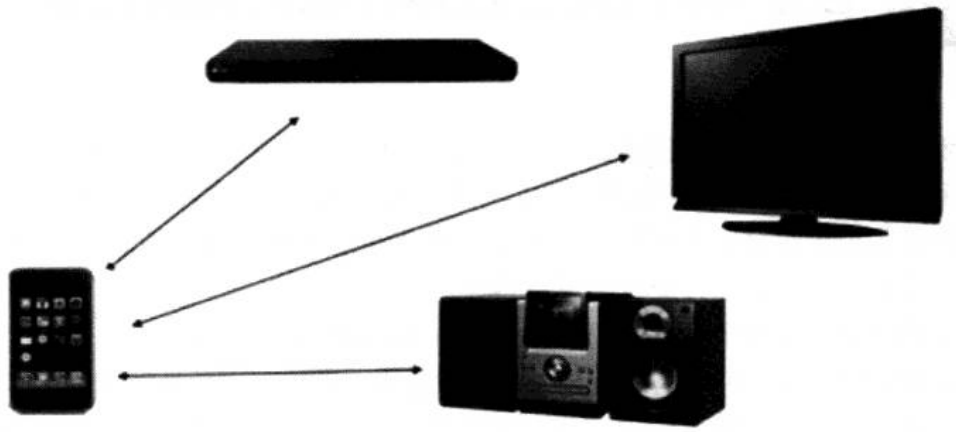


图 1-7　无线智能家居控制

在汽车电子领域，汽车制造商可以将压力传感器整合到轮胎中，以支持汽车诊断应用。采用蓝牙低功耗技术的胎压传感器可以挂载于轮胎或车轮内，通过蓝牙无线通信让驾驶员知道何时该更换轮胎。这类应用使得消费者能以最方便、最安全的方式掌握关键的车况，如图 1-8 所示。

图 1-8　蓝牙低功耗技术在汽车电子领域应用

在不久的将来，将会看到很多蓝牙 4.0 低功耗的体育和健身设备（如手表、心率监测器、计步器）、保健设备（如体重秤、血压计、血糖仪）、消费类电子产品（如 3D 眼镜、先进的远程控制）、智能家居设备（如恒温器、安全系统）等各式各类型的设备面世，蓝牙 4.0 低功耗技术未来还会有很大的发展。

1.7 CC2540 开发板硬件资源概述

进行蓝牙 4.0 BLE 应用的开发，需要有相关的硬件和软件，在硬件方面，TI 公司[●]已经推出了完全支持蓝牙 4.0 BLE 协议的单片机 CC2540，同时也推出了相应的开发套件；在软件方面，TI 公司也推出了相应的低功耗协议栈。在国内也有很多公司推出了蓝牙 4.0 BLE 相关的开发套件，下面以智造者科技有限公司推出的一款 CC2540-EB 开发板为例进行一些硬件方面的概述。

CC2540-EB 开发板如图 1-9 所示，分为底板和核心板，这种设计方式可以满足用户的不同需求。

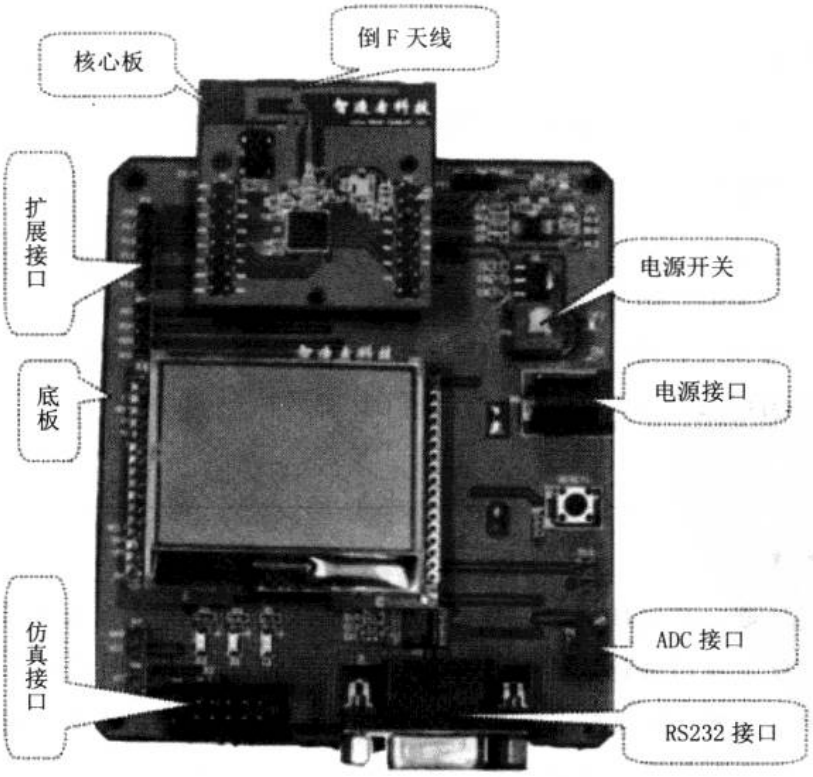


图 1-9　CC2540-EB 开发板（带液晶）

● 德州仪器（TI）是模拟及数字半导体 IC 设计制造公司。除了提供模拟技术、数字信号处理（DSP）和微处理器（MCU）半导体以外，TI 还设计制造用于模拟和数字嵌入式处理及应用于终端设备的半导体解决方案。

硬件资源主要有以下几个部分。

（1）电源接口

实现开发板供电，CC2540单片机正常工作需要的电压2～3.6V，一般输入电压典型值为3.3V。

（2）ADC接口

可以方便用户进行AD采样实验。

（3）RS232接口

串口通信电路接口，可以方便用户进行串口实验，可以使用串口输出相应的数据，便于调试使用。

（4）仿真接口

10针JATG接口，可以实现程序的在线仿真、调试、下载功能，可以结合SmartRF Flash Programmer软件进行程序的下载；此外，在蓝牙4.0 BLE无线通信实验过程中，可以结合Packet Sniffer软件进行无线数据抓包实验。

（5）天线

射频天线使用了全尺寸的倒F天线（IFA），极大地提高了发射、接收性能。

（6）扩展接口

开发板预留了I/O端口，用户可以使用这些I/O扩展口进行外接传感器实验，例如，可以外接ADXL345加速度传感器模块，进行加速度传感器实验。

此外，还提供了一个128*64点阵液晶模块，使用此液晶可以方便的进行数据的显示，便于用户的开发。

注意 ▶▶▶

本节只是对开发板进行宏观的讲解，具体电路的设计在本书第3章还将进行详细的讲解，读者只需要明白：进行蓝牙4.0 BLE应用的开发，上述资源已经足够了。笔者结合自身的开发经验给读者阐述这种观点：当什么都看不清的时候，尽量从宏观上把握，随着学习过程的深入，一些问题会自然而然地清晰起来。

1.8 本章小结

本章主要讲述了蓝牙4.0低功耗（BLE）的基础知识，包括蓝牙4.0 BLE的特点、网络拓扑、通信信道等，此外还给出CC2540-EB开发板的部分功能模块图，使读者对硬件模块有一个整体的概念。笔者力图从宏观上把握这部分知识点，把一些更细节、更具体的内容放在本书的后面章节进行讲解，尽量使每一个知识点出现在最恰当的应用环境中。

[扩展阅读] 蓝牙技术发展历程

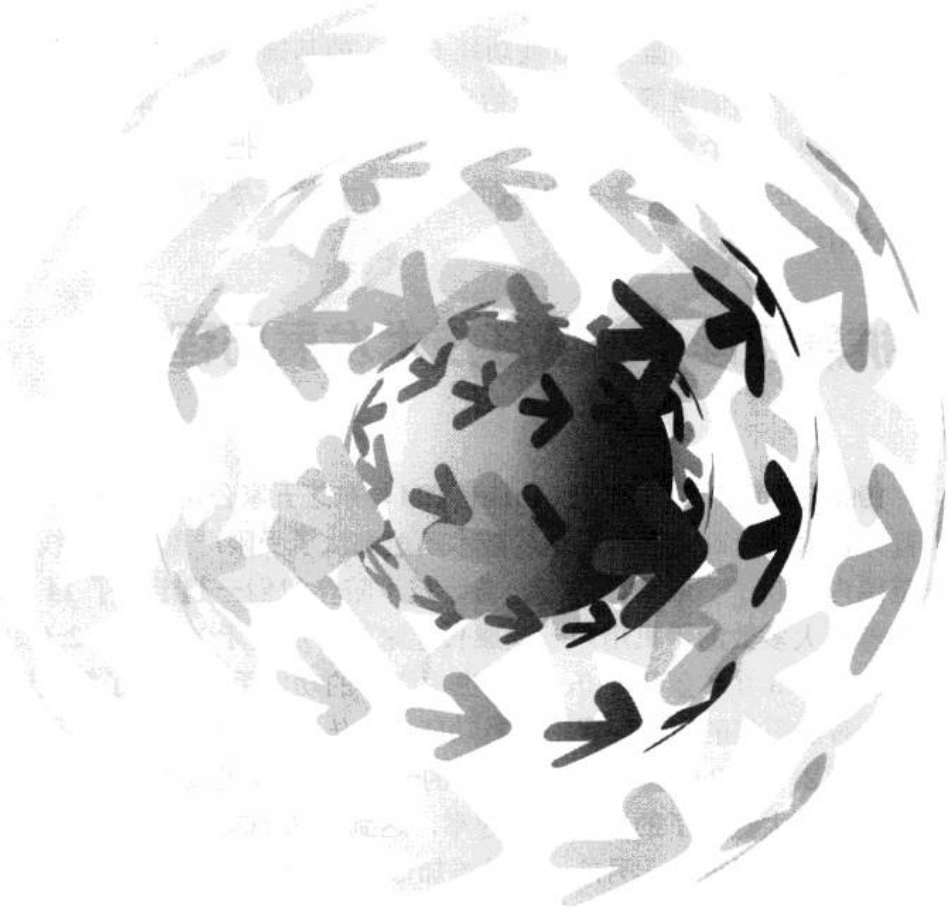
1998年，爱立信、诺基亚、Intel、IBM与东芝五家公司发起成立了蓝牙技术联盟，当蓝牙技术规范确定之后，这项技术迅速淘汰了受限严重的红外线传输技术，在全球范围内掀起了一股蓝牙热潮。蓝牙技术陪伴我们一起走过了14年，这项技术是伴随着绝大多数的手机用户一起成长起来的，从蓝牙1.1/1.2/2.0/2.1，到近几年推出的蓝牙3.0和蓝牙4.0，手机也由当年的功能手机步入了智能时代。蓝牙技术已成为智能手机的标准配置，并且功能也从传输文件扩展到蓝牙耳机、键盘等。

蓝牙2.1+EDR是蓝牙发展进程中的一个里程碑，也是目前应用最为广泛的标准，其特色是大大提高了的数据传输速率，达到了2.1Mbps，还可充分利用带宽优势同时连接多个蓝牙设备。另外，通过添加低耗电监听模式极大降低了功耗。

蓝牙3.0+HS作为新版本规范，数据传输速率得到了极大提高。通过集成“802.11 PAL”（协议适应层），蓝牙3.0的数据传输率提高到了24Mbps，即在需要的时候通过WiFi实现高速数据传输，可以轻松用于录像机至高清电视、PC至播放器、PC至打印机之间的资料传输。

蓝牙4.0技术是一个革命性突破，是三位一体的蓝牙技术，它将传统蓝牙、高速蓝牙和低功耗蓝牙技术融合在一起，这三个规格可以组合使用也可以单独使用。蓝牙4.0最重要的特色是低功耗技术，极低的运行和待机功耗可以使一粒纽扣电池连续工作数年之久。此外，还具有低成本和跨厂商互操作性、3ms启动、100m以上超长传输距离、AES-128安全加密等诸多特色，可应用于对成本和功耗都有严格要求的无线方案，广泛用于医疗保健、体育健身、家庭娱乐、传感器物联网等众多领域。

任何技术的创新与突破都是为了能让我们生活的更轻松、便利。蓝牙技术发展至今，已经由最初的文件传输功能发展到全方位、多领域的应用，它令我们的生活方式发生了许多改变。全球业界不断开发和推出基于蓝牙技术的应用产品，使蓝牙技术呈现出极其广阔的市场前景。



第 2 章

IAR 集成开发环境

由于蓝牙 4.0 低功耗协议的发布，以及 TI 公司推出的协议栈逐渐完善，市场上出现了各种各样的蓝牙 4.0 低功耗解决方案，但是对于初学蓝牙 4.0 低功耗技术的读者来说，如何准确地选择一款适合自己的开发平台显得至关重要。

大多数读者对于 51 内核的单片机较为熟悉，笔者结合自身学习经验给读者推荐以下开发平台：

▲ 选用一套 C8051 内核的单片机；

▲ 使用 IAR 8.10 软件集成开发环境；

▲ 蓝牙 4.0 BLE 协议栈。

其中，C8051 内核的单片机要支持蓝牙 4.0 BLE 协议，这是进行蓝牙 4.0 BLE 无线通信开发的硬件平台，IAR 开发环境用于蓝牙协议栈应用程序的编写。

本章主要讲述如何使用 IAR 开发环境及 CC2540 单片机进行开发，CC2540 单片机是TI 公司推出的兼容蓝牙 4.0 BLE 协议的无线射频单片机，只需要外接一个天线，即可实现蓝牙 4.0 BLE 应用的开发。如果读者已经熟悉 IAR 开发环境，完全可以跳过本章，直接进行后面章节的学习。

2.1 IAR 集成开发环境简介

IAR Embedded Workbench（又称做EW）的C交叉编译器是一款完整、稳定且很容易使用的专业嵌入式应用开发工具。EW对不同的微处理器提供统一的用户界面，目前可以支持至少35种的8位、16位、32位ARM微处理器结构。

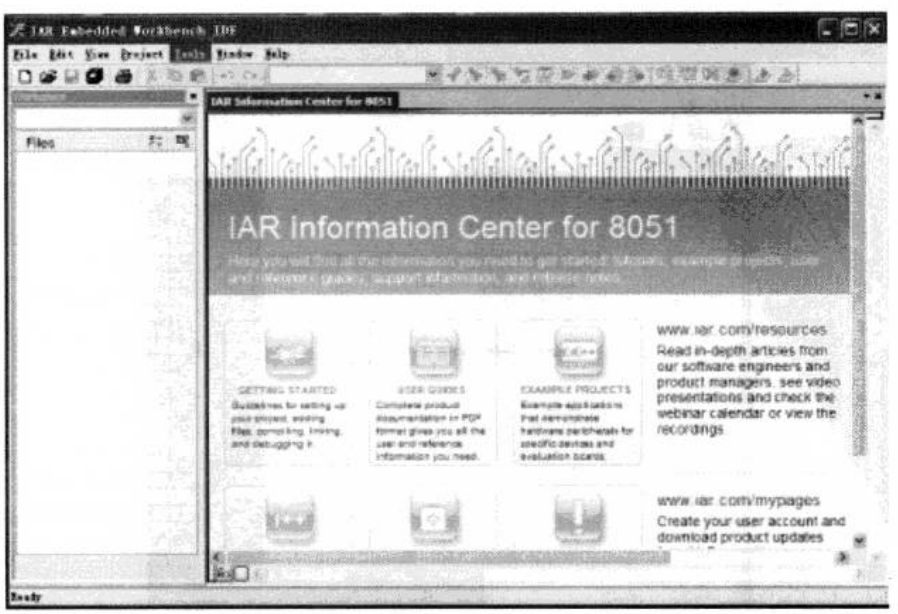


图 2-1 IAR Embedded Workbench 软件界面

IAR Embedded Workbench 集成的编译器有以下特点：

▲ 完全兼容标准 C 语言；

▲ 内建相应芯片的程序速度和内部优化器；

▲ 高效浮点支持；

▲ 内存模式选择；

▲ 高效的 PRO Mable 代码。

IAR Embedded Workbench 软件界面如图 2-1 所示。

安装 IAR Embedded Workbench 软件的方法，跟其他 Windows 程序安装方法一致，在此不作赘述，下面着重讲解 IAR 集成开发环境中工程的相关操作。

2.2 工程的编辑与修改

IAR 集成开发环境中，对应工程的编辑操作主要涉及以下几方面的内容：

▲ 如何建立、保存一个工程；

▲ 如何向工程中添加源文件；

▲ 如何编译源文件。

下面进行详细讲解。

2.2.1 建立一个新工程

打开 IAR 集成开发环境，单击菜单栏的 Project，在弹出的下拉菜单中选择 Create New Project，如图 2-2 所示。

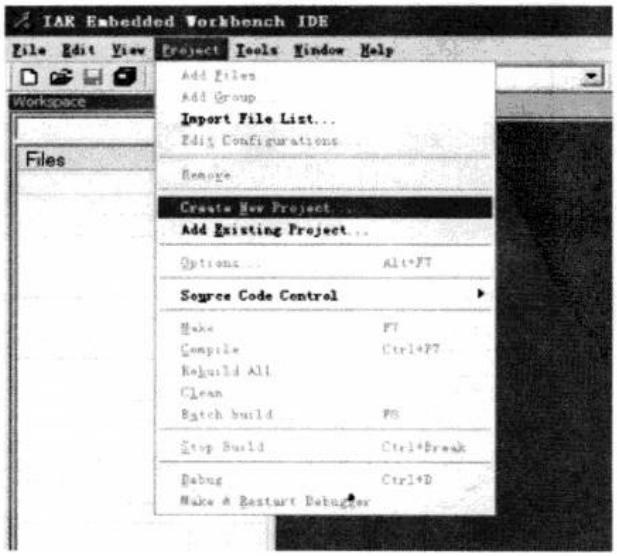


图 2-2 选择 Create New Project

此时，系统会弹出 Create New Project 对话框，在 Tool chain 后面的下拉列表框中选择 8051，然后在 Project templates 列表框中选择 Empty project，最后单击 OK 按钮即可。Create New Project 对话框设置如图 2-3 所示。

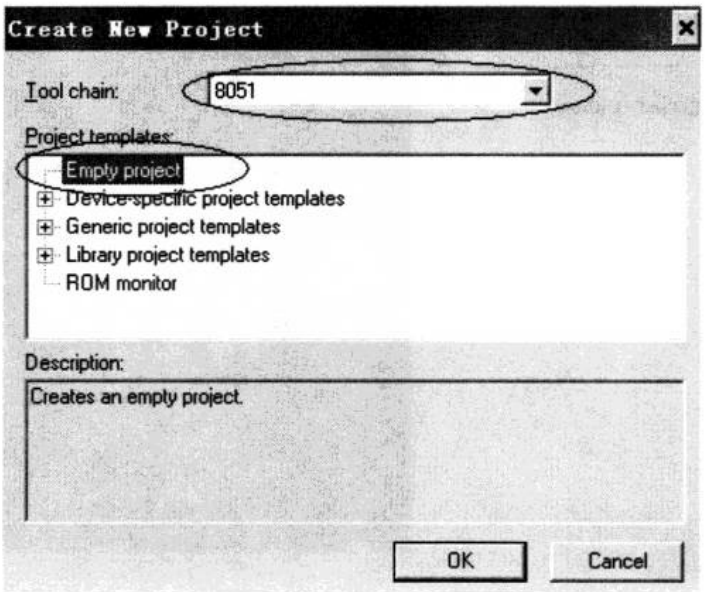


图 2-3　Create New Project 对话框设置

此时，系统会弹出另存为对话框如图 2-4 所示，根据用户需要可以自行更改工程名和保存位置。

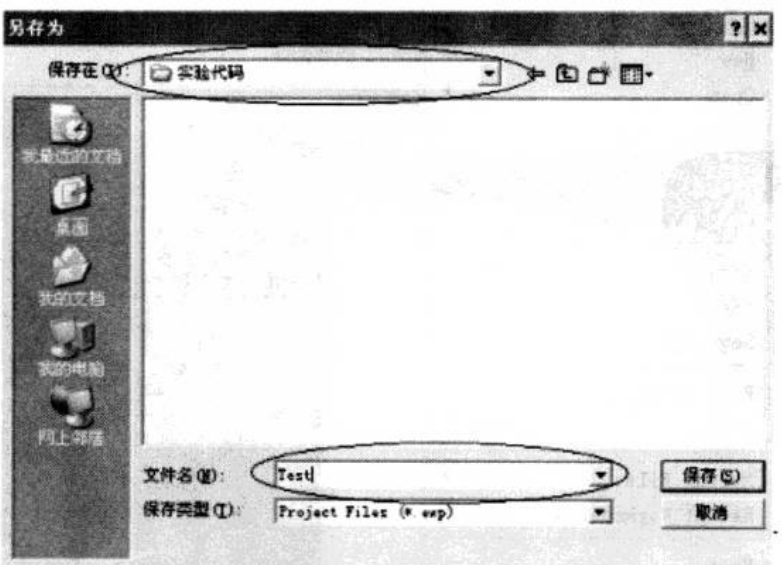


图 2-4　另存为对话框

此时，新建工程窗口如图 2-5 所示。

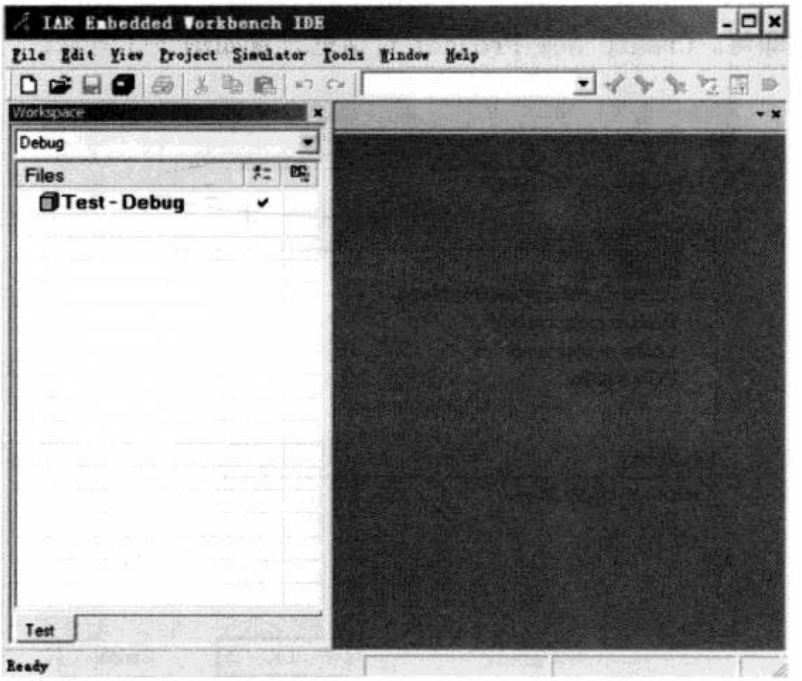


图 2-5　新建工程主窗口

选择菜单栏上的 File，在弹出的下拉菜单中选择 Save Workspace，如图 2-6 所示。

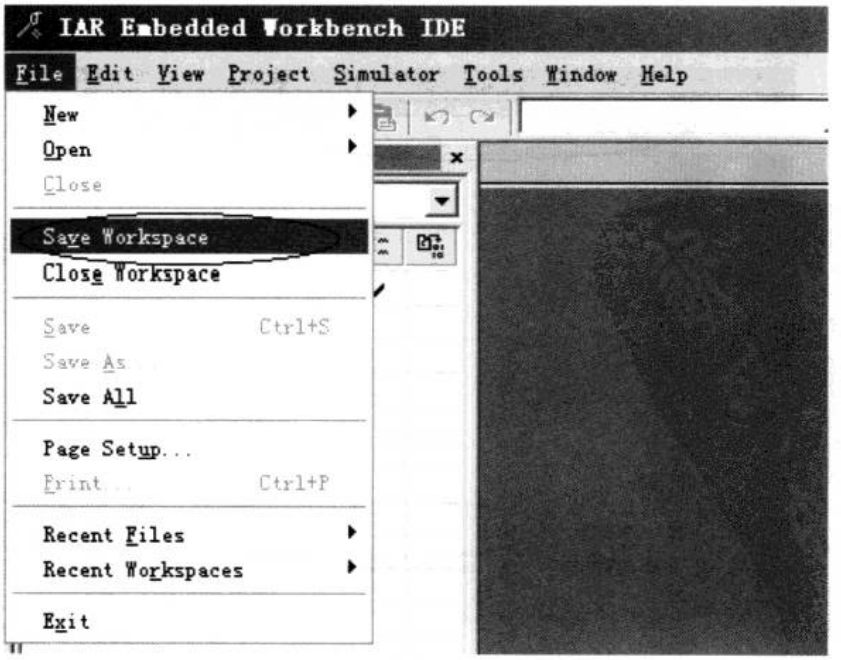


图 2-6　选择 Save Workspace

在弹出的 Save Workspace As 对话框中选择保存位置，输入文件名即可，保存 Workspace 如图 2-7 所示。

图 2-7　保存 Workspace

2.2.2　建立一个源文件

接下来需要添加源文件到该项目，选择File→New→File，新建源文件如图2-8所示。

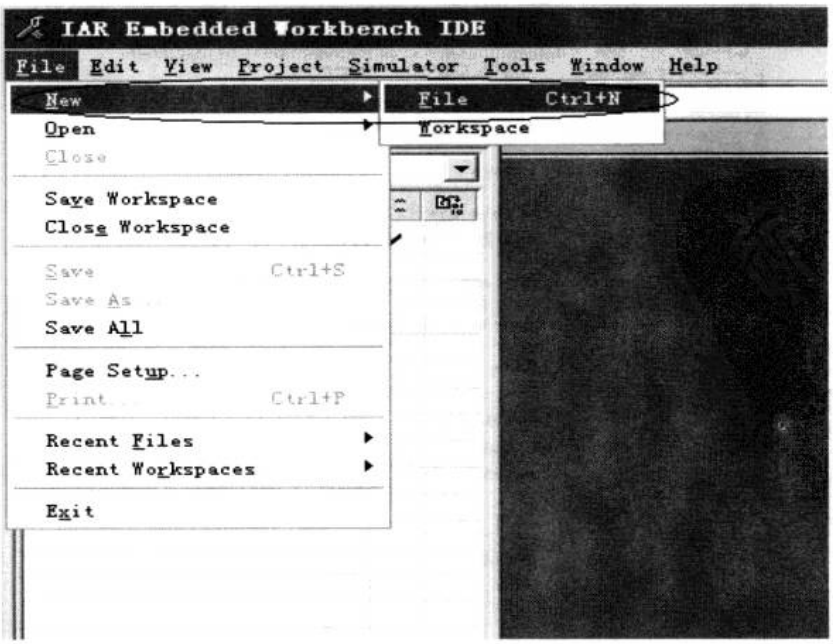


图 2-8　新建源文件

然后，将源文件保存为Led.c，保存源文件如图2-9所示。

图2-9 保存源文件

2.2.3 添加源文件到工程

将上述源文件添加到项目中，选择Project→Add Files，添加源文件如图2-10所示。

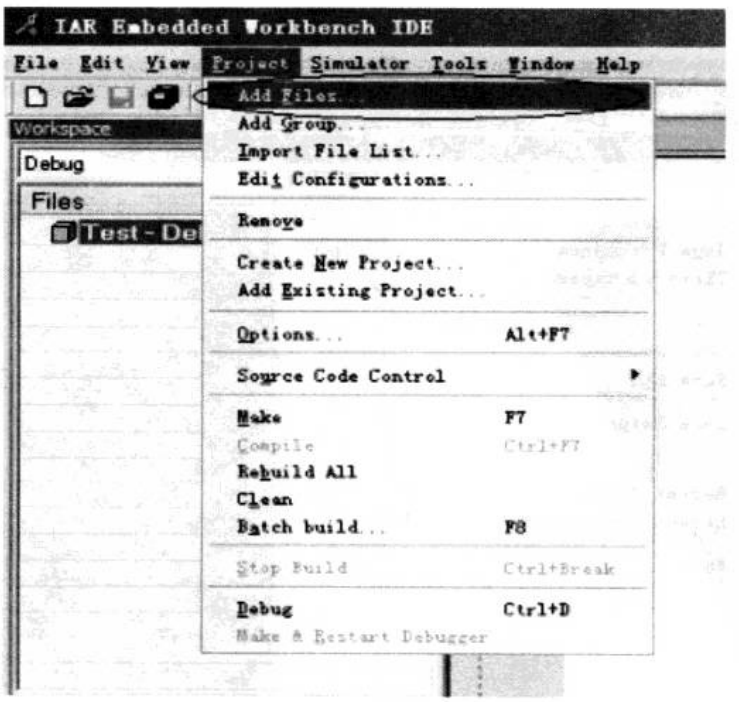


图2-10 添加源文件

在弹出的对话框中，选择 Led. c 即可，如图 2-11 所示。

图 2-11　选择 Led.c

此时，项目左边的 Workspace 栏已经发生了变化，Workspace 栏如图 2-12 所示。

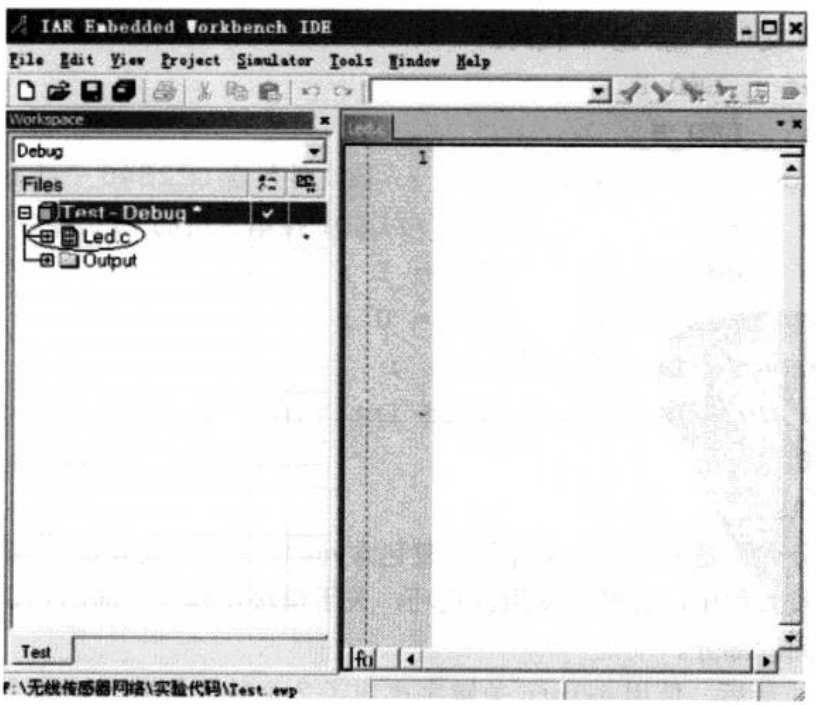


图 2-12　Workspace 栏

然后，按照前文讲述的向工程中添加源文件的方法，向该工程中添加 Led. h、

main.c 文件，Test 工程文件布局如图 2-13 所示。

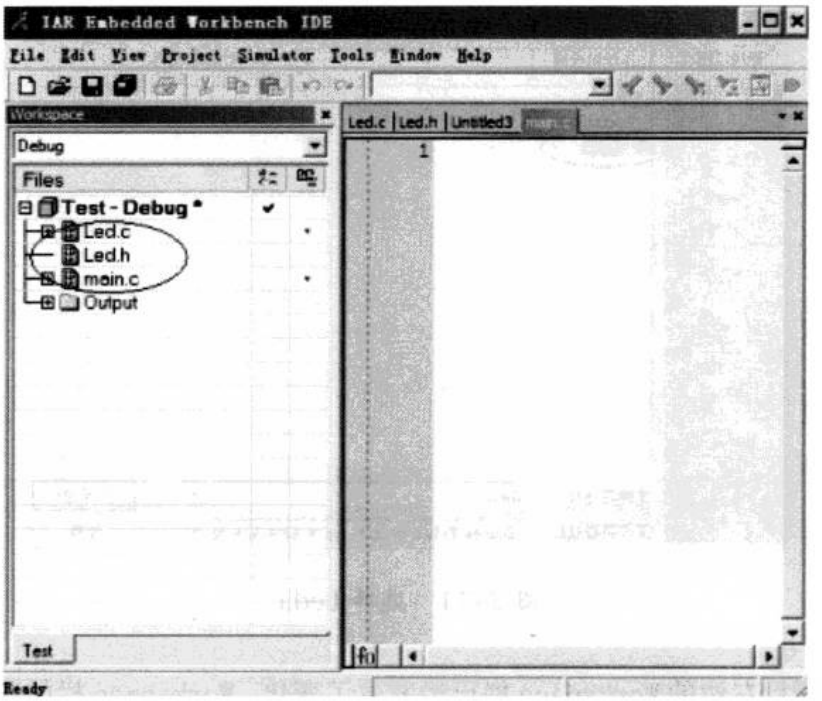


图 2-13　Test 工程文件布局

向 Led.h 文件中输入以下代码：

```
#ifndef __LED_H_
#define __LED_H_
#include <ioCC2540.h>        // 该文件包含 CC2540 一些寄存器宏定义
#define  LED1          P1_0  //LED1 接单片机的 P10 端口
#define  Led1_On()     LED1 = 1 ;
#define  Led1_Off()    LED1 = 0 ;
extern void Led_Init(void) ;
extern void Delay(unsigned int time) ;
#endif
```

程序说明：

加粗部分代码是为了防止文件的重复包含问题，读者只需要记住这种格式即可，在模块化开发过程中，会经常使用该技巧，关于模块化编程方面的知识点请读者参见本章扩展阅读部分。

在程序的最后，使用 extern 关键字声明了 2 个外部函数，这两个函数的实现放在 Led.c 文件中。

Led.c 文件内容如下：

```
#include <ioCC2540.h>        // 该文件包含 CC2540 一些寄存器宏定义
```

```
#include "led.h"
void Led_Init(void)
{
  P1SEL &= ~ (1 << 0) ;   // 将 p1_0 设置为通用 I/O 口
  P1DIR |= (1 << 0) ;     // 将 p1_0 设置为输出模式
  LED1 = 0 ;              //LED1 初始化为熄灭状态
}
void Delay(unsigned int time)
{
    unsigned int i,j;
    for(i = 0 ; i < time ; i++)
      for(j = 0 ; j < 10000; j++) ;
}
```

程序说明：

在 Led_Init() 函数中用到了寄存器 P1SEL 和 P1DIR。

关于这两个寄存器的详细使用方法请读者参考 CC2540 单片机数据手册。

main.c 文件内容如下：

```
#include "Led.h"
void main(void)
{
   Led_Init() ;
   while(1)
   {
      Led1_On() ;
      Delay(10) ;
      Led1_Off() ;
      Delay(10) ;
   }
}
```

程序说明：

技巧提示：该程序中，要使用到 LED 初始化函数，而该函数又是在 Led.c 文件中实现的，在 Led.h 文件中使用 extern 关键字对其进行了声明，那么，在 main.c 文件中需要使用 LED 初始化函数，则只需要包含 Led.h 文件即可，即 #include “Led.h”。

注意 ▶▶▶

上述程序实现的基本功能是：在主循环中，点亮LED1，然后延时一段时间，然后熄灭LED1，然后再点亮……

2.2.4 工程设置

IAR集成开发环境支持多种处理器，因此，建立工程后，要对工程进行基本的设置，使其符合用户所使用的单片机。

单击菜单栏上的Project，在弹出的下拉菜单中选择Options，如图2-14所示。

图 2-14 选择 Options

此时，弹出的Options for node “Test”对话框如图2-15所示。

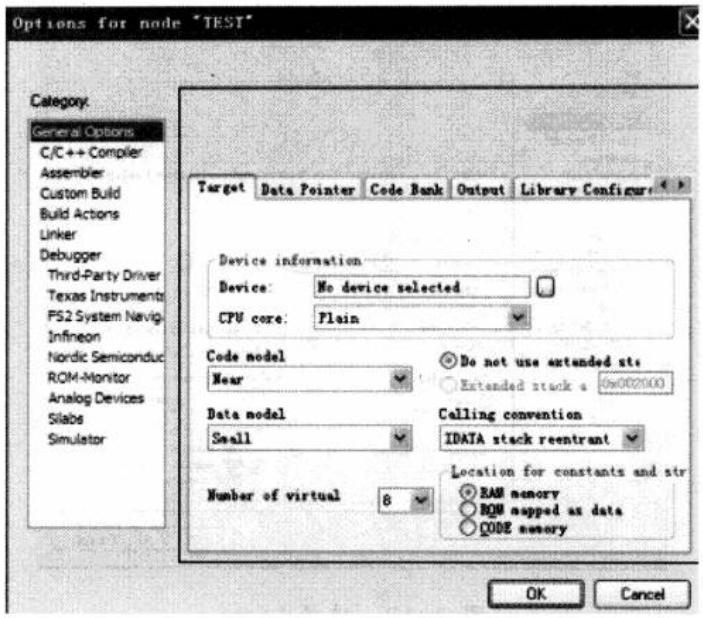


图 2-15　弹出的 Options for node "Test" 对话框

(1) General Options 选项

在 Target 标签下，Device 栏选择"Texas Instruments"文件夹下 CC2540F256.i51，如图 2-16 所示；Data model 栏的下拉菜单选择"Large"，如图 2-17 所示。

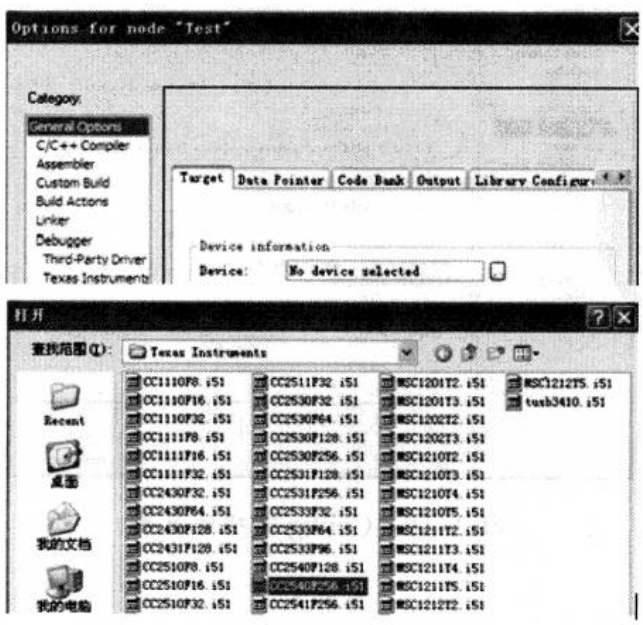


图 2-16　选择 CC2540F256.i51

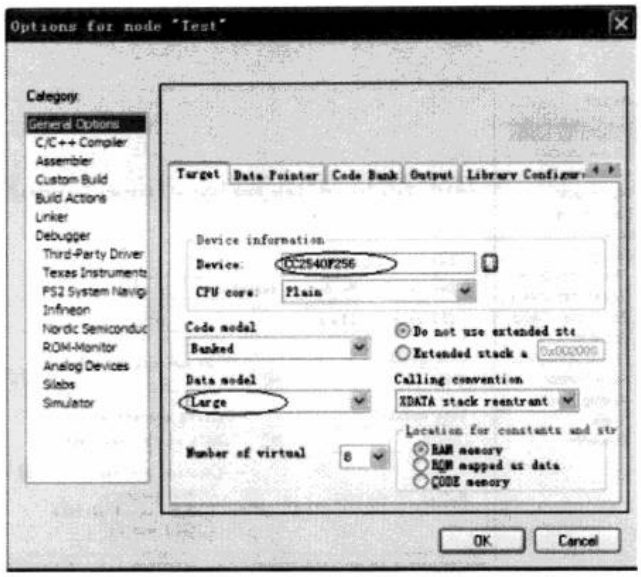


图 2-17　选择 Large

(2) Linker 选项

Config 标签的设置如下：勾选 Linker command file 栏“Override default”，单击右边的省略号按钮，在弹出的打开对话框中选择“lnk51ew_CC2540F256_banked.xcl”，Config 选项的设置如图 2-18 所示。

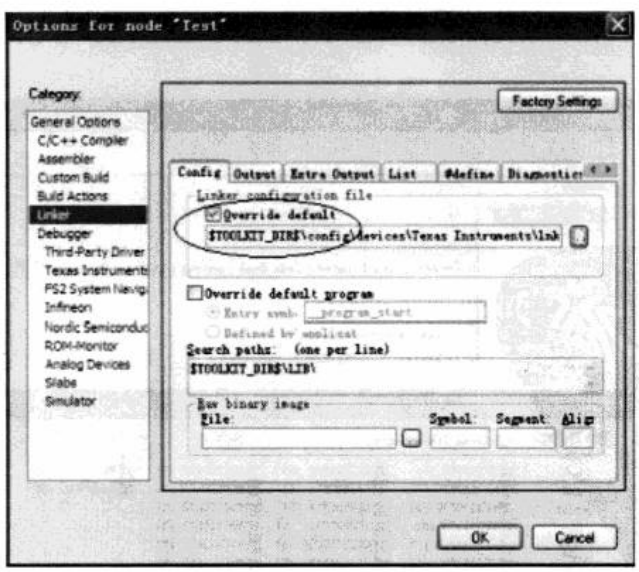


图 2-18　Config 选项的设置

Output 标签下的选项主要用于设置输出文件名及格式，将 Output file 标签下面的文本框中输入“Test.hex”。如果需要生成仿真调试格式的“.hex”文件，

勾选“Allow C-SPY-specific extraoutput file”（特别注意：如果需要生成供SmartRF Flash Programmer下载的“.hex”文件，要在Format 选项选择“Other”项）。Output 标签的设置如图 2-19 所示。

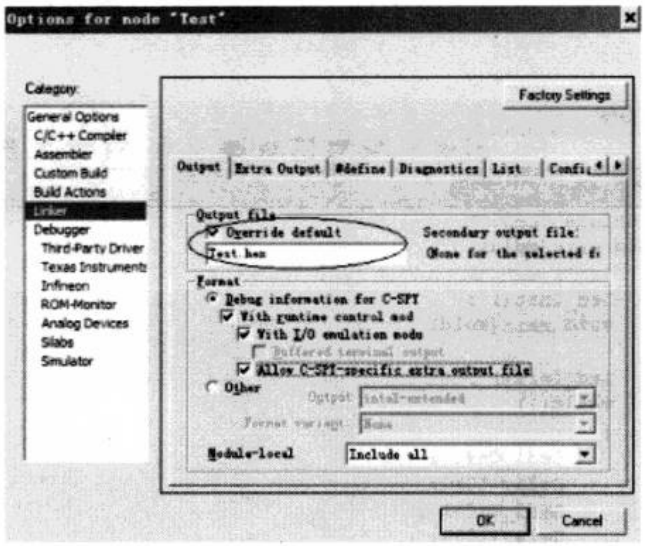


图 2-19　Output 标签的设置

（3）Debugger 选项

Setup 标签下 Driver 栏设置为“Texas Instruments”，勾选“Device Description File”项，Setup 标签的设置如图 2-20 所示。

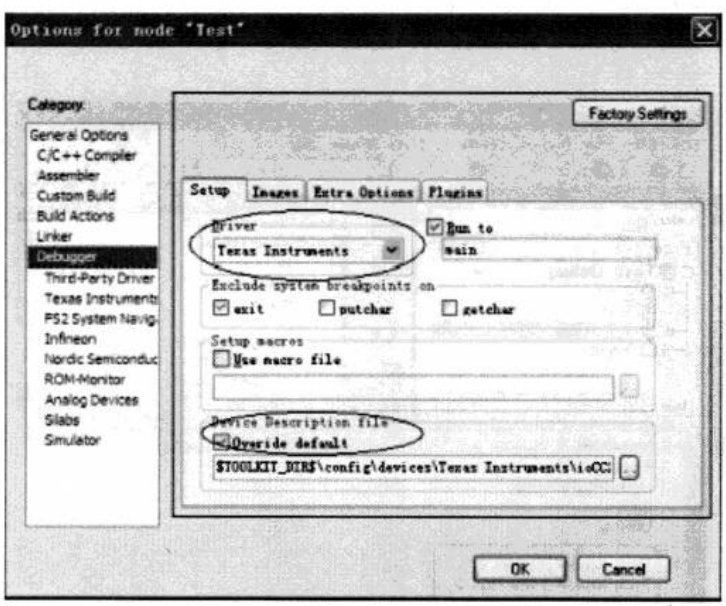


图 2-20　Setup 标签的设置

然后单击OK按钮即可完成所有的配置工作。

2.2.5 源文件的编译

设置好工程后，接下来需要对工程中的源文件进行编译，单击Make图标如图2-21所示。

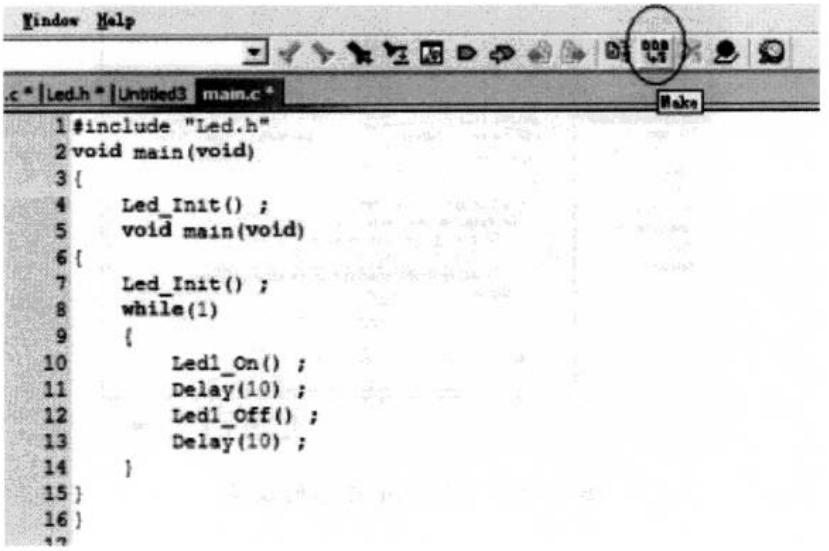


图 2-21 单击 Make 图标

如果源文件编译完成，则此时在IAR集成开发环境的左下角会弹出Message窗口，该窗口中显示源文件的错误和警告信息，Message窗口如图2-22所示。

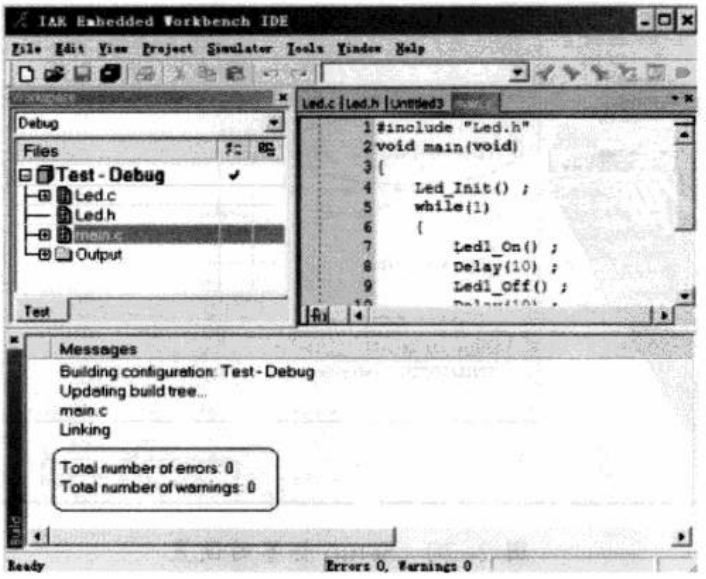


图 2-22 Message 窗口

2.3 本章小结

本章主要讲述了使用 IAR 集成开发环境进行应用程序开发的基本流程，讲解了工程的建立、源文件的添加及编译。

[扩展阅读] 模块化编程技巧

在单片机开发过程中，经常遇到模块复用问题，这时模块化编程将大大加快产品的开发进度，此外，TI 公司推出的蓝牙 4.0 BLE 协议栈也是以模块化编程为基础进行的设计，学好模块化编程对于产品的开发以及蓝牙 4.0 BLE 协议栈的学习都有较大的帮助作用，下面对模块化编程进行简要的讲解。

模块化编程分析与设计的基本理论如下。

在理想的模块化编程中，每个模块都可以看做是一个黑盒子，只需要了解模块提供的功能，不需要关心具体实现该功能的策略和方法，即提供的是机制而不是策略，机制即功能，策略即方法。好比用户买了一部 iPhone，只需要会用它所提供的各种功能即可，至于各种功能在底层是如何实现的，用户不需要关心。

在大型程序开发中，一个程序由不同的模块组成，可能不同的模块会由不同的人员负责。在编写某个模块的时候，很可能需要调用别人写好的模块的接口。这个时候关心的是：其他模块提供了什么样的接口，应该如何去调用，至于模块内部是如何实现的，对于调用者而言，无须过多关注。模块对外提供的只是接口，把不需要的细节尽可能对外部屏蔽起来，这正是采用模块化程序设计所需要注意的地方。

一个模块包含两个文件：一个是“.h”文件（又称为头文件）；另一个是“.c”文件。

“.h”文件可以理解为一份接口描述文件，其文件内部一般不包含任何实质性的函数代码，可以把这个头文件理解成为一份说明书，其内容就是这个模块对外提供的接口函数或接口变量。

此外，该文件也可以包含一些很重要的宏定义（如前文中 Led.h 中实现的宏 Led1_On()）以及一些数据结构的信息，离开了这些信息，该模块提供的接口函数或接口变量很可能就无法正常使用。

头文件的基本构成原则是：不该让外界知道的信息就不应该出现在头文件里，而供外界调用的模块内接口函数或接口变量所必需的信息就一定要出现在头文件里，否则，外界就无法正确地调用该模块提供的功能。

当外部函数或者文件调用该模块提供的接口函数或变量时，就必须包含该模块

提供的这个接口描述文件——“.h”文件（头文件）。同时，该模块的“.c”文件也需要包含这个模块头文件（因为它包含了模块源文件中所需要的宏定义或数据结构等信息）。

通常，头文件的名字应该与源文件的名字保持一致，这样便可以清晰地知道哪个头文件是对哪个源文件的描述。

“.c”文件主要功能是对“.h”文件中声明的外部函数进行具体的实现，对具体实现方式没有特殊规定，只要能实现其函数的功能即可。

第 3 章

CC2540 开发板硬件资源

进行蓝牙4.0 BLE应用开发，首先，需要有相应的硬件支持（支持蓝牙4.0 BLE协议栈的硬件）；此外还需要相应的软件支持（支持蓝牙4.0 BLE协议的软件协议栈）。当然，还需要下载器将程序下载到相应的硬件平台。本章主要讲解硬件电路方面的设计及程序的仿真调试、下载方法。

3.1 CC2540核心板硬件资源

CC2540-EB核心板主要包括CC2540单片机、全尺寸倒F天线、晶振以及I/O扩展接口，CC2540-EB核心板如图3-1所示。

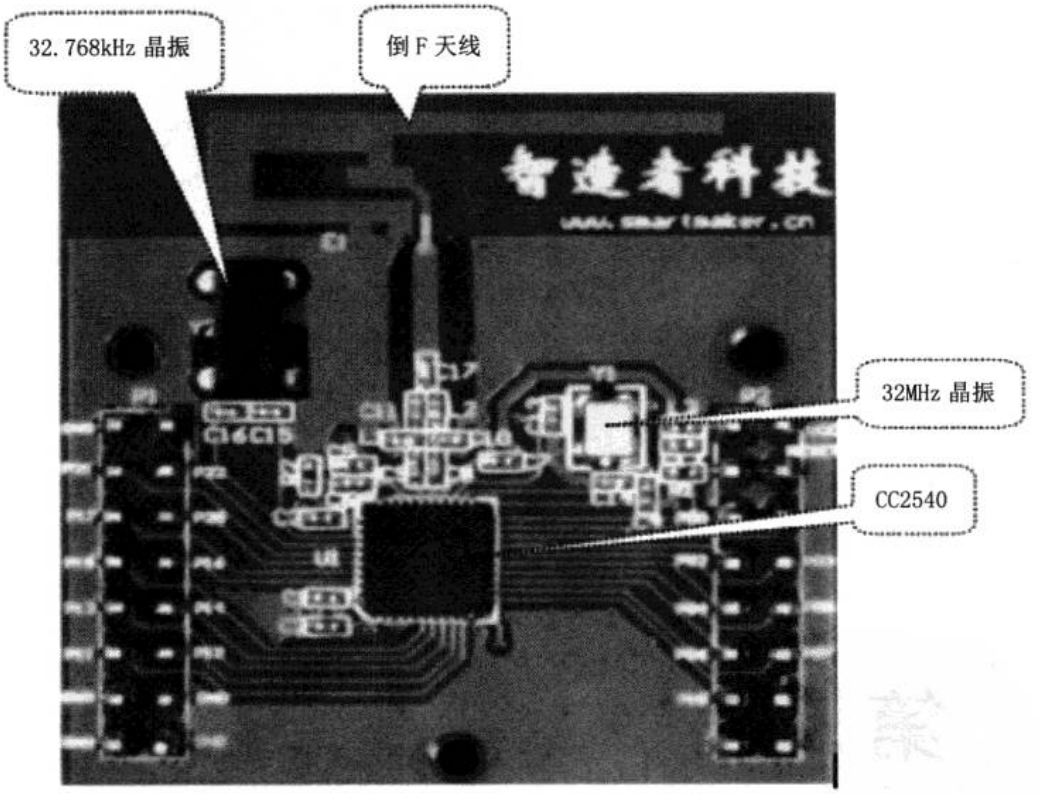


图3-1 CC2540-EB核心板

3.1.1 CC2540简介

CC2540集成了2.4GHz射频收发器，是一款完全兼容8051内核的无线射频单片机，它与蓝牙低功耗协议栈共同构成高性价比、低功耗的片上系统（SoC）解决方案，非常适合蓝牙低功耗应用。它有3个不同的存储器访问总线：

▲ 特殊功能寄存器（SFR）；

▲ 数据（DATA）；

▲ 代码 / 外部数据（CORE/XDATA）。

CC2540 单片机使用单周期访问 SFR、DATA 和主 SRAM。当 CC2540 处于空闲模式时，任何的中断可以把 CC2540 恢复到主动模式。某些中断还可以将 CC2540 从睡眠模式唤醒。位于系统核心存储器交叉开关使用 SFR 总线将 CPU、DMA 控制器与物理存储器和所有的外接设备连接起来。

CC2540 的 Flash 容量可以选择，有 128kB、256kB，这就是 CC2540 单片机的在线可编程非易失性存储器，并且映射到代码和外部数据存储器空间。除了保存程序代码和常量以外，非易失性存储器允许应用程序保存必要的数据，以保证这些数据在设备重启后可用。

3.1.2 天线及巴伦匹配电路设计

在基于蓝牙 4.0 BLE 协议的应用开发过程中，天线以及巴伦匹配电路的设计较为重要，这涉及到射频通路指标是否优良，对通信距离、系统功耗都有较大影响。

天线设计可以使用 PCB 天线，如：倒 F 天线、螺旋天线等，也可以使用 SMA 接口的杆状天线，根据不同的应用来选择。天线及巴伦匹配电路设计如图 3-2 所示。

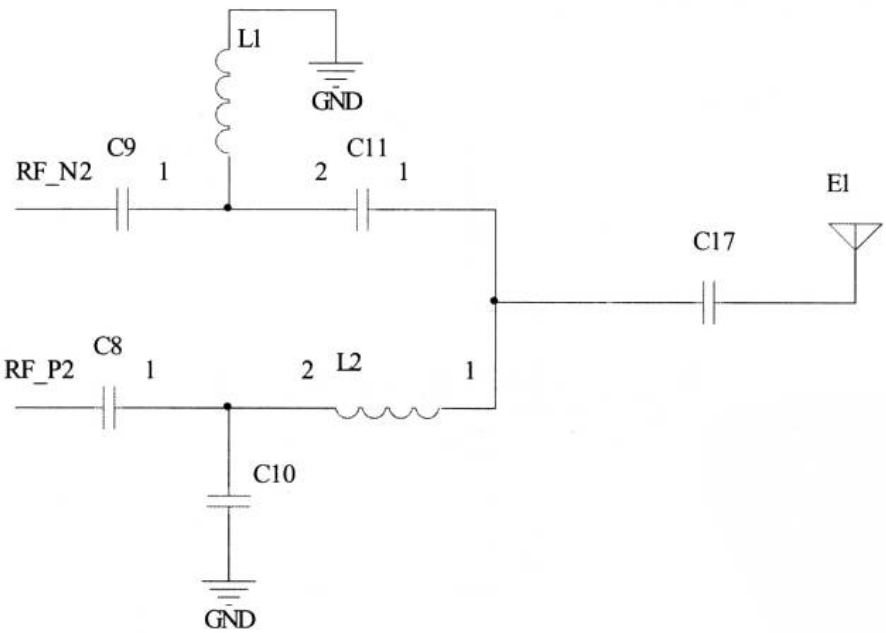


图 3-2 天线及巴伦匹配电路设计

3.1.3 晶振电路设计

CC2540 需要 2 个晶振，32MHz 晶振和 32.768kHz 晶振，晶振电路接口如图 3-3 所示。

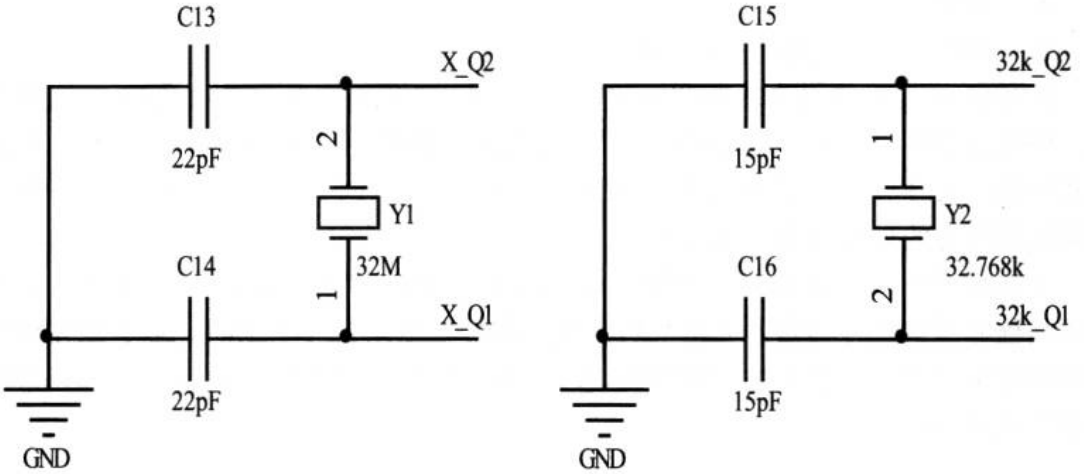


图 3-3 晶振电路接口

3.1.4 电源电路设计

电源电路可以采用 5V 电源通过 DC-DC 变换器得到 3.3V 工作电压，此外，也可以采用 2 节 5 号电池供电的方案，电源电路如图 3-4 所示。

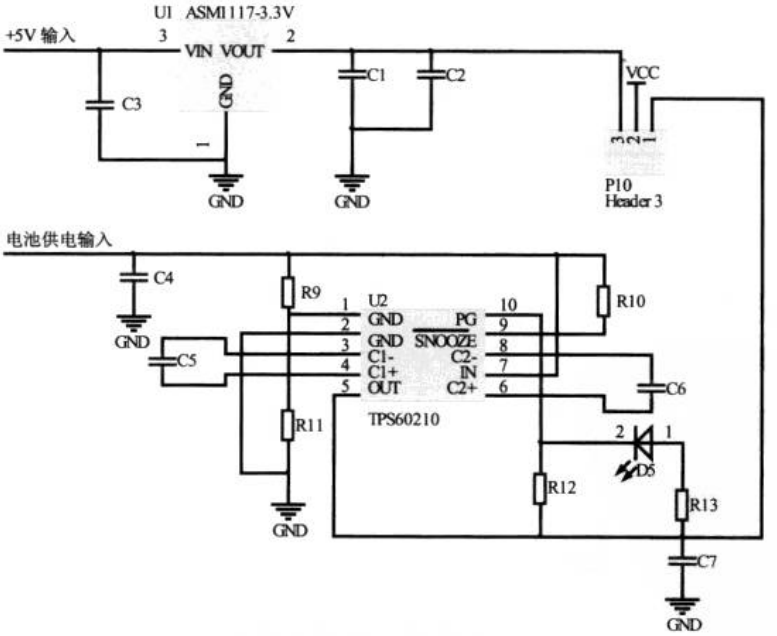


图 3-4 电源电路

3.1.5 LED 电路设计

LED 主要用于指示电路的工作状态，如：建立连接、信号良好、正在传输数据

等信息，LED 电路如图 3-5 所示。

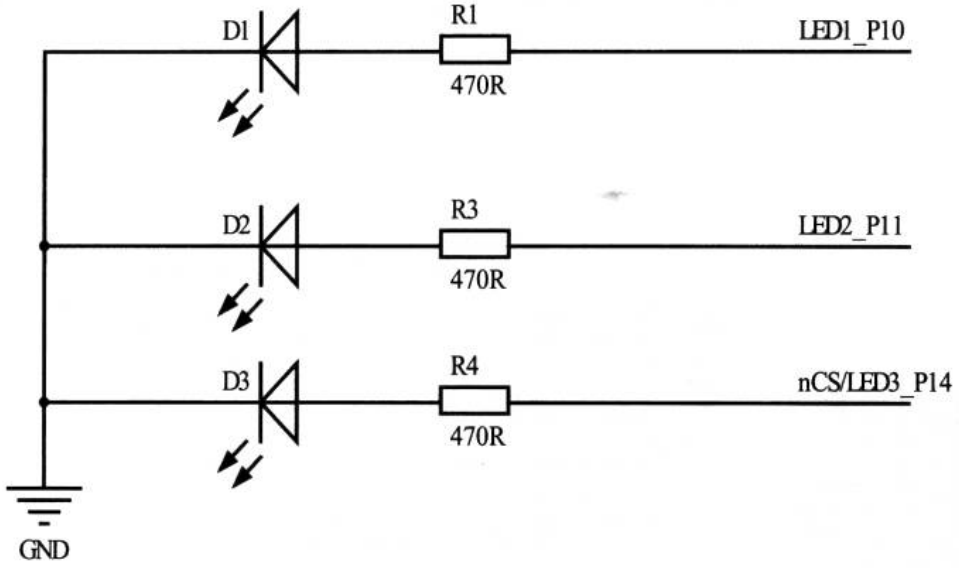


图 3-5　LED 电路

3.1.6　AD 转换电路设计

AD 转换电路主要用于在实验中模拟传感器，通过调节滑动变阻器的阻值大小，来改变电位器电压，可以在实验部分测试开发板上的 AD 输出电压值，AD 转换电路如图 3-6 所示。

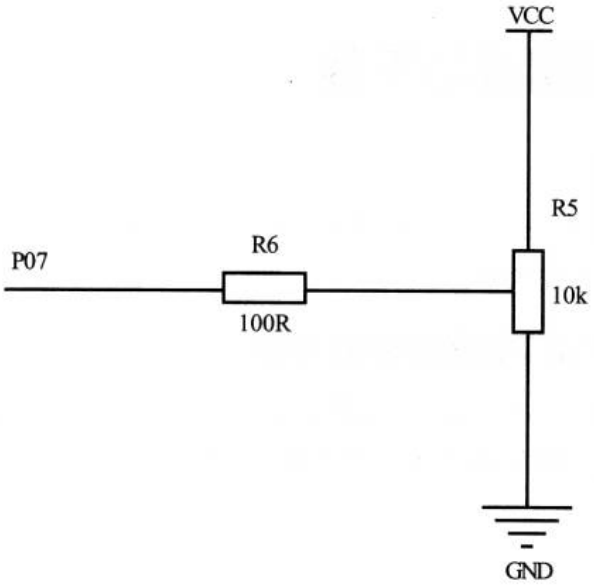


图 3-6　AD 转换电路

3.1.7 串口电路设计

串口电路主要用于实现CMOS/TTL电平到RS232电平的转换，串口电路如图3-7所示。

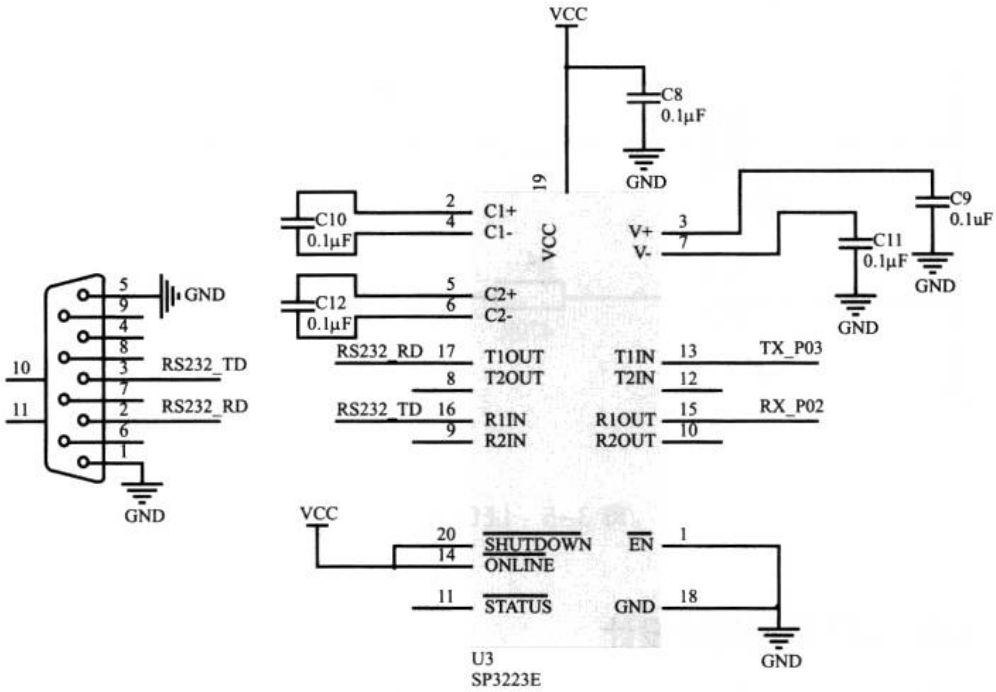


图3-7 串口电路

3.2 仿真调试与下载

源程序编译完成后，就需要进行源程序的仿真调试和程序下载，在此之前需要安装相应的仿真下载器驱动程序。

3.2.1 仿真调试下载器驱动的安装

将Usb Debug Adapter仿真下载器[●]通过USB电缆连接到PC机，在Windows XP系统下，系统会自动检测到新硬件，弹出新硬件更新向导对话框如图3-8所示。

● 智造者科技有限公司推出的Usb Debug Adapter仿真器与TI公司CC Debugger完全兼容，可与IAR EW8051软件配合实现对CC2540的固件调试，可与TI SmartRF Studio 7配合实现对CC2540的控制，可与SmartRF Flash Programmer配合实现对CC2540的编程。

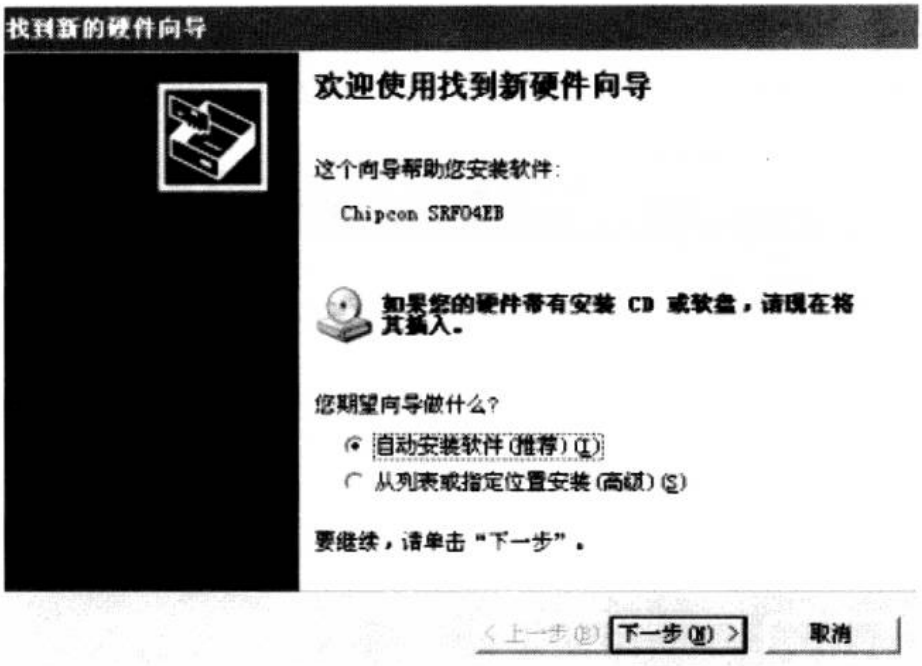


图 3-8　新硬件更新向导

选择“自动安装软件（推荐）”选项，然后单击下一步，该向导会自动搜索驱动程序并进行安装，安装驱动过程如图 3-9 所示。

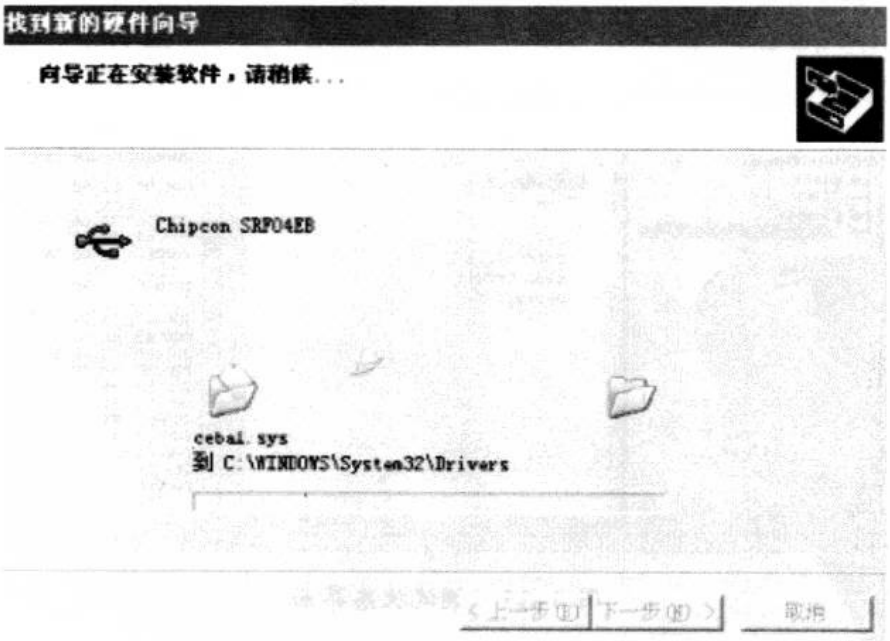


图 3-9　安装驱动过程

驱动程序安装完成后，即可进行仿真调试和程序下载等功能了。

3.2.2 程序仿真调试

连接仿真器和开发板，并接通电源，单击 Debug 按钮，如图 3-10 所示。

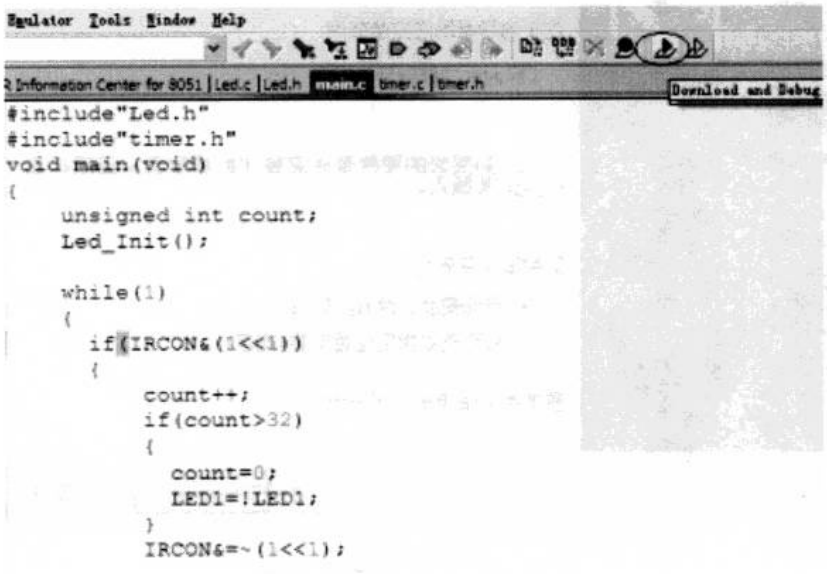


图 3-10 单击 Debug 按钮

此时，会出现调试状态界面如图 3-11 所示。

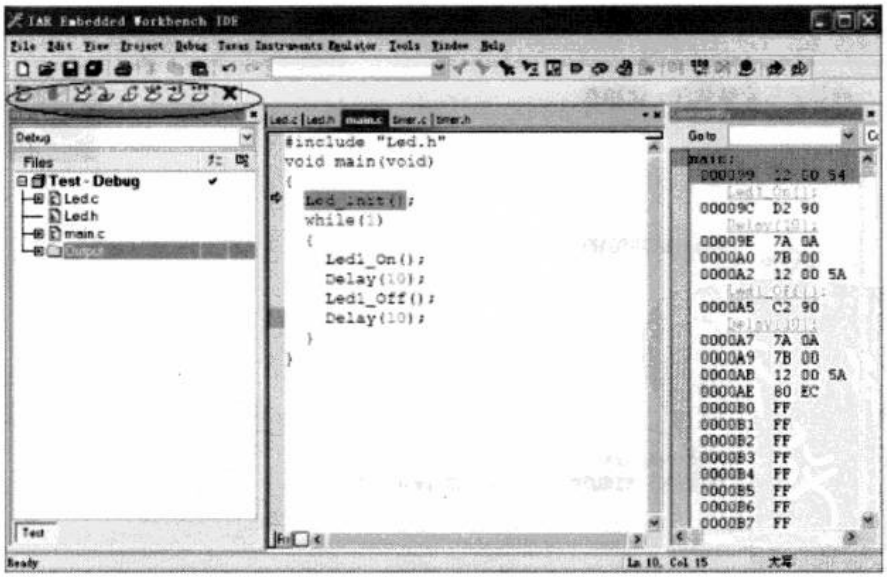


图 3-11 调试状态界面

其中，绿色的小箭头指示了当前程序的运行位置，使用左上方的调试功能可对程序进行调试。如果想退出调试状态，则只需要单击Stop Debugging按钮，如图3-12所示。

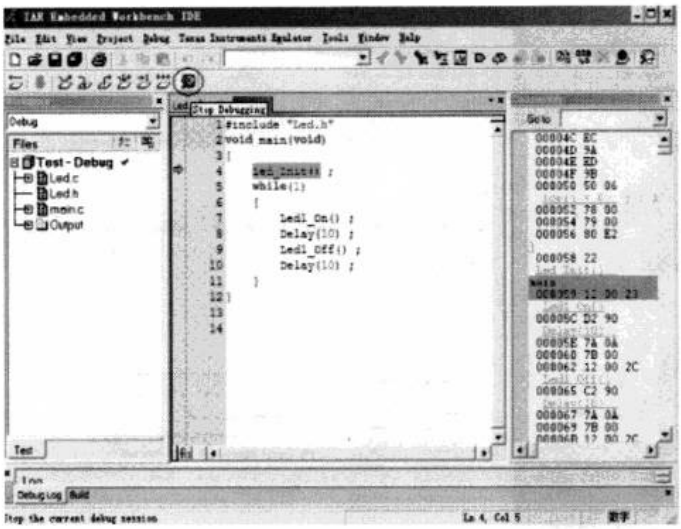


图 3-12　单击 Stop Debugging 按钮

3.2.3　结合 SmartRF Flash Programmer 软件实现程序下载

（1）Linker 选项

Output 标签下的选项主要用于设置输出文件名及格式，将 Output file 标签下面的文本框中输入“Test.hex”。要生成供 SmartRF Flash Programmer 下载的“.hex”文件，需要在 Format 选项选择“Other”项。Output 标签的设置如图 3-13 所示。

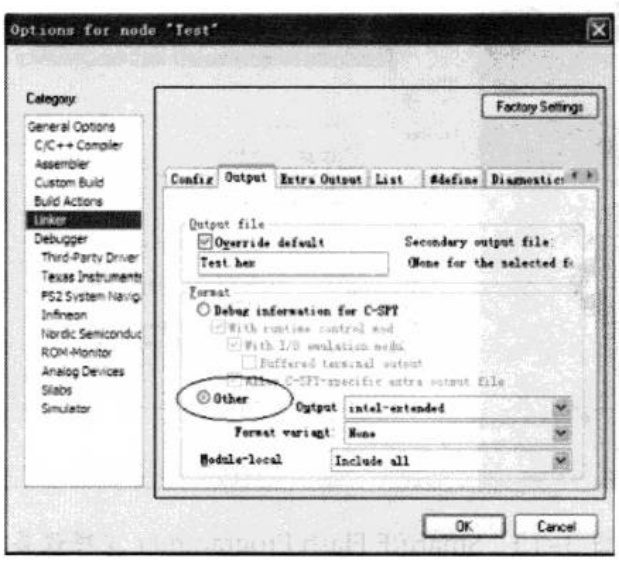


图 3-13　Output 标签的设置

（2）启动 SmartRF Flash Programmer 软件，出现如图 3-14 所示界面。

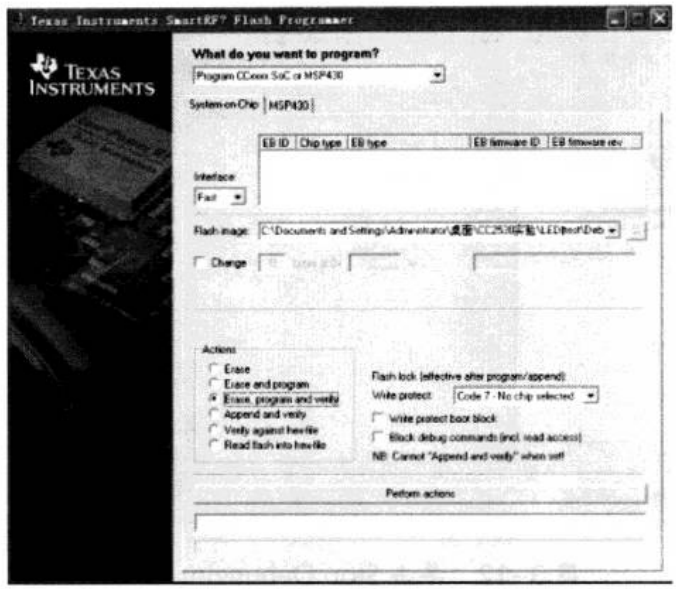


图 3-14　SmartRF Flash Programmer 启动界面

（3）连接 Usb Debug Adapter 仿真下载器、蓝牙开发板，并接通电源，按下 Usb Debug Adapter 仿真下载器复位键后，出现如图 3-15 所示界面。

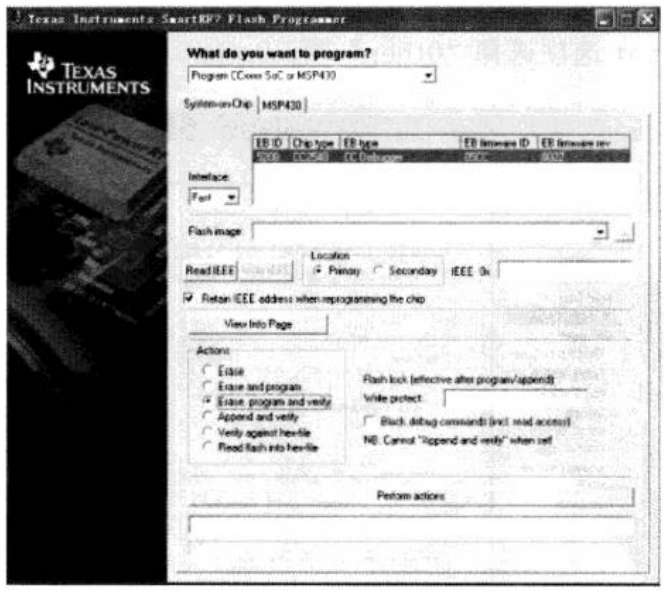


图 3-15　SmartRF Flash Programmer 发现设备

（4）选择要下载的“.hex”文件的路径，然后单击“Perform actions”，即可实现对 CC2540 的编程，如图 3-16 所示。

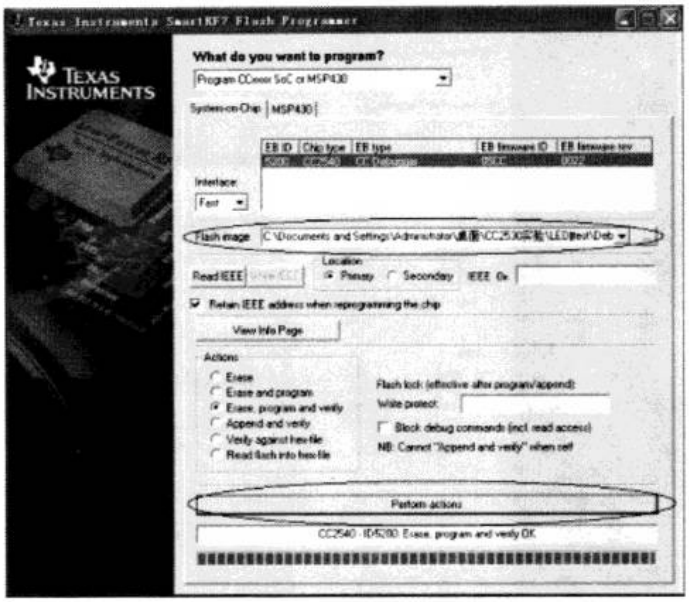


图 3-16　实现对 CC2540 的编程

3.2.4　读取 CC2540 蓝牙设备的 IEEE 地址

在 SmartRF Flash Programmer 界面选择“Read IEEE”按钮，如图 3-17 所示。

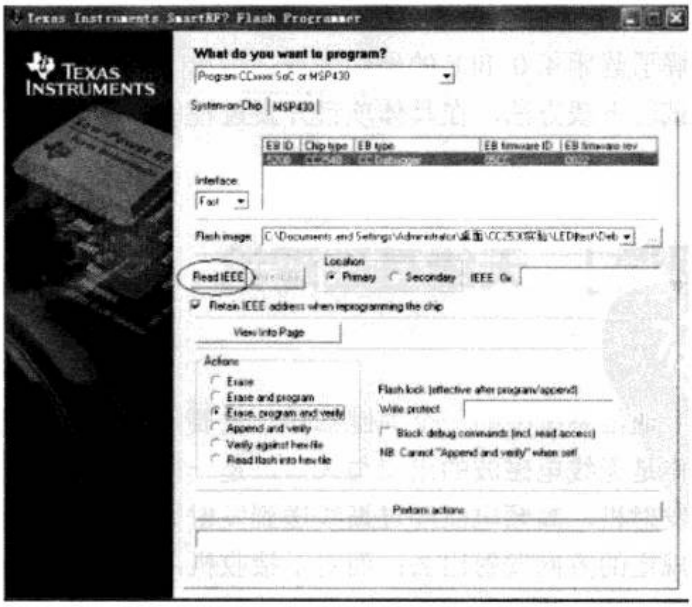


图 3-17　SmartRF Flash Programmer 读取设备 IEEE 地址

此时，在右侧的文本框中显示了 CC2540 的 IEEE 地址（48bit，以 16 进制表示），如图 3-18 所示。

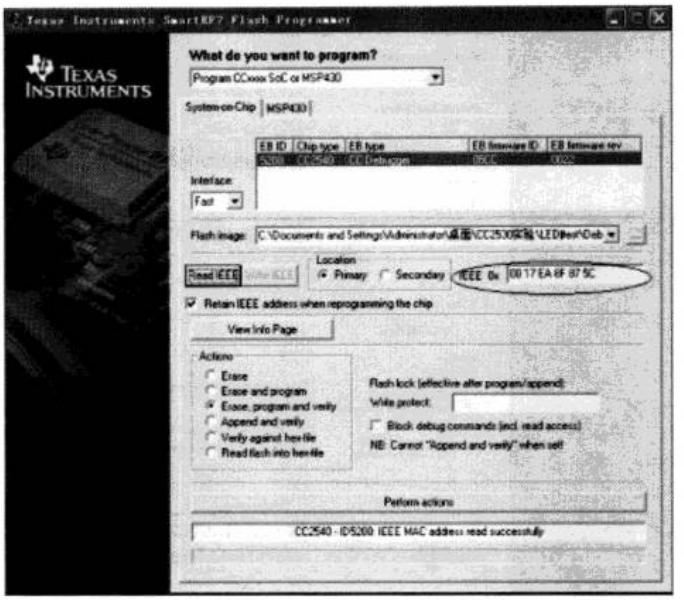


图 3-18　显示设备 IEEE 地址

3.3 本章小结

本章主要讲解了蓝牙 4.0 BLE 的硬件平台，给出了具体的硬件电路设计方法以及程序的仿真调试、下载方法，在具体项目开发过程中，需要结合系统需求对硬件平台进行设计。

[扩展阅读] 天线基础理论

天线作为无线通信系统中的一个关键部件，主要是用来辐射或接收电磁波，因此可以将天线看作是无线电磁波的出口与入口，是一种导行波与自由空间波之间的转换器件。对于发射机，高频电流经过馈线送到发射天线，发射天线将高频电流变换成电磁波，向规定的方向发射出去；而对于接收机，则是将来自一定方向的电磁波转换为高频电流，通过馈线送入接收机的输入回路。

1. 天线的一些基本参数

为了表征天线的电特性，需要使用方向图、输入阻抗、驻波系数、增益、带宽和极化等特性参量对天线进行表述。

下面对各个特性参量进行简单解释，以便读者对于天线理论有一个基本的认识。

（1）辐射方向图

天线辐射的功率在有些方向上大，有些方向上小，而这种表示辐射功率大小在空间的分布图，就称之为天线的辐射方向图。在工程上一般采用两个相互正交的主平面上的方向图来表示天线的方向性，通常称为E面和H面（E面指的是通过天线最大辐射方向并平行于电场矢量的平面，H面是通过天线最大辐射方向并垂直于E面的平面）。不同形式的天线，其方向图也不相同；对于同样的一副天线，其E面和H面方向图也不相同。如对于半波天线，其E面方向图为八字形，而H面方向图为圆形。

（2）输入阻抗

天线的输入阻抗指的是天线输入端电压与输入端电流的比值，输入阻抗的大小表征了天线与发射机或接收机的匹配状况。

（3）驻波系数

主要用来表征天线与馈线匹配状况，通过它的大小可以计算从天线反射回发射机或接收机的功率多少。

（4）增益

在相同的输入功率下，天线在某方向某点产生的场强平方与点源天线在同方向同一点产生场强平方的比值，表征了天线集中辐射的程度。

（5）带宽

电性能下降到容许值的频率范围称为天线的带宽，因此有驻波带宽、方向图带宽、圆极化轴比带宽等，一般情况下带宽指驻波带宽。

（6）极化

用来描述天线辐射电磁波矢量空间指向的参数。接收和发射天线的极化不匹配将会影响接收效果。

2. 常见的天线形式

对于天线分类，有各种不同的方法。如按工作性质可以分为发射天线和接收天线；按用途可以分为通信天线、广播天线、雷达天线等。在此按照天线原理来分类，主要包括线天线、口径天线和阵列天线。

▲ 线天线：天线的长度远大于其横截面，如半波天线，单极子天线等；

▲ 口径天线：由整块金属板或导线栅格组成阵面，如喇叭天线，反射面天

线等；

▲ 阵列天线：为了提高天线的增益，将天线单元按照一定的规律进行布阵，同时通过控制各个阵元的幅度和相位来控制整个阵列的电性能。如微带天线等。

几种常见的天线形式如图 3-19 所示。

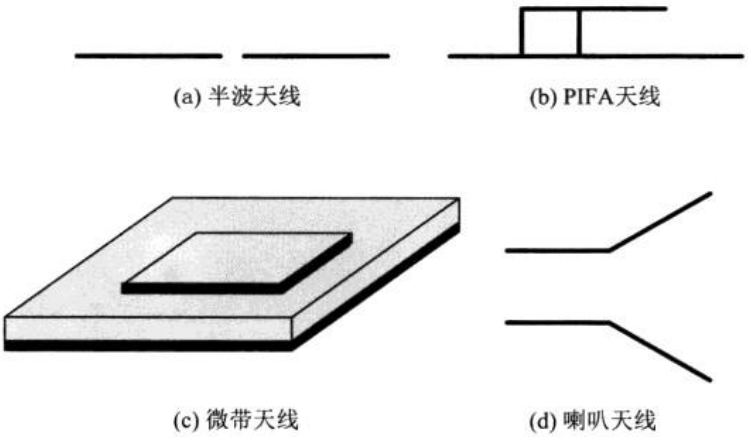


图 3-19 几种常见的天线形式

3. 蓝牙 4.0 BLE 模块天线选型

对于蓝牙 4.0 BLE 模块，要求天线的辐射方向图为全向，因此，一般使用偶极子、单极子和倒 F 天线等。在设计过程中可以将天线单独设计，也可以将天线和整个模块整体设计，这取决于具体的应用。因此需要按照具体的情况来选择天线，一般主要从性能、尺寸和成本三方面来考虑。有两种可以用于蓝牙 4.0 BLE 模块的天线有倒 F 天线和折叠偶极子天线。

第 4 章

CC2540 基础实验

第2章讲述了使用IAR集成开发环境进行应用程序开发的基本流程，包括工程的建立、源文件的添加、编译与调试。本章主要讲述CC2540单片机I/O口、定时器、串口等硬件资源的使用。

4.1 流水灯基础实验

本实验目的：让读者学会使用CC2540射频单片机的I/O口来控制外设，本实验用LED灯作为外设。按照第2章中讲述的方法建立工程，并添加Led.c、Led.h、main.c源文件。如图4-1所示。

图4-1 添加源文件到工程

向“Led.h”文件中输入以下代码：

```
#ifndef __LED_H__
#define __LED_H__
#include<ioCC2540.h>        // 该文件包含CC2540一些寄存器宏定义
#define LED1 P1_0
#define LED2 P1_1
#define LED3 P1_4
#define Led1_On()    LED1=1;
#define Led1_Off()    LED1=0;
#define Led2_On()    LED2=1;
#define Led2_Off()    LED2=0;
```

```
#define Led3_On()    LED3=1;
#define Led3_Off()    LED3=0;
extern void Led_Init(void);
extern void Delay(unsigned int time);
#endif
```

程序说明：

▲ 加粗部分代码是为了防止文件的重复包含问题，读者只需要记住这种格式即可，在模块化程序开发过程中，会经常使用该技巧；

▲ 在程序的最后，使用 extern 关键字声明了 2 个外部函数，这两个函数的实现放在“Led.c”文件中。

“Led.c”文件内容如下：

```
#include "Led.h"
void Led_Init()
{
1  P1SEL&=~((1<<0)|(1<<1)|(1<<4));
2  P1DIR|=(1<<0)|(1<<1)|(1<<4);
3  Led1_Off();
4  Led2_Off();
5  Led3_Off();
}
6 void Delay(unsigned int time)
{
7  unsigned int i;
8  while(time--)
  {
9   for(i=0;i<10000;i++);
  }
}
```

第 1 行，将 p1_0、p1_1、p1_4 设置为通用 I/O 口。

第 2 行，将 p1_0、p1_1、p1_4 设置为输出。

第 3 ～ 5 行，使 LED 处于熄灭状态。

第 6 ～ 9 行，延时函数，参数为 time。

程序说明：

在 Led_Init() 函数中用到了寄存器 P1SEL 和 P1DIR，用于配置 I/O 口功能，关于这两个寄存器的详细使用方法请读者参考 CC2540 单片机数据手册。

“main.c”文件内容如下：

```
#include"Led.h"
```

```
void main(void)
{
  Led_Init();
  while(1)
  {
    Led1_On() ;
    Delay(10) ;
    Led1_Off() ;
    Delay(10) ;
  }
}
```

程序说明：

▲ 该程序中，要使用到LED初始化函数，而该函数又是在“Led.c”文件中实现的，在“Led.h”文件中使用extern关键字对其进行了声明，那么，在“main.c”文件中需要使用该函数时，只需要包含“Led.h”文件即可，即#include“Led.h”；

▲ 该工程文件实现的功能为在主循环中，点亮LED1，延时一段时间，然后熄灭LED1，延时一段时间后，再次点亮LED1，一直循环执行。

单击工具栏上make图标，完成工程文件的编译，如图4-2所示。

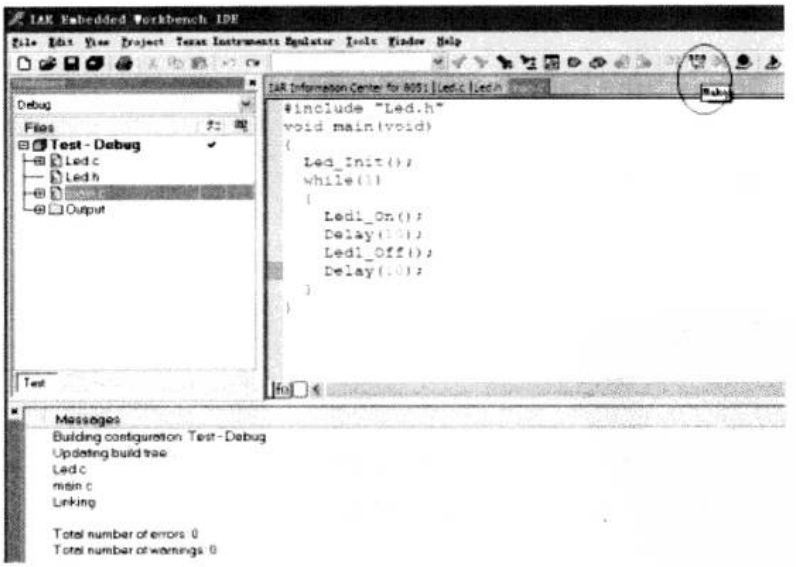


图4-2 编译工程文件

在该工程所在的文件夹下有个Debug文件夹，打开该文件夹，有个Exe文件，打开该文件夹，会看到生成的“Test.hex”文件，然后按照第3章讲述的方法将编译后生成的“.hex”文件烧写到硬件中，就可以观察到LED灯不断闪烁。这里只是讲解了点亮LED1的方法，点亮其他LED的方法与此类似，在此不再赘述。

4.2 定时器 / 计数器基础实验

定时器 T1 是一个支持典型定时器 / 计数器功能（例如输入捕获、输出比较和 PWM 功能）的独立 16 位定时器。它有 5 个独立的捕获 / 比较通道。每个通道使用一个 I/O 引脚。该定时器用于范围广泛的控制和测量应用，5 个通道具备正计数 / 倒计数模式，将允许例如电机控制应用的实现。

定时器 T1 的特征如下：

- ▲ 5 个捕获 / 比较通道；
- ▲ 上升沿、下降沿或任何边沿输入捕获；
- ▲ 设置、清除或切换输出比较；
- ▲ 自由运行、模或正计数 / 倒计数操作；
- ▲ 1、8、32 或 128 时钟分频；
- ▲ 在每个捕获 / 比较和最终计数上产生中断请求。

本实验目的：让读者初步学会使用 T1 定时器。涉及到的寄存器主要有：P1，P1DIR，P1SEL，T1CTL，T1STAT，IRCON，具体寄存器各个位的含义定义如表 4-1 所示。

表 4-1　定时器 T1 寄存器各位含义

T1CTL（0xE4）	Timer1 控制寄存器
	Bit3 ~ Bit2：定时器时钟分频倍数选择 00：不分频　01：8 分频　10：32 分频　11：128 分频
	Bit1 ~ Bit0：定时器模式选择 00：暂停 01：自动重装 0×0000-0xFFFF 10：比较计数 0×0000-T1CC0 11：PWM 方式
T1STAT（0xAF）	**Timer1 状态寄存器**
	Bit5：OVFIF 定时器溢出中断标志，在计数器达到计数终值时置位 1
	Bit4：定时器 T1 通道 4 中断标志位
	Bit3：定时器 T1 通道 3 中断标志位
	Bit2：定时器 T1 通道 2 中断标志位
	Bit1：定时器 T1 通道 1 中断标志位
	Bit0：定时器 T1 通道 0 中断标志位
IRCON（0xC0）	**中断标志位寄存器**

控制寄存器T1CTL通常用来控制定时器操作。状态寄存器T1STAT保持中断标志。

注意 ▶▶▶

读者在初学时不需要过多地关注众多的寄存器，只需关注用到的几个寄存器即可，等基础实验学完后，相信读者对CC2540单片机的一些常用寄存器和基本I/O控制会很熟悉。

下面讲解使用定时器T1来实现如下功能：使LED1每秒钟闪烁一次。

分析：要想使LED1每秒钟闪烁一次，只需要定时0.5秒，定时时间到来时，点亮LED1，然后再定时0.5秒，定时时间到来时，熄灭LED1，依此循环……

向Test工程中添加两个源文件“Timer.h”和“Timer.c”，如图4-3所示。

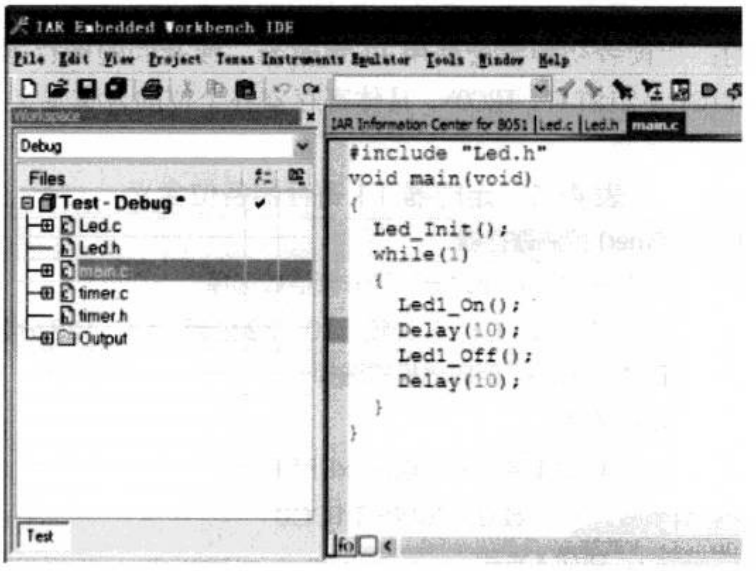


图4-3 添加源文件

然后，向“Timer.c”文件中输入以下内容：

```
# ifndef __TIMER_H__
#define  __TIMER_H__
extern void Timer1_Init(void);
#endif
```

向“Timer.c”文件中输入以下内容：

```
#include<ioCC2540.h>
#include"timer.h"
```

```
void Timer1_Init()
{
  T1CTL&=~((3<<0)|(3<<2));
  T1CTL|=((1<<0)|(1<<2));        // 自动重装，8 分频
}
```

程序说明：

▲ Timer_Init() 函数初始化了定时器 T1，因为系统时钟是 32MHz，将定时器输入时钟设置为系统时钟的 8 分频，即 4MHz，然后设置定时器工作模式为自由运行模式；

▲ 关于自由运行模式有如下知识点：定时器 T1 在自由运行操作模式，计数器从 0x0000 开始，每一个有效时钟边沿增加 1。当计数器到达 0xFFFF（溢出）时，计数器自动加载计数初值 0x0000，然后继续进行递增计数。当达到最终计数值 0xFFFF 时，“IRCON.T1IF”标志会置位。自由运行模式用于产生独立的时间间隔和输出信号频率。因此，可以计算最大定时时间如下。

系统时钟频率为：32MHz；

8 分频后作为定时器输入时钟：4MHz；

因此，定时器每次递增计数的时间间隔为：1/4M 秒；

总的定时时间

= 计数总次数 × 定时器每次递增计数的时间间隔

= (0xFFFF+1) ×1/4M 秒

= $2^{16}/(4\times2^{20})$

= $1/2^{6}$)

= 1/64

因此，自由运行模式下，8 分频的最大定时时间为 1/64 秒，定时 0.5 秒只需要定时器工作 32 次即可。

这里需要注意一下数据之间的计算方法。

▲ 0xFFFF+1=0x10000，转换为十进制即 2^{16}；

▲ 1M = 2^{20}。

修改“main.c”文件内容如下：

```
#include"Led.h"
#include"timer.h"
void main(void)
{
  unsigned int count;
  Led_Init();
  while(1)
  {
```

```
        if(IRCON&(1<<1))
        {
          count++;
          if(count>32)
          {
            count=0;
            LED1=!LED1;
          }
        IRCON&=~(1<<1);
        }
      }
    }
```

程序说明：

因为在“main.c”文件中，会使用到LED初始化函数和定时器T1初始化函数，所以要包含相应的头文件。当定时器溢出时，IRCON寄存器中的T1IF位会置1，IRCON寄存器中各位含义如表4-2所示。

表4-2 IRCON中断标志位寄存器

位	名称	复位	R/W	描述
7	STIF	0	R/W	睡眠定时器中断标志 0：中断未挂起 1：中断挂起
6	—	0	R/W	必须写为0，写为1总是使能中断源
5	P0IF	0	R/W	端口0中断标志 0：中断未挂起 1：中断挂起
4	T4IF	0	R/W H0	定时器4中断标志：当定时器4中断发生时置为1，当CPU指向中断服务程序时清0 0：中断未挂起 1：中断挂起
3	T3IF	0	R/W H0	定时器3中断标志：当定时器3中断发生时置为1，当CPU指向中断服务程序时清0 0：中断未挂起 1：中断挂起

续表

位	名称	复位	R/W	描述
2	T2IF	0	R/W H0	定时器 2 中断标志：当定时器 2 中断发生时置为 1，当 CPU 指向中断服务程序时清 0 0：中断未挂起 1：中断挂起
1	T1IF	0	R/W H0	定时器 1 中断标志：当定时器 1 中断发生时置为 1，当 CPU 指向中断服务程序时清 0 0：中断未挂起 1：中断挂起
0	DMAIF	0	R/W	DMA 完成中断标志 0：中断未挂起 1：中断挂起

因此，只需要检测 IRCON 寄存器的第 1 位是否为 1，即可知道定时器 T1 是否产生中断标志，检测方法为：IRCON & (1 << 1)；定时器中断发生后，需要清除中断标志位，方法为：IRCON &= ～(1 << 1) ；

注意 ▶▶▶

请读者一定要掌握这种检测寄存器某位是否为 1、清 0 寄存器中的某一位的方法，这在嵌入式系统设计中尤其重要。

最后，单击 Make 按钮，将生成的“.hex”格式的文件下载到开发板，即可实现开发板上的 LED1 每秒闪烁一次。

这证明，上述分析是正确的，读者可以根据上述分析方法，根据不同的定时时间来计算相应的参数。当然，这里只是讲解了定时器 T1 的自由运行模式，读者可以结合 CC2540 数据手册练习其他模式的使用方法，在此不再赘述。

4.3 定时器 / 计数器高级实验——定时器中断

前一节讲解了定时器 T1 的使用方法，使用的是查询法来确定定时器 T1 何时溢出，因为当定时器 T1 溢出时，IRCON 寄存器的 T1IF 位会置 1，通过在程序中查询这一位是否置 1，来判断定时器 1 是否溢出。

但是，这种方法需要不断地查询该中断标志位的值，比较消耗CPU时间，一般使用中断的方式来处理定时器问题，具体思路如下：

▲ 在定时器T1初始化函数中，设定定时时间、打开定时器中断；

▲ 单片机正常执行程序，当定时时间到来时，调用相应的中断处理程序来处理即可。

下面结合具体实例进行讲解。

修改“Timer.c”文件中的Timer1_Init()函数，修改后的内容如下：

```
void Timer1_Init()
{
  1  T1CTL&=~((3<<0)|(3<<2));
  2  T1CTL|=((1<<0)|(1<<2));
  3  IEN1|=1<<1;
  4  IEN0|=1<<7;
}
```

第1～2行，将定时器设置为自动重装，8分频模式。

第3行，开启定时器T1中断。

第4行，开启系统总中断。

程序说明：

在定时器初始化函数中设定了定时器的运行模式，这里是自由运行模式，该模式的最大定时时间在前文已经进行了讲解，在此不作赘述。然后开启定时器T1中断，开启系统总中断。

注意 ▶▶▶

打开定时器T1中断，只需要将IEN1寄存器的第2位置1即可，一般将某位置1，采用按位或操作来实现，如：IEN1 |= 1 << 1。

“main.c”文件内容如下：

```
#include"Led.h"
#include"timer.h"
unsigned int count=0;
void main(void)
{
  Led_Init();
  Timer1_Init();
```

```
  while(1);
}
#pragma vector=T1_VECTOR
__interrupt void T1_ISR(void)
{
    count++;
    if(count>32)
    {
      count=0;
      LED1=!LED1;
    }
    IRCON&=~(1<<1);
}
```

这里需要注意在 IAR EW 软件环境下，声明中断处理函数的方法如下：

```
#pragma vector=T1_VECTOR
__interrupt void T1_ISR (void)
{
    ……        //在此添加自己的中断处理内容
}
```

其中，加粗部分是固定的格式，中断处理函数的名字可以随便取，如 T1_ISR，或者 timer_isr。

扩展：如果读者使用的是定时器 T3 的中断处理函数，只需要使用如下方式即可：

```
#pragma vector=T3_VECTOR
__interrupt void T3_ISR (void)
{
    ……        //在此添加自己的中断处理内容
}
```

读者可能会有如下疑问：上述格式中的 T1_VECTOR 和 T3_VECTOR 是什么呢，从哪里定义的呢？

在 IAR EW 开发环境中，如果想查看某个变量或者函数的定义，只需要在该变量或者函数上面右键单击，然后在弹出的下拉菜单中选择“go to definition of” 即可，如图 4-4 所示。

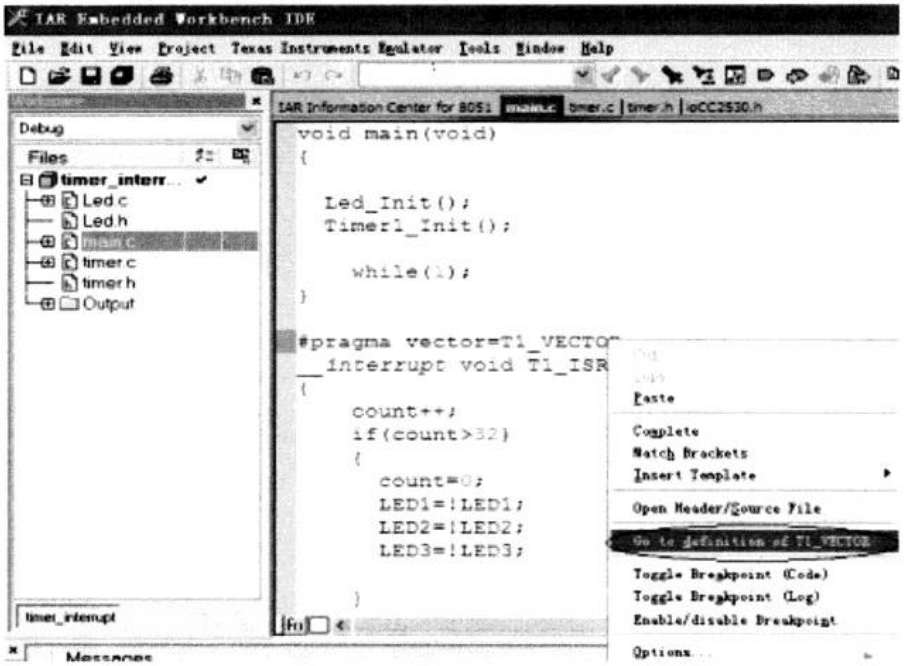


图 4-4　查看变量或者函数的定义

此时，会跳转到 T1_VECTOR 的定义处，如图 4-5 所示。

```
/*
*
*
*/
#define  RFERR_VECTOR    VECT(  0, 0x03 )
#define  ADC_VECTOR      VECT(  1, 0x0B )
#define  URX0_VECTOR     VECT(  2, 0x13 )
#define  URX1_VECTOR     VECT(  3, 0x1B )
#define  ENC_VECTOR      VECT(  4, 0x23 )
#define  ST_VECTOR       VECT(  5, 0x2B )
#define  P2INT_VECTOR    VECT(  6, 0x33 )
#define  UTX0_VECTOR     VECT(  7, 0x3B )
#define  DMA_VECTOR      VECT(  8, 0x43 )
#define  T1_VECTOR       VECT(  9, 0x4B )
#define  T2_VECTOR       VECT( 10, 0x53 )
#define  T3_VECTOR       VECT( 11, 0x5B )
#define  T4_VECTOR       VECT( 12, 0x63 )
#define  P0INT_VECTOR    VECT( 13, 0x6B )
#define  UTX1_VECTOR     VECT( 14, 0x73 )
#define  P1INT_VECTOR    VECT( 15, 0x7B )
```

图 4-5　CC2540 单片机的中断向量表

下面的问题是：如何再返回到“main.c”文件中？当然可以使用鼠标选择“main.c”文件，但是当文件较多时这样很不方便，在此介绍一种简便的方法：可以使用如下两个按钮来切换，如图 4-6 所示。

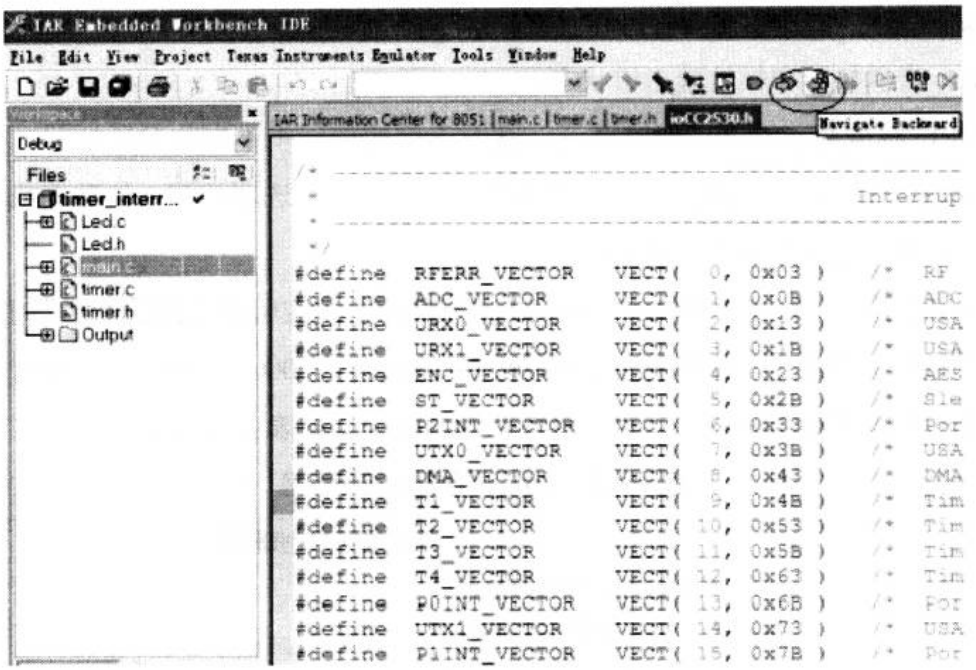


图 4-6 切换按钮

下面接着讲解例程，每当定时器 T1 溢出时，都会调用该中断处理函数，将 count 的值加 1，定时器溢出 32 次，所经过的时间是 0.5 秒，因此可以使用如下代码来实现 LED1 每隔 1 秒闪烁一次的功能：

```
if(count > 32)
  {
    count=0;
    LED1=!LED1;
  }
```

4.4 UART 串口基础实验

UART 模式提供异步串行接口。在 UART 模式中，接口使用 2 线或者含有 RXD、TXD、可选的 RTS 和 CTS 的 4 线。UART 模式的操作具有下列特点：

- ▲ 8 位或者 9 位负载数据；
- ▲ 奇校验、偶校验或者无奇偶校验；
- ▲ 配置起始位和停止位电平；
- ▲ 配置 LSB 或者 MSB 首先传送；
- ▲ 独立收发中断；

▲ 独立收发 DMA 触发；

▲ 奇偶校验和帧校验出错状态。

UART 模式提供全双工异步传送，接收器中的位同步不影响发送功能。传送一个 UART 字节包含 1 个起始位、8 个数据位、1 个作为可选项的第 9 位数据或者奇偶校验位、再加上 1 个或 2 个停止位。

本实验目的：让读者初步学会串口的使用方法。本实验涉及到的寄存器主要有：P1，P1DIR，P1SEL，CLKCONCMD，CLKCONSTA，U0CSR，U0GCR，U0BAUD（前面已介绍过的这里不再重复介绍）。几个主要寄存器的位含义如表 4-3 所示。

表 4-3 UART0 相关寄存器

	时钟频率控制寄存器
CLKCONCMD（0xC6）	Bit7：32k 时钟源选择 0：32k 晶振 1：32k RC 振荡
	Bit6：系统主时钟源选择 0：32k 晶振 1：16k RC 振荡
	Bit5 ~ Bit3：定时计数器时钟源选择 [2:0] 000:32M 001:16M 010:8M 011:4M 100:2M 101:1M 110:500k 111:250k
	Bit2 ~ Bit0：定时计数器时钟源选择 [2:0] 000:32M 001:16M 010:8M 011:4M 100:2M 101:1M 110:500k 111:250k
CLKCONSTA（0×9E）	时钟频率状态寄存器（只读）
U0CSR（0×S6）	UART0 状态和控制寄存器 Bit7：串口模式选择 0：UART 1：SPI Bit6：UART 接收使能 0：关 1：开
U0GCR（0×C5）	UART0 通用控制寄存器 Bit4 ~ Bit0：BAUD-E 波特率设置
U0BAUD（0×C2）	UART0 波特率控制寄存器
UTX0IF（0×EF）	UART0 TX 中断标志位

读者如果对单片机不熟悉，不容易掌握上述寄存器的使用方法，下面将结合具体的例程进行讲解，读者可以暂时不理会具体寄存器的功能，先按照本书的寄存器初始化方法使用，当观察到串口输出字符后，再结合实验效果来理解各寄存器的使用方法。

按照第 2 章讲解的方法，建立一个 uart 工程，向 uart 工程中添加三个文件“Uart.h”、“Uart.c”、“main.c”，工程文件布局如图 4-7 所示。

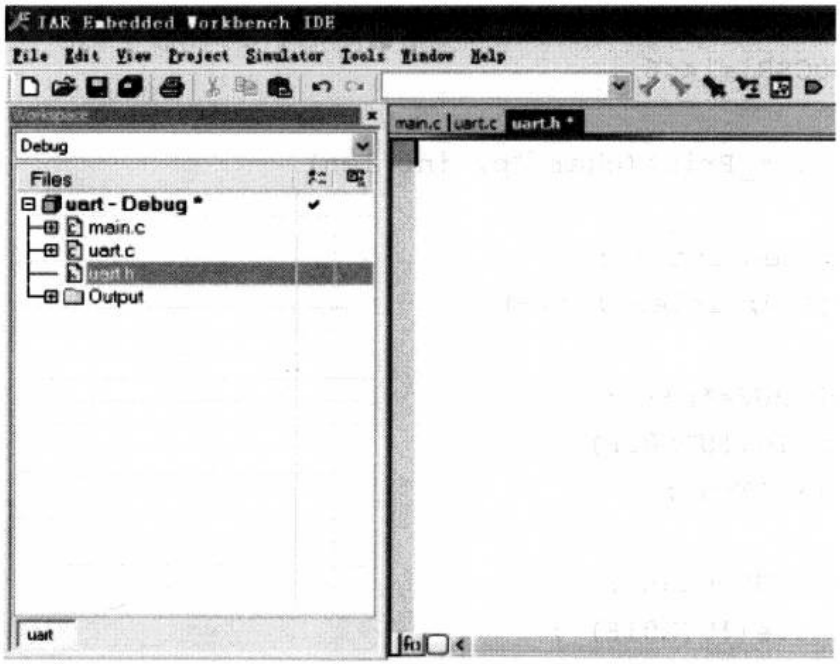


图 4-7　工程文件布局

在“Uart.h”文件中输入以下内容：

```
#ifndef __UART_H__
#define __UART_H__
extern void Uart_Init(void) ;
extern void Uart_Print(char *p, int len) ;
#endif
```

“Uart.h”文件内容如下：

```
#include<ioCC2540.h>
#include "uart.h"
void Uart_Init(void)
{
1  CLKCONCMD&=~(1<<6) ;
2  while(CLKCONSTA&(1<<6)) ;
3  CLKCONCMD&=~((1<<6)|(7<<0));
4  PERCFG=0x00;
5  P0SEL|=(0xf<<2);
6  P2DIR&=~(3<<6);
7  U0CSR|=1<<7;
8  U0GCR=9 ;
9  U0BAUD=59 ;
```

```
10  UTX0IF=0 ;
11  U0CSR|=1<<6 ;
}
void Uart_Print(char *p, int len)
  {
  unsigned int i ;
  for(i=0; i<len ; i++)
  {
     U0DBUF=*p++ ;
12  while(!UTX0IF) ;
13  UTX0IF=0 ;
  }
14  U0DBUF=0x0A ;
15  while(!UTX0IF) ;
16  UTX0IF=0 ;
}
```

第 1 ～ 3 行，设置系统时钟，选择 32M 晶振。

第 4 行，用于选择串口管脚位置。

第 5 行，进行 I/O 口配置。

第 6 行，设置 UART0 的优先级最高。

第 7 行，选择串口 0。

第 8 ～ 9 行，波特率设为 19200。

第 10 行，将串口发送标志位清零。

第 11 行，允许接收数据。

第 12 行，等待数据发送完成。

第 13 行，将串口发送标志位清零。

第 14 行，“0x0A”为换行符的 ASCII 码。

第 15 ～ 16 行，等待发送完成，然后将串口发送标志位清零。

程序说明：

▲ 初始化串口波特率为 19200bps，8 个数据位，1 个停止位，无校验位。CLKCONCMD 寄存器主要用于设置系统时钟，读者只需要记住这种初始化方法即可，具体寄存器定义可以参考 CC2540 数据手册；

▲ 使用串口发送数据时，只需要将要发送的字符写入寄存器 U0DBUF 中即可自动发送，数据发送完成后，UTX0IF 置 1，因此，可以检测该位是否置 1 来确定数据是否发送完毕。

“main.c”文件内容如下：

```
#include"uart.h"
```

```
void main(void)
{
  Uart_Init() ;
  while(1)
  {
    Uart_Print("uart_test", 9) ;
  }
}
```

然后编译，将程序下载到开发板上，将开发板串口与电脑连接，打开串口调试助手，波特率设为19200，此时可观察到CC2540单片机向串口输出了字符串“uart_test”，如图4-8所示。

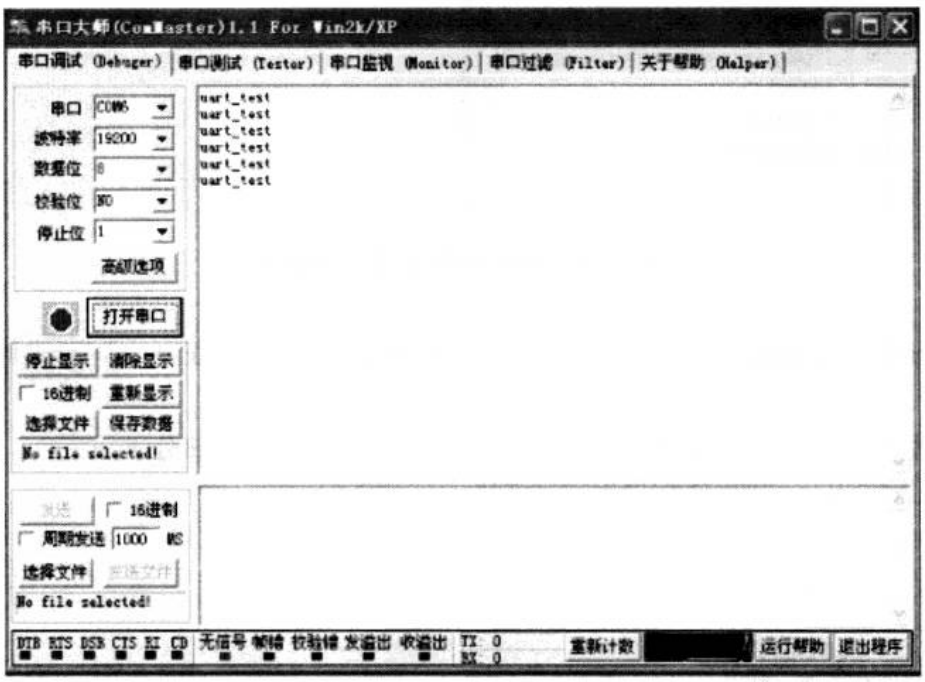


图 4-8　向串口输出数据

4.5 UART 串口高级实验

下面在串口实验的基础上，扩展部分功能，练习一下串口中断的使用方法。

实验功能：通过串口调试助手发送命令来控制开发板上LED1的亮灭。结合前面讲解定时器中断的思路，下面分析一下实现串口中断的基本步骤。

第一：初始化串口、打开串口接收中断；

第二：接收到PC机发送的命令后，执行相应的操作。

按照第2章讲解的方法，建立一个uart_interrupt工程，向uart_interrupt工程中添加五个源文件“Uart.h”、“Uart.c”、“Led.h”、“Led.c”、“main.c”，工程文件布局如图4-9所示。

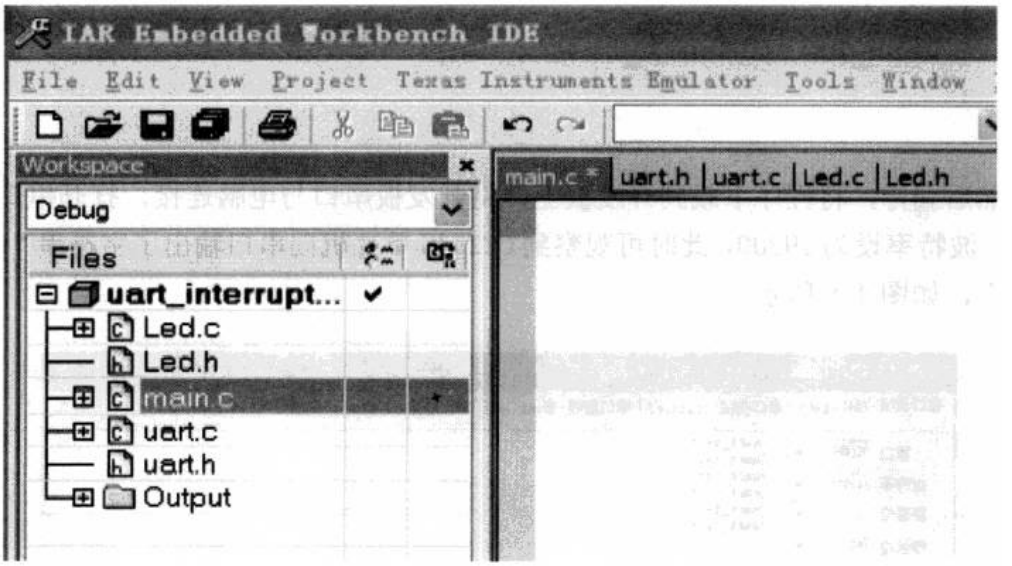


图4-9 串口中断工程文件布局

假设当开发板收到A时，点亮LED1，收到B时，熄灭LED1。下面讲解基本实现方法。

修改“Uart.c”文件内容如下（程序添加两行，见加粗部分）：

```
#include<ioCC2540.h>
#include "uart.h"
void Uart_Init(void)
{
1  CLKCONCMD&=~(1<<6) ;
2  while(CLKCONSTA&(1<<6)) ;
3  CLKCONCMD&=~((1<<6)|(7<<0));
4  PERCFG=0x00;
5  P0SEL|=(0xf<<2);
6  P2DIR&=~(3<<6);
7  U0CSR|=1<<7;
8  U0GCR=9 ;
9  U0BAUD=59 ;
10  UTX0IF=0 ;
11  U0CSR|=1<<6 ;
```

```
12  URX0IE = 1 ;
13  EA = 1 ;
}
void Uart_Print(char *p, int len)
{
  unsigned int i ;
  for(i=0; i<len ; i++)
  {
    U0DBUF=*p++ ;
    while(!UTX0IF) ;
    UTX0IF=0 ;
  }
  U0DBUF=0x0A ;
  while(!UTX0IF) ;
  UTX0IF=0 ;
}
```

第 1 ～ 3 行，设置系统时钟，选择 32M 晶振。

第 4 行，用于选择串口管脚位置。

第 5 行，进行 I/O 口配置。

第 6 行，设置 UART0 的优先级最高。

第 7 行，选择串口 0。

第 8 ～ 9 行，波特率设为 19200。

第 10 行，将串口发送标志位清零。

第 11 行，允许接收数据。

第 12 行，打开串口接收中断。

第 13 行，打开系统总中断。

修改“main.c”文件内容如下：

```
#include"Led.h"
#include"uart.h"
unsigned char cmd ;
void main(void)
{
  Led_Init();
  Uart_Init();
  while(1)
  {
    switch(cmd)
```

```
      {
      case 'A':
        cmd=0;
        Uart_Print("LED1_ON", 7);
        Led1_On();
        break;
      case 'B':
        cmd=0;
        Uart_Print("LED1_OFF", 8);
        Led1_Off();
        break;
      default:
        break;
      }
    }
  }
  #pragma vector=URX0_VECTOR
  __interrupt void URX0_ISR(void)
  {
    cmd=U0DBUF;
    URX0IF=0;
  }
```

程序说明：

▲ 接收到不同的命令使用不同的操作，可以使用多个 if 语句来实现，在工程实践中，更多的时候使用的是 switch-case 语句；

▲ 当串口接收到数据后，将其存放到变量 cmd 中，然后在主循环中对该变量进行判断，如果是大写字母 A，则点亮 LED1，如果是大写字母 B，则熄灭 LED1。

将程序编译并下载到开发板，打开串口调试助手，波特率设为 19200，发送字母 A，开发板上的 LED1 点亮，同时串口上输出了开发板发送过来的 LED1_ON 字符串。如图 4-10 所示。

图 4-10　发送字母 A

同理发送字母 B 时，LED1 熄灭，同时串口上输出了开发板发送过来的 LED1_OFF 字符串，如图 4-11 所示。

图 4-11　发送字母 B

一般情况下，开发板要对命令进行一定的判断，如果命令不正确，需要给读者提示，下面修改“main.c”文件，添加相应的错误处理机制。

```
#include"Led.h"
#include"uart.h"
unsigned char cmd ;
void main(void)
{
  Led_Init();
  Uart_Init();
  while(1)
  {
    switch(cmd)
    {
    case 'A':
      cmd=0;
      Uart_Print("LED1_ON", 7);
      Led1_On();
      break;
    case 'B':
      cmd=0;
      Uart_Print("LED1_OFF", 8);
      Led1_Off();
      break;
    case 0:
      Break;
    default:
      cmd=0;      //清为0不可少，不然一直打印程序下一行中的字符串
      Uart_Print("Please check cmd!", 17);
      break;
    }
  }
}
#pragma vector=URX0_VECTOR
__interrupt void URX0_ISR(void)
{
  cmd=U0DBUF;
  URX0IF=0;
}
```

将程序编译并下载到开发板，打开串口调试助手，波特率设为19200，发送字母C，开发板向串口输出了字符“Please check cmd!”可见实现了对输入命令进行判别的功能。执行效果如图4-12所示。

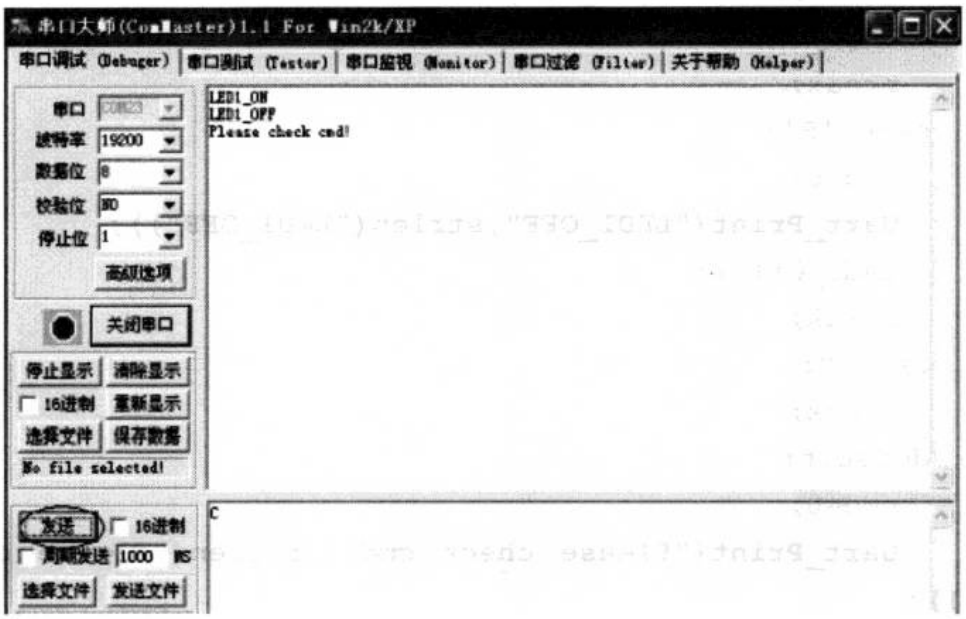


图4-12 发送字母C

现在的一个主要问题是Uart_Print(char *p, int len)第二个参数是要发送的字符串的长度，每次发送时都需要数一下字符串的长度，这样使用起来很不方便。下面讲解一种常用的处理方法：使用C库里面提供的函数int strlen(char *)，该函数会自动返回字符串的长度。当然，要想使用该函数，需要包含头文件<string.h>。

因此，需修改main.c文件如下（修改内容见加粗部分）：

```
#include"Led.h"
#include"uart.h"
#include"string.h"
unsigned char cmd;
void main(void)
{
  Led_Init();
  Uart_Init();
  while(1)
  {
    switch(cmd)
```

```
        {
        case 'A':
          cmd=0;
          Uart_Print("LED1_ON",strlen("LED1_ON"));
          Led1_On();
          break;
        case 'B':
          cmd=0;
          Uart_Print("LED1_OFF",strlen("LED1_OFF"));
          Led1_Off();
          break;
        case 0:
          break;
        default:
          cmd=0;
          Uart_Print("Please check cmd!",strlen("Please check
cmd!"));
          break;
        }
      }
    }
    #pragma vector=URX0_VECTOR
    __interrupt void URX0_ISR(void)
    {
      cmd=U0DBUF;
      URX0IF=0;
    }
```

4.6 UART串口编程高级技巧

在前面的实验中，读者可以发现，串口的波特率是固定的19200，有些时候需要支持多种串口，因此，想到在串口初始化时使用一个参数来传递串口的波特率信息。下面对此展开讲解。

初始化时可以设定串口的波特率，那么，在编写串口初始化函数时，需要对波特率进行判断，不同的波特率，会影响到下面两个寄存器的值：

▲ U0GCR = 11;

▲ U0BAUD = 216 ; // 波特率设为 115200。

这两个参数的初始化值可以参照表 4-4 来选择。

表 4-4 32MHz 系统时钟常用的波特率设置

波特率 /bps	UxBAUD.BAUD_M	UxGCR.BAUD_E	误差 /%
2400	59	6	0.14
4800	59	7	0.14
9600	59	8	0.14
14400	216	8	0.03
19200	59	9	0.14
28800	216	9	0.03
38400	59	10	0.14
57600	216	10	0.03
76800	59	11	0.14
115200	216	11	0.03
230400	216	12	0.03

对于不同的波特率，可以使用 switch-case 语句来实现选择。修改波特率初始化函数如下程序所示：

```
void Uart_Init(unsigned int baudrate)
{
CLKCONCMD&=~(1<<6) ;
while(CLKCONSTA&(1<<6)) ;
CLKCONCMD&=~((1<<6)|(7<<0));
PERCFG=0x00;
P0SEL|=(0xf<<2);
P2DIR&=~(3<<6);
U0CSR|=1<<7;
switch( baudrate)
{
  case 115200:
        U0GCR=11 ;
        U0BAUD=216 ;   // 波特率设为 115200
        break ;
  case 9600:
      U0GCR=8 ;
```

```
        U0BAUD=59;    // 波特率设为 9600
        break ;
    default:
        U0GCR=9 ;
        U0BAUD=59;   // 波特率设为 19200
        break ;
}
UTX0IF=0 ;
U0CSR|=1<<6 ;
URX0IE = 1 ;
EA = 1 ;
}
```

然后，修改 Uart.h 文件中 Uart_Init() 函数的声明如下：

```
void Uart_Init(unsigned int baudrate) ;
```

修改 main.c 文件如下：

```
#include"Led.h"
#include"uart.h"
#include"string.h"
unsigned char cmd;
void main(void)
{
  Led_Init();
  Uart_Init(9600);
  while(1)
  {
  switch(cmd)
  {
  case 'A':
      cmd=0;
      Uart_Print("LED1_ON",strlen("LED1_ON"));
      Led1_On();
      break;
  case 'B':
      cmd=0;
      Uart_Print("LED1_OFF",strlen("LED1_OFF"));
      Led1_Off();
      break;
```

```
    case 0:
        break;
    default:
        cmd=0;
        Uart_Print("Please check cmd!",strlen("Please check
cmd!"));
        break;
      }
    }
  }
  #pragma vector=URX0_VECTOR
  __interrupt void URX0_ISR(void)
  {
    cmd=U0DBUF;
    URX0IF=0;
  }
```

编译后将程序下载到开发板，此时的波特率已经可以修改了，当然这里只是添加了 2 个波特率，读者可以根据自己的实际需要，添加不同波特率的支持。

4.7 AD 转换实验之测量 VDD 实验

ADC 支持多达 14 位模数转换，有效位数（ENOB）12 位。ADC 包含一个具有 8 个独立配置通道的模拟多路转换器，一个参考电压发生器，并且通过 DMA 将转换结果写入存储器。具有多种运行模式。ADC 可以编程为从任何通道执行一个转换。这样一个转换通过写 ADCCON3 寄存器触发。

ADC 的主要特征如下：

- ▲ 可选的采样率，这也设置了分辨率（7 ～ 12 位）；
- ▲ 8 个独立的输入通道，单端或差分；
- ▲ 参考电压可选为内部、外部单端、外部差分或 AVDD5；
- ▲ 中断请求产生；
- ▲ 转换结束时 DMA 触发；
- ▲ 温度传感器输入；
- ▲ 电池测量能力。

主要用到的寄存器如表 4-5 所示。

表 4-5　ADC 寄存器

位	名称	复位	R/W	描述
7:6	EREF[1:0]	00	R/W	选择用于额外转换的参考电压 00：内部参考电压 01：AIN7 引脚上的外部参考电压 10：AVDD5 引脚 11：在 AIN6-AIN7 差分输入的外部参考电压
5:4	EDIV[1:0]	00	R/W	设置用于额外转换的抽取率。抽取率也决定了完成转换需要的时间和分辨率 00：64 抽取率（7 位 ENOB）　01：12S 抽取率（9 位 ENOB）　10：256 抽取率（10 位 ENOB）　11：512 抽取率（12 位 ENOB）
3:0	ECH[3:0]	0000	R/W	单个通道选择，选择写 ADCCON3 触发的单个转换所在的通道号码。当单个转换完成，该位自动清除 0000：AIN0　0001：AIN1　0010：AIN2　0011：AIN3 0100：AIN4　0101：AIN5　0110：AIN6　0111：AIN7 1000：AIN0-AIN1　1001：AIN2-AIN3　1010：AIN4-AIN5　1011：AIN6-AIN7　1100：GND　1101：正电压参考

本实验目的：让读者初步掌握使用 ADC 测量 CC2540 单片机工作电压 VDD。在前面串口实验的基础上，向 uart 工程中添加两个源文件“ADC.c”和“ADC.h”如图 4-13 所示。

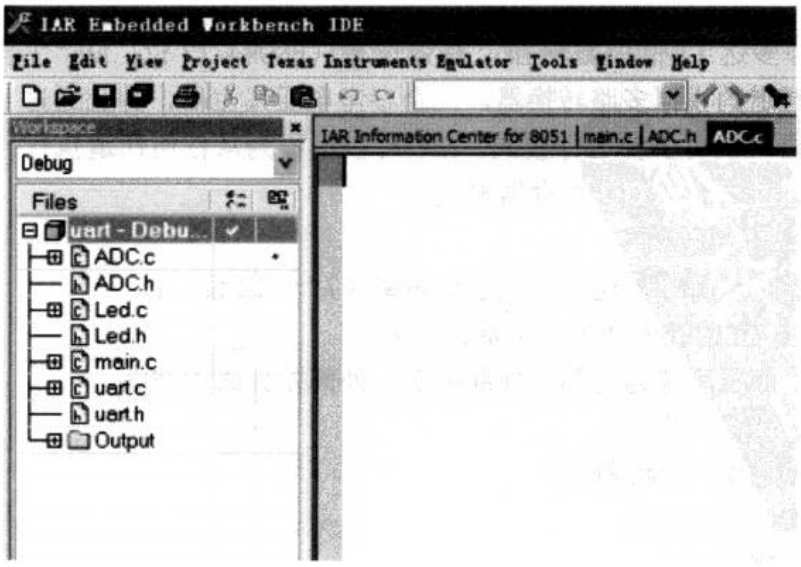


图 4-13　ADC 工程布局

“ADC.h”文件内容如下：

```
#ifndef __VDDCHECK_H__
#define __VDDCHECK_H__
extern unsigned int Read_Vddvalue(void);
#endif
```

“ADC.c”文件内容如下：

```
#include<ioCC2540.h>
#include"ADC.h"
1  #define HAL_ADC_REF_125V     0x00
2  #define HAL_ADC_DEC_064      0x00
3  #define HAL_ADC_CHN_VDD3   0x0f
unsigned int Read_Vddvalue(void)
{
   unsigned int value;
4  unsigned char tmpADCCON3=ADCCON3;
5  ADCIF=0;
   ADCCON3=(HAL_ADC_REF_125V|HAL_ADC_DEC_064|HAL_ADC_CHN_
VDD3);
6  while(!ADCIF);
7  value=ADCL>>2;
8  value|=((unsigned int)ADCH)<<6;
9  ADCCON3=tmpADCCON3;
10  return(value);
}
```

第 1 行，选择内部参考电压用于序列转换。

第 2 行，设置抽取率，7 位有效数字位。

第 3 行，选择 VDD/3 通道。

第 4 行，保存 ADCCON3 寄存器的值。

第 5 行，清除 ADC 中断标志位。

第 6 行，等待转换结束。

第 7 ～ 8 行，将采样到的 14 位 ADC 值赋值给 value。

第 9 行，恢复寄存器 ADCCON3 的值。

第 10 行，将 value 的值返回。

“main.c”文件内容如下：

```
#include"Led.h"
#include"uart.h"
#include"string.h"
```

```
#include"ADC.h"
unsigned char cmd;
void main(void)
{
  unsigned int vddvalue;
  unsigned char buf[5]="0.0V";
  Led_Init();
  Uart_Init();
  while(1)
  {
    vddvalue=Read_Vddvalue();
    vddvalue=(vddvalue*25)>>8;
    vddvalue=vddvalue*3;
    buf[0]=vddvalue/10+'0';
    buf[1]='.';
    buf[2]=vddvalue%10+'0';
    Uart_Print(buf, strlen(buf));
    Delay(100);
  }
}
```

在CC2540单片机中，VDD/3作为一个ADC输入通道，因此可以对该通道进行检测，得到VDD/3的值，然后将该值乘以3就得到了VDD的值。

将程序下载到开发板，打开串口调试助手，设置好波特率，执行效果如图4-14所示。

图4-14 采集到VDD的值

这里需要注意一下数据处理技巧：

vddvalue = ((vddvalue*25) >> 8) ;

参考电压是：1.25V；

AD 转换后的电压值为：（vddvalue*1.25/2^7），将其扩大 10 倍：（vddvalue*12.5/2^7）；

继续将小数部分转换为整数：（vddvalue*25/2^8）；

然后，将除法使用移位运算代替：vddvalue = ((vddvalue*25) >> 8) 。

因为使用 8 位单片机来计算乘法和除法比较慢，且消耗资源较多，在程序中常用的处理思路为：

▲ 尽量处理整数；

▲ 尽量将乘法用左移或右移来实现，除法用右移来实现。

注意 ▶▶▶

从上面的结果可以看出，测量的 VDD 不是很准确，这是由 ADC 的误差引起的，如果对 ADC 精度要求较高时，可以使用专门的 AD 芯片来完成。

4.8 AD 转换实验之温度采集实验

CC2540 内部集成了一个温度传感器，温度传感器的输出也可以作为 ADC 的输入，用于温度测量。在测量 VDD 实验的基础上，修改“ADC.h”、“ADC.c”文件内容。

修改“ADC.h”文件内容如下：

```
#ifndef __TEMPERATURE_H__
#define __TEMPERATURE_H__
extern void Temperature_Init(void);
extern unsigned int Read_Temperature(void);
#endif
```

“ADC.c”文件内容如下：

```
#include<ioCC2540.h>
#include"ADC.h"
1  #define HAL_ADC_REF_125V     0x00
2  #define ADC_12_BIT           0x30
```

```
3  #define ADC_TEMP_SENS        0x0E
4   #define ADC_TO_CELSIUS(ADC_VALUE)      ((ADC_VALUE>>4)-334);
 void Temperature_Init(void)
{
5  TR0|=0x01;
6  ATEST|=0x01;
}
unsigned int Read_Temperature(void)
{
  unsigned int value;
7  unsigned char tmpADCCON3=ADCCON3;
8  ADCIF=0;
9  ADCCON3=(HAL_ADC_REF_125V|ADC_12_BIT|ADC_TEMP_SENS);
10  while(!ADCIF);
11  value=ADCL>>2;
12  value|=((unsigned int)ADCH)<<6;
13  ADCCON3=tmpADCCON3;
14  return ADC_TO_CELSIUS(value);
}
```

第1行，选择内部参考电压用于序列转换。

第2行，设置抽取率，12位有效数字位。

第3行，选择温度传感器通道。

第4行，校正测量到的温度值。

第5～6行，使能内部温度传感器。

第7行，保存ADCCON3寄存器的值。

第8行，清除ADC中断标志位

第9行，配置ADCCON3寄存器，选择参考电压，ADC通道以及采样有效位。

第10行，等待转换结束。

第11～12行，获取采样得到的温度值。

第13行，恢复寄存器ADCCON3的值。

第14行，返回得到的温度值。

“main.c”文件内容如下：

```
#include"Led.h"
#include"uart.h"
#include"string.h"
#include"ADC.h"
```

```
unsigned char cmd;
void main(void)
{
  unsigned int tempvalue;
  unsigned char buf[4]="00C";
  Led_Init();
  Uart_Init();
  while(1)
  {
  1  tempvalue=Read_Temperature();
  2   sprintf(buf, (char *)"%dC", (unsigned char)
tempvalue);
  3  Uart_Print(buf, strlen(buf));
  4  Delay(200);
  }
}
```

第 1 行，获取温度值，赋值给一个临时变量。

第 2 行，将温度值格式化为指定的数据类型，存储到一个数组 buf 中。

第 3 行，向串口输出测得的温度值。

第 4 行，调用延时函数，延时一段时间。

将程序下载到开发板，打开串口调试助手，设置好波特率，将手指按在 CC2540 单片机上，会发现温度值在变化，如图 4-15 所示。

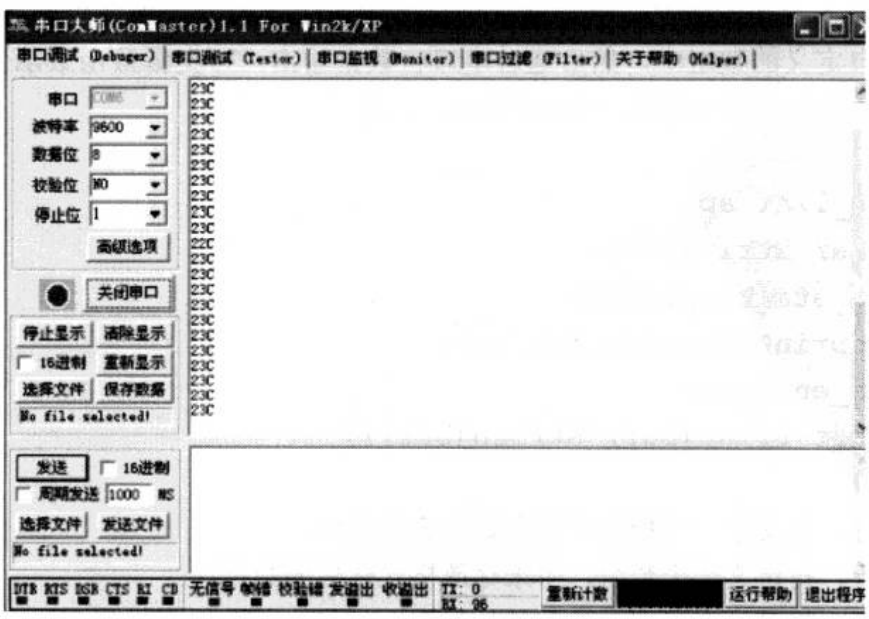


图 4-15　采集到的温度值

注意 ▶▶▶

由于CC2540集成的温度传感器精度较低，因此，一般不用来测量温度，这里只是向读者展示ADC的基本使用方法。

4.9 本章小结

本章在讲解CC2540单片机基础实验的基础上，穿插讲述了一些程序编写的技巧。读者通过本章的学习，可以初步掌握CC2540单片机I/O口、定时器、串口、ADC等硬件资源的使用方法，为以后的程序开发奠定基础。

[扩展阅读] 可变参数函数的使用技巧

在C语言中，使用printf（）进行格式化输出非常方便，例如，使用printf（“%d\n”,a）可将a的值以十进制格式输出，然后换行。printf（）的函数原型为：int printf(const char*format,...)，函数参数中的“...”表示可变参数，即参数的个数是可变的，例如函数printf("%d %d\n",a,b)中有两个参数a和b，这种参数个数不确定的函数就称为可变参数函数。读者在调试串口实验时，如果遇到多个输入参数的情况，可尝试采用可变参数函数的方法来将数据格式化后存放到一个自定义的数组中，然后通过串口将数据输出。可变参数函数示例如下：

```
Void Printf(const char*fmt,...)
{
1  va_list ap;
2  char string[50];
3  va_start(ap,fmt);
4  vsprintf(string,fmt,ap);
5  va_end(ap);
6  Uart_Print(buf, strlen(buf));
}
```

第1行，定义了一个指向可变参数列表的指针。

第2行，自定义一个数组，用来存放格式化后的数据。

第3行，将第一个可变参数的地址赋给ap指针，即ap指针指向可变参数列表

的开始。fmt 是最后一个固定参数。例如，若函数声明是 void va_test(char a, char b, char c, ...)，则它的固定参数依次为 a，b，c，最后一个固定参数为 c，此时，就要写为 va_start(ap, c)；

第 4 行，vsprintf() 的函数原型为 int vsprintf(char*string, char*format, va_list param)，其作用是将 param 按 format 格式写入到字符串 string 中。

第 5 行，清空参数列表，将参数指针 ap 置为无效。该“宏”的作用是结束可变参数的获取。

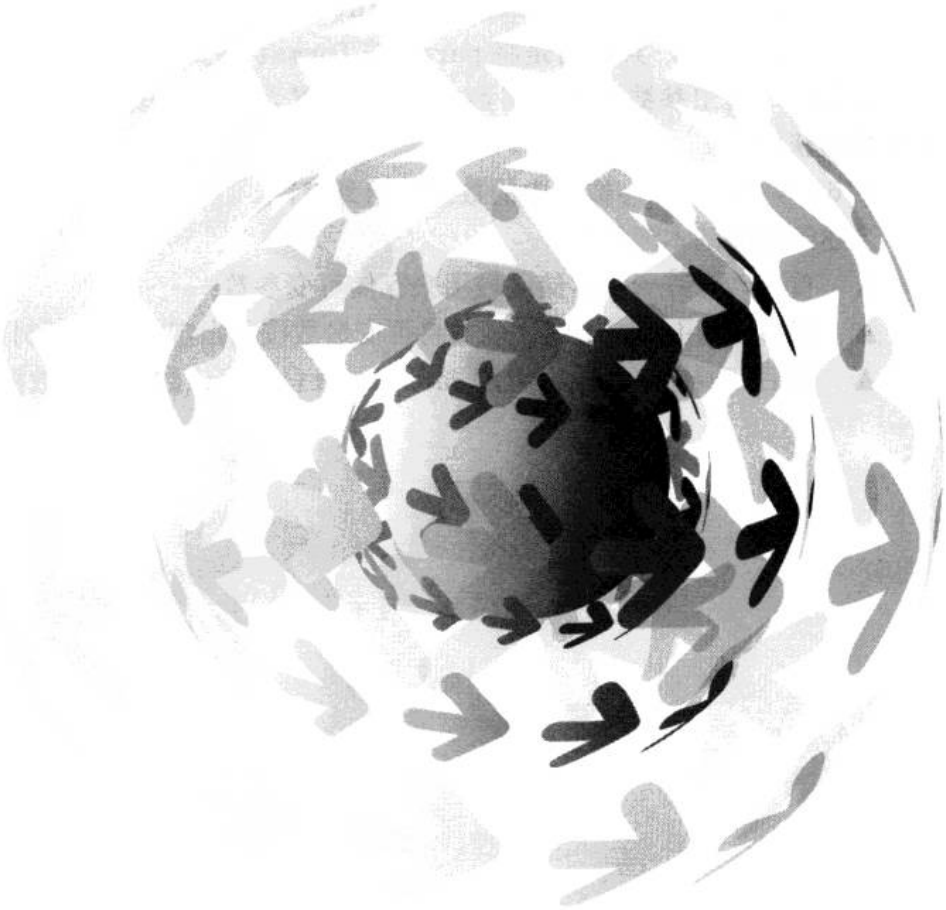
第 6 行，将 string 数组中的数据输出到串口。

上述函数的基本流程是：

▲ 首先，开辟一块存储区域用来存储可变参数；

▲ 然后，调用 vsprintf() 函数将可变参数按照指定的格式复制到一个数组中；

▲ 最后，调用串口输出函数 Uart_Print()，将该数组中的数据输出到串口。



第 5 章

蓝牙 4.0 BLE 技术入门

蓝牙无线通信技术涉及到电子电路、通信、射频等多学科的知识，这对于初学者来说，有一定的学习难度，很多读者看蓝牙 4.0 BLE 协议、射频电路……学了很长时间，但是连基本的点对点通信都无法实现，更别说基于对蓝牙 4.0 BLE 协议栈的理解来进行应用开发了。

基于此原因，本书推荐另一种学习思路，初学时不要将学习重点放在复杂的蓝牙 4.0 BLE 协议、射频、天线等知识上面，而是直接进行蓝牙 4.0 BLE 点对点通信的学习，基本思路是：从发送端发送一个数据，接收端接收到数据后，校验收到的数据是否正确，并给出相应的显示。很简单的功能，但是这里涉及到以下问题：

▲ 数据在协议栈里面是如何流动的；

▲ 如何调用蓝牙 4.0 BLE 协议栈提供的发送函数；

▲ 如何使用蓝牙 4.0 BLE 协议栈进行数据的接收；

▲ 如何理解蓝牙 4.0 BLE 协议栈；

▲ 蓝牙 4.0 BLE 协议栈是采用分层的思想，各层都具有哪些功能；

▲ 如何利用蓝牙 4.0 BLE 协议栈提供的函数来实现基本的低功耗应用程序开发；

▲ 系统硬件对蓝牙 4.0 BLE 协议都提供了哪些支持。

……

一个看似简单的实验引起了读者对于蓝牙 4.0 BLE 技术方方面面的思考，也正是由于上述思考，笔者才鼓起勇气带领读者去探究蓝牙 4.0 BLE 无线通信技术的原理，触摸那神秘的无线通信世界，读者的蓝牙 4.0 BLE 应用开发之旅将由此开始……

本章带领读者从功能上理解蓝牙 4.0 BLE 协议栈，并对协议栈几个主要的层进行详细讲解，初步展示蓝牙 4.0 BLE 协议栈中数据的传输及处理流程，在此基础上讲解一下蓝牙4.0 BLE协议栈中的串口工作原理，通过本章的学习，读者会对蓝牙4.0 BLE 协议栈有一个感性的认识。

5.1 蓝牙 4.0 BLE 协议栈

进行蓝牙 4.0 BLE 应用的开发，首先面临的问题是什么？是蓝牙 4.0 BLE 协议栈，以及由此引发的如下问题：

▲ 蓝牙 4.0 BLE 协议栈和蓝牙 4.0 BLE 协议是什么关系；

▲ 如何使用蓝牙 4.0 BLE 协议栈进行应用程序的开发。

5.1.1 什么是蓝牙 4.0 BLE 协议栈

协议定义的是一系列的通信标准，通信双方需要共同按照这一标准进行正常的数据收发；协议栈是协议的具体实现形式，通俗的理解为用代码实现的函数库，以

便于开发人员调用。

蓝牙4.0 BLE协议栈就是将各个层定义的协议都集合在一起，以函数的形式实现，并提供一些应用层API，供用户调用。

注意 ▶▶▶

虽然协议是统一的，但是协议的具体实现形式是变化的，即不同厂商提供的协议栈是有区别的，例如：函数名称和参数列表可能有区别，读者在选择协议栈以后，需要学习具体的例子，查看厂商提供的Demo演示程序、说明文档（通常，实现协议栈的厂商会提供一些API手册供用户查询）来学习各个函数的使用方式，进而快速地使用协议栈进行应用程序的开发工作。

5.1.2 如何使用蓝牙4.0 BLE协议栈

既然蓝牙4.0 BLE协议栈已经实现了蓝牙4.0 BLE协议，那么用户就可以使用协议栈提供的API进行应用程序的开发，在开发过程中不必过多地关注蓝牙4.0 BLE协议的具体实现细节，只需要关注一个核心的问题：应用程序数据从哪里来到哪里去。

至于调用协议栈中函数后，如何初始化硬件进行数据发送等工作，蓝牙4.0 BLE协议栈已经完成了所需要的初始化。

5.1.3 蓝牙4.0 BLE协议栈的安装、编译与下载

蓝牙4.0 BLE协议栈具有很多版本，不同厂商提供的蓝牙4.0 BLE协议栈有一定的区别，本书选用TI公司推出的蓝牙4.0 BLE协议栈进行讲解。

蓝牙4.0 BLE协议栈BLE-CC254x-1.2.1（可以在TI的官方网站进行下载）需要安装以后才能使用，下面讲解安装步骤。

从TI官方网站下载蓝牙4.0 BLE协议栈BLE-CC254x-1.2.1.exe，双击LE-CC254x-1.2.1.exe即可进行协议栈的安装，默认是安装到C盘根目录下。在路径C:\Texas Instruments\BLE-CC254x-1.2.1\Projects\ble\SimpleBLEPeripheral\CC2540DB下找到SimpleBLEPeripheral.eww，如图5-1所示，打开该工程即可。

图 5-1　SimpleBLEPeripheral.eww 工程路径

打开该工程后，可以看到 SimpleBLEPeripheral 工程文件布局如图 5-2 所示。

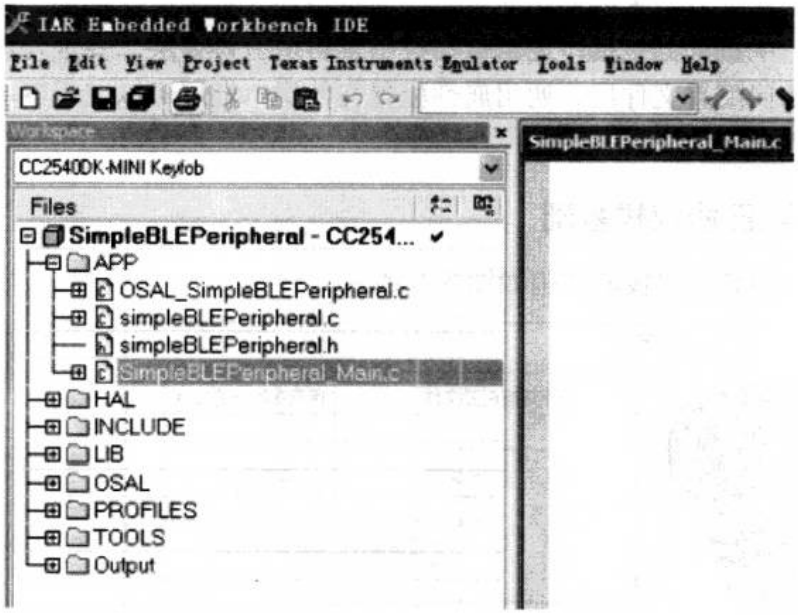


图 5-2　SimpleBLEPeripheral 工程文件布局

在如图 5-2 所示的文件布局中，左侧有很多文件夹，如 APP、HAL、OSAL、PROFILES 等，这些文件夹对应了蓝牙 4.0 BLE 协议中不同的层，使用蓝牙 4.0 BLE 协议栈进行应用程序的开发，一般只需要修改 APP 目录下的文件即可。

> **提示 ▶▶▶**
>
> 关于协议栈各层数据传输的流程，请读者结合后续章节的数据传输实验进行学习，通过具体的实例来学习应用开发的基本流程是笔者推荐的学习方法。

使用蓝牙 4.0 BLE 协议栈进行应用程序开发过程中，虽然说读者可以不必关心蓝牙 4.0 BLE 协议栈的具体细节，但是读者需要对蓝牙 4.0 BLE 协议栈的基本构成与内部工作原理有个清晰的认识，只有这样才能将蓝牙 4.0 BLE 协议栈提供的函数充分地融入到自己的实际项目开发过程中。

5.2 深入理解蓝牙 4.0 BLE 协议栈的构成

TI 的蓝牙 4.0 BLE 协议栈包含两部分：主机和控制器。主机和控制器的分离要追溯到蓝牙 BR/EDR 设备时期，控制器和主机通常会分开实现。

协议栈的实现方式采用分层的思想，控制器部分包括：物理层、链路层、主机控制接口层；主机部分包括：逻辑链路控制及自适应协议层、安全管理层、属性协议层、通用访问配置文件层、通用属性配置文件层；上层可以调用下层提供的函数来实现需要的功能。

5.2.1BLE 协议栈基础

蓝牙 4.0 BLE 协议栈的结构如图 5-3 所示。

图 5-3　蓝牙 4.0 BLE 协议栈

物理层（Physical Layer）是 1Mbps 自适应跳频的 GFSK 射频，工作于免许可证的 2.4GHz ISM（工业、科学和医疗）频段。

链路层（Link Layer）用于控制设备的射频状态，设备将会处于五种状态之一：等待、广告、扫描、初始化、连接。广播设备不需要建立连接就可以发送数据；而扫描设备收听广播设备发出的数据；发起连接的设备通过发送连接请求来回应广播设备，如果广播设备接受连接请求，那么广播设备与发起连接的设备将会进入连接状态。发起连接的设备称为主机，接受连接请求的设备称为从机。

主机控制接口层（Host Controller Interface）为主机和控制器之间提供标准通信接口。这一层可以是软件 API 或硬件接口，如 UART、SPI、USB。

逻辑链路控制及自适应协议层（Logical Link Control and Adaptation Protocol）为上层提供数据封装服务，允许逻辑上的点对点数据通信。

安全管理层（Security Manager）定义了配对和密钥分配方式，并为协议栈其他层与另一个设备之间的安全连接和数据交换提供服务。

属性协议层（Atttibute protocol）允许设备向另一个设备展示一块特定的数据，称之为“属性”。在 ATT 环境中，展示“属性”的设备称为服务器，与之配对的设备称为客户端。链路层状态（主机和从机）与设备的 ATT 角色是相互独立的。例如，主机设备既可以是 ATT 服务器，也可以是 ATT 客户端；从机设备既可以是 ATT 服务器，也可以是 ATT 客户端。

通用属性配置文件层（Generic Attribute profile）定义了使用 ATT 的服务框架。GATT 规定了配置文件（profile）的结构。在 BLE 中，所有被 profile 或服务用到的数据块都成为“特性”，两个建立连接的设备之间的所有数据通信都是通过 GATT 子程序处理。应用程序和 profiles 直接使用 GATT 层。

5.2.2BLE 协议栈开发平台配置

TI 公司免费的蓝牙 4.0 BLE 软件开发套件是完整地支持单模蓝牙 4.0 BLE 应用开发的平台，它基于 CC2540/CC2541 射频单片机，是一套完整的 SoC 解决方案。

蓝牙 4.0 BLE 软件平台支持两种不同的应用开发配置。

（1）单一设备

控制器、主机、配置文件（Profiles）、和应用程序在一片 CC2540 上实现，这是最简单和最常见的配置。这种方式提供最低的成本和功耗，大部分实际应用都采用这种方式。SimpleBLEPeripheral 和 SimpleBLECentral 示例工程都是采用单一设备的配置方式。单一设备配置如图 5-4 所示。

图 5-4 单一设备配置

（2）网络处理器

控制器和主机部分在CC2540上执行，而应用程序和Profiles在另一个设备执行。应用程序和Profiles通过厂商特定的HCI命令与CC2540通信，这一过程需要使用硬件或UART接口，或者通过USB使用虚拟的UART接口。网络处理器配置适用于应用程序在另一个设备（外部微控制器或PC）上运行的情况。在这种情况下，应用程序可以在外部独立开发，而协议栈仍然在CC2540上运行。要使用网络处理器，HostTestRelease工程必须使用。网络处理器配置如图5-5所示。

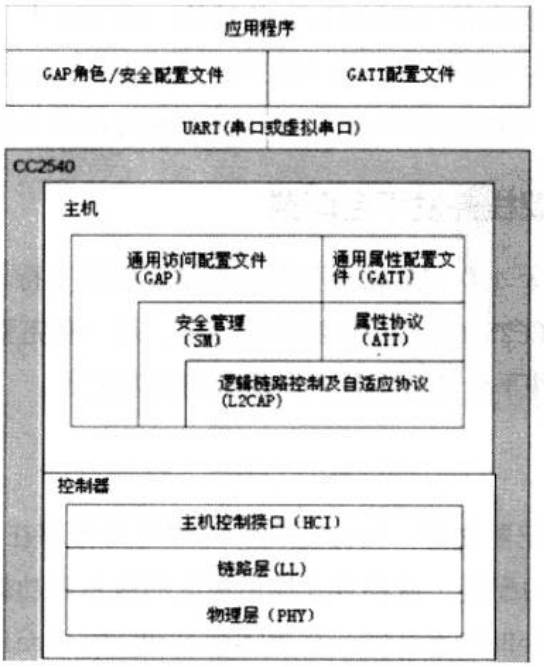


图 5-5 网络处理器配置

5.2.3 BLE 协议栈执行流程浅析

在 IAR 工程的左侧有很多文件夹，如 APP、HAL、OSAL、PROFILES 等，如图 5-6 所示，这些文件夹下面包含了很多源代码，这种实现方式与蓝牙 4.0 BLE 协议的分层思想是相对应的，尽量将实现某些功能的函数放在同一个文件夹下。

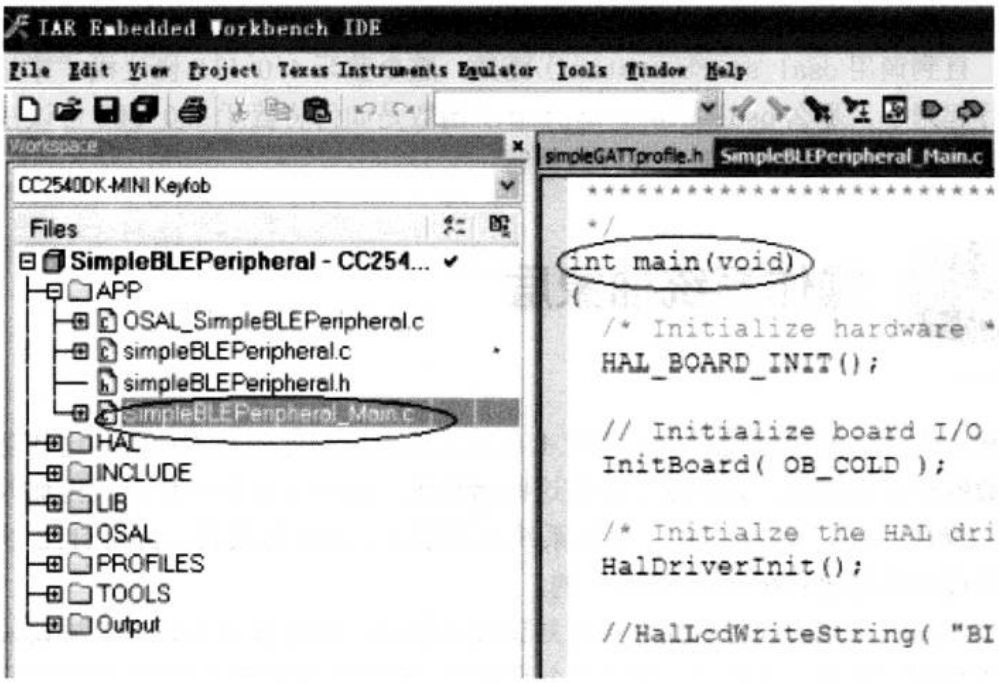


图 5-6 工程主界面

那么，整个协议栈是从哪里开始执行的呢？请读者观察图 5-6，打开 APP 文件夹下的 SimpleBLEPeripheral.c 文件，可以找到 main() 函数，这就是整个协议栈的入口点，即从该函数开始执行！

下面看一下 main() 函数主要做了哪些工作，main() 函数原型如下：

```
int main(void)
{
  HAL_BOARD_INIT();
  InitBoard( OB_COLD );
  HalDriverInit();
  osal_snv_init();
  osal_init_system();
  HAL_ENABLE_INTERRUPTS();
  InitBoard( OB_READY );
  #if defined ( POWER_SAVING )
  osal_pwrmgr_device( PWRMGR_BATTERY );
```

```
    #endif
    osal_start_system();
    return 0;
}
```

在 main() 函数中调用了很多其他文件中的函数，在此可以暂不考虑，重点是 osal_start_system() 函数，在此之前的函数都是对板载硬件以及协议栈进行的初始化，直到调用 osal_start_system() 函数，整个蓝牙 4.0 BLE 协议栈才算是真正的运行起来了，那么 osal_start_system() 函数是如何将蓝牙 4.0 BLE 协议栈调动起来的呢？下面将进行这部分内容的讲解。

5.3 操作系统抽象层

蓝牙 4.0 BLE 协议栈包含了蓝牙 4.0 BLE 协议所规定的基本功能，这些功能是以函数的形式实现的，为了便于管理这些函数集，蓝牙 4.0 BLE 协议栈内加入了一个小的操作系统，称为 OSAL（操作系统抽象层）。BLE 协议栈、profiles 以及所有应用程序都是建立在 OSAL 基础上的。

非计算机专业的读者对操作系统知识较为欠缺，但是蓝牙 4.0 BLE 协议栈内嵌的操作系统很简单，读者只需要做几个小实验，会很快掌握整个 OSAL 的工作原理。

OSAL (Operating System Abstraction Layer)，即操作系统抽象层，如何理解 OSAL 呢，从字面意思看是跟操作系统有关，但是后面为什么又加上“抽象层”呢，在蓝牙 4.0 BLE 协议栈中，OSAL 有什么作用呢？下面将对上述问题进行讨论。

5.3.1 OSAL 常用术语

在讲解之前，先介绍一下操作系统有关的部分基础知识。

（1）资源（Resource）

任何任务所占用的实体都可以称为资源，如一个变量、数组、结构体等。

（2）共享资源（Shared Resource）

至少可以被两个任务使用的资源称为共享资源，为了防止共享资源被破坏，每个任务在操作共享资源时，必须保证是独占该资源。

（3）任务（Task）

一个任务，又称作一个线程，是一个简单的程序的执行过程，在任务执行过程中，可以认为 CPU 完全属于该任务。在任务设计时，需要将问题尽可能地分为多个

任务，每个任务独立完成某种功能，同时被赋予一定的优先级，拥有自己的 CPU 寄存器和堆栈空间。一般将任务设计为一个无限循环。

（4）多任务运行（Muti-task Running）

实际上，一个时间点只有一个任务在运行，但是 CPU 可以使用任务调度策略将多个任务进行调度，每个任务执行特定的时间，时间片到了以后，就进行任务切换，由于每个任务执行时间很短，例如：10ms，因此，任务切换很频繁，这就造成了多任务同时运行的“假象”。

（5）内核（Kernel）

在多任务系统中，内核负责管理各个任务，主要包括：为每个任务分配 CPU 时间；任务调度；负责任务间的通信。内核提供的基本的内核服务是任务切换。使用内核可以大大简化应用系统的程序设计方法，借助内核提供的任务切换功能，可以将应用程序分为不同的任务来实现。

（6）互斥（Mutual Exclusion）

多任务间通信最简单，常用的方法是使用共享数据结构。对于单片机系统，所有任务都在单一的地址空间下，使用共享的数据结构包括全局变量、指针、缓冲区等。虽然共享数据结构的方法简单，但是必须保证对共享数据结构的写操作具有唯一性，以避免晶振和数据不同步。

保护共享资源最常用的方法是：

▲ 关中断；

▲ 使用测试并置位指令（T&S 指令）；

▲ 禁止任务切换；

▲ 使用信号量。

其中，在蓝牙 4.0 BLE 协议栈内嵌操作系统中，经常使用的方法是关中断。

（7）消息队列（Message Queue）

消息队列用于任务间传递消息，通常包含任务间同步的信息。通过内核提供的服务、任务或者中断服务程序将一条消息放入消息队列，然后，其他任务可以使用内核提供的服务从消息队列中获取属于自己的消息。为了降低传递消息的开支，通常传递指向消息的指针。

在蓝牙 4.0 BLE 协议栈中，OSAL 主要提供如下功能：

▲ 任务注册、初始化和启动；

▲ 任务间的同步、互斥；

▲ 中断处理；

▲ 存储器分配和管理；

▲ 提供定时器功能。

5.3.2 OSAL 运行机理

在基于蓝牙 4.0 BLE 协议栈的应用程序开发过程中，读者只需要实现应用层的程序开发即可。可以把一个应用程序对象看作为一个任务，因此，需要一个机制来实现任务的切换、同步与互斥，这就是 OSAL 产生的根源。从上面的分析可以得出下面的结论：

OSAL 就是一种支持多任务运行的系统资源分配机制。

OSAL 与标准的操作系统还是一定区别的，OSAL 实现了类似操作系统的某些功能，例如：任务切换、提供了内存管理功能等，但 OSAL 并不能称之为真正意义上的操作系统。

在蓝牙 4.0 BLE 协议栈中，OSAL 负责调度各个任务的运行，如果有事件发生了，则会调用相应的事件处理函数进行处理，OSAL 的工作原理示意图如图 5-7 所示。

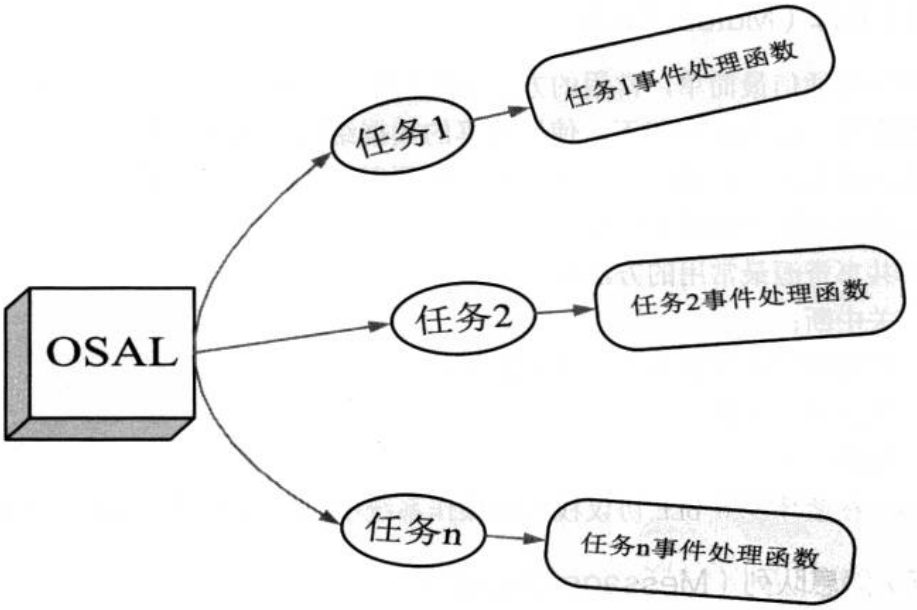


图 5-7 OSAL 的工作原理示意图

那么，事件和任务的事件处理函数是如何联系起来的呢？

蓝牙 4.0 BLE 中采用的方法是：建立一个事件表，保存各个任务对应的事件，建立另一个函数表，保存各个任务事件处理函数的地址，然后将这两张表建立某种对应关系，当某一事件发生时则查找函数表找到对应的事件处理函数即可。

现在问题转变为：用什么样的数据结构来实现事件表和函数表呢？如何将事件表和函数表建立对应关系呢？可以说，只要将上述两个问题解决，在整个协议栈的开发将会变的很容易。

在蓝牙 4.0 BLE 协议栈中，有三个变量至关重要。

▲ tasksCnt：该变量保存了任务的总个数。

该变量的声明为：uint8 tasksCnt。其中 uint8 的定义为：typedef unsigned char uint8。

▲ tasksEvents：这是一个指针，指向了事件表的首地址。

该变量的声明为：uint16 *tasksEvents。其中 uint16 的定义为：typedef unsigned short uint16。

▲ tasksArr：这是一个数组，该数组的每一项都是一个函数指针，指向了事件处理函数。

该数组的声明为：pTaskEventHandlerFn tasksArr[]。其中 pTaskEventHandlerFn 的定义（需要特别注意）如下：

typedef unsigned short (*pTaskEventHandlerFn)(unsigned char task_id, unsigned short event)

这是定义了一个函数指针。

因此，tasksArr 数组的每一项都是一个函数指针，指向了对应的事件处理函数。

事件表和函数表的关系如图 5-8 所示。

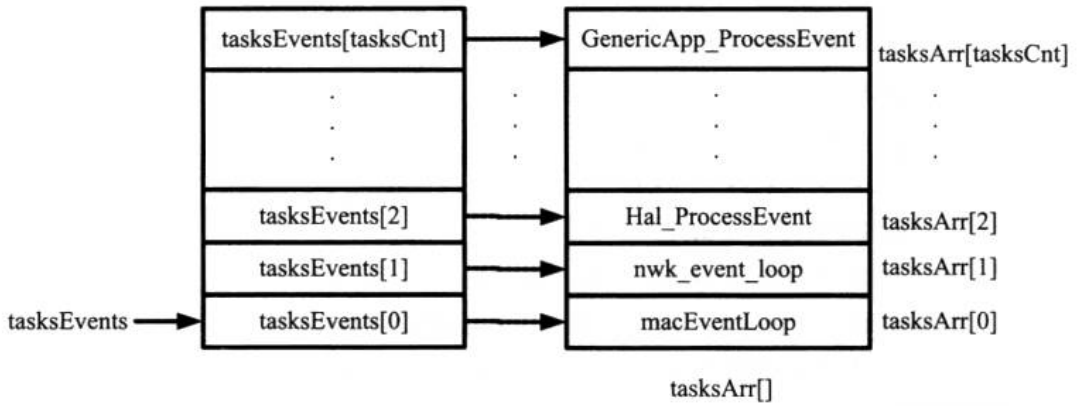


图 5-8 事件表和函数表的关系

讲到这里，可以总结一下 OSAL 的工作原理：通过 tasksEvents 指针访问事件表的每一项，如果有事件发生，则查找函数表找到事件处理函数进行处理，处理完后，继续访问事件表，查看是否有事件发生，无限循环。

从这种意义上说，OSAL 是一种基于事件驱动的轮询式操作系统。事件驱动是指发生事件后采取相应的事件处理方法，轮询指的是不断地查询是否有事件发生。

前文提到，在 main() 函数中，直到调用 osal_start_system() 函数时，整个蓝牙 4.0 BLE 协议栈才算是真正地运行起来了，下面将深入到 osal_start_system() 函数的内部去探究协议栈是如何被调动起来的。

osal_start_system() 函数原型如下：

```
void osal_start_system( void )
{
#if !defined ( ZBIT ) && !defined ( UBIT )
  for(;;)
#endif
  {
    uint8 idx = 0;
    osalTimeUpdate();
    Hal_ProcessPoll();
    do {
        if (tasksEvents[idx])
        {
            break;
        }
      } while (++idx < tasksCnt);
      if (idx < tasksCnt)
      {
      uint16 events;
      halIntState_t intState;
      HAL_ENTER_CRITICAL_SECTION(intState);
      events = tasksEvents[idx];
      tasksEvents[idx] = 0; .
      HAL_EXIT_CRITICAL_SECTION(intState);
      events = (tasksArr[idx])( idx, events );
      HAL_ENTER_CRITICAL_SECTION(intState);
      tasksEvents[idx] |= events;
      HAL_EXIT_CRITICAL_SECTION(intState);
      }
  #if defined( POWER_SAVING )
      else
      {
        osal_pwrmgr_powerconserve();
      }
  #endif
    }
}
```

为了看清 osal_start_system() 函数的工作机制，可以将条件编译指令去掉，

此外，前文讲到，在访问共享变量时需要保证该变量不被其他任务同时访问，因此，这里采用的是关中断的方法，如下代码块就是一种典型的使用方法。

```
HAL_EXIT_CRITICAL_SECTION(intState);
    ......
HAL_EXIT_CRITICAL_SECTION(intState);
```

使用 HAL_EXIT_CRITICAL_SECTION 关中断，访问完共享变量后，使用 HAL_EXIT_CRITICAL_SECTION 宏恢复中断即可。

因此，为了分析问题方便，可以将这些宏去掉，经过简化的 osal_start_system() 函数如下（简化后便于分析，同时工作原理没有变化）：

```
1  void osal_start_system( void )
   {
2     for(;;)
      {
3       uint8 idx  =  0;
4       osalTimeUpdate();
5       Hal_ProcessPoll();
6       do
        {
7         if (tasksEvents[idx])
          {
8           break;
          }
9     } while (++idx < tasksCnt);
10    if (idx < tasksCnt)
      {
11      uint16 events;
12      events  =  tasksEvents[idx];
13      tasksEvents[idx]  =  0; .
14      events  =  (tasksArr[idx])( idx, events );
15      tasksEvents[idx]  |=  events;
      }
   }
  }
```

这个函数是蓝牙 4.0 BLE 协议栈的灵魂，实现的功能是：不断地查看事件表，如果有事件发生就调用相应的事件处理函数。

第 2 行，整个函数体包含在 for 循环中，不断循环。

第 3 行，定义了一个变量 idx，用来在事件表中索引。

第4、5两行，更新系统时钟，同时查看硬件方面是否有事件发生，如串口是否收到数据、是否有按键按下等信息，这部分内容在此可以暂时不予考虑。

第6～9行，使用do-while循环查看事件表是否有事件发生。

这里需要注意一点，如何表示一个事件呢？

蓝牙4.0 BLE协议栈使用一个unsigned short型的变量，因为unsigned short类型占2个字节，即16个二进制位，因此，可以使用每个二进制位表示一个事件，如表5-1所示。

表5-1 使用二进制位表示不同事件

事件	十六进制	二进制
启动设备	0x01	0b00000001
周期事件	0x02	0b00000010
连接状态继续广播	0x04	0b00000100

在系统初始化时，将所有任务的事件初始化为0，因此，第7行通过tasksEvents[idx]是否为0来判断是否有事件发生，如果有事件发生了，则跳出循环。

第11～12行，读取该事件。

第13行，将事件表中该项事件标志清零，注意有可能几个事件同时发生，这里清零是暂时的。

第14行，调用事件处理函数去处理，为什么是调用事件处理函数呢？还记得tasksArr[]数组中每个元素的类型吗？函数指针！这就是函数指针的典型应用。执行完事件处理函数后，需要将未处理的事件返回，即事件处理函数的返回值保存了未处理的事件，需要将未处理的事件再次存放到事件表中，以便于下次进行处理。

第15行，将未处理的事件重新存放到事件表中。

现在遇到的问题是：如何在事件处理函数中返回未处理的事件呢？下面结合SimpleBLEPeripheral_ProcessEvent函数来讲解。SimpleBLEPeripheral_ProcessEvent函数原型如下：

```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
uint16 events )
{
1  if ( events & SYS_EVENT_MSG )
  {
    uint8 *pMsg;
2   if ( (pMsg = osal_msg_receive( simpleBLEPeripheral_
TaskID )) != NULL )
    {
3   simpleBLEPeripheral_ProcessOSALMsg( (osal_event_hdr_
```

```
t *)pMsg );
   4    VOID osal_msg_deallocate( pMsg );
     }
   5  return (events ^ SYS_EVENT_MSG);
     }
   6  if ( events & SBP_START_DEVICE_EVT )
     {
   7    VOID GAPRole_StartDevice( &simpleBLEPeripheral_
PeripheralCBs );
   8  VOID GAPBondMgr_Register( &simpleBLEPeripheral_
BondMgrCBs );
   9  osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_
PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
   10  return ( events ^ SBP_START_DEVICE_EVT );
     }
   11  if ( events & SBP_PERIODIC_EVT )
     {
   12  if ( SBP_PERIODIC_EVT_PERIOD )
       {
   13  osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_
PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
       }
   14  performPeriodicTask();
   15  return (events ^ SBP_PERIODIC_EVT);
     }
   16  #if defined ( PLUS_BROADCASTER )
   17  if ( events & SBP_ADV_IN_CONNECTION_EVT )
     {
       uint8 turnOnAdv = TRUE;
   18  GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof(
uint8 ), &turnOnAdv );
   19  return (events ^ SBP_ADV_IN_CONNECTION_EVT);
     }
   #endif
   20  return 0;
   }
```

代码解析：

第1行，判断是否有系统消息任务，如果有，则定义一个消息指针pMsg。

第2行，判断是否从消息队列接收到数据。

第3行，处理从消息队列接收到的任务信息。

第4行，释放消息的缓存空间。

第5行，返回未处理的任务标志。

第6行，判断是否有启动设备任务。

第7行，启动设备，括号内为设备状态改变时的回调函数。在回调函数中，读者可以自定义设备启动时想要显示的信息或要执行的操作。

第8行，启动绑定管理函数。在函数中处理认证信息并在GATT服务应用程序中注册任务信息。

第9行，设置一个定时时间，当定时时间到后，周期事件的任务ID被置起。关于周期事件后面会结合例子具体讲解。

第10行，返回未处理的任务标志。

第11行，判断是否有周期事件。

第12～15行，如果有周期事件任务，设置周期事件的定时时间。当时间到达时，执行周期事件中需要处理的工作，具体的任务工作读者可以根据需要自定义。最后返回未处理的任务标志。

第16行，PLUS_BROADCASTER为宏定义，本行含义为判断PLUS_BROADCASTER是否已经在前面定义了。注意：部分宏定义可在编译器设置项中配置。

第17～19行，如果有连接状态广告事件，则调用GAPRole_SetParameter（）函数，实现设备处于连接状态时，仍然能发送广告的功能。

第20行，将未知的任务事件标志清零并返回。

SimpleBLEPeripheral_ProcessEvent函数的基本实现方法是：使用osal_msg_receive函数从消息队列上接收一个消息（在该消息中包含了事件以及接收到的数据），然后使用switch-case语句判断事件类型，根据事件的类型，调用相应的消息事件处理函数。

注意加粗部分的return语句，使用了异或运算，可以通过下面的例子看到异或运算的作用。

使用表5-1中的事件，假设同时发生了启动设备和连接状态继续广播事件，则此时events=0b00000101，即0x05，假设现在处理完周期事件，则应该将events第0位清零，如何实现呢？只需要使用异或运算即可，即events^0x01=0x5^0x01=0x04，即0b00000100，可见使用异或运算恰好可以将处理完的事件清除，仅留下未处理的事件。

前文讲到可以使用每个二进制位表示一个事件，因此在蓝牙4.0 BLE协议栈中，读者可以自己定义事件，但是，协议栈同时也给出了几个已经定义好的事件，由协议栈定义的事件成为系统强制事件（Mandatory Events），SYS_EVENT_MSG就是其中的一个事件，SYS_EVENT_MSG的定义如下：#define SYS_EVENT_MSG　0x8000

可以将 OSAL 的运行机理总结为：

▲ 通过不断地查询事件表来判断是否有事件发生，如果有事件发生，则查找函数表找到对应的事件处理函数对事件进行处理；

▲ 事件表使用数组来实现，数组的每一项对应一个任务的事件，每一位表示一个事件；函数表使用函数指针数组来实现，数组的每一项是一个函数指针，指向了事件处理函数。

5.3.3 OSAL 消息队列

讲解消息队列之前需要讲解一下消息与事件的区别。

事件是驱动任务去执行某些操作的条件，当系统中产生了一个事件，OSAL 将这个事件传递给相应的任务后，任务才能执行一个相应的操作（调用事件处理函数去处理）。

通常某些事件发生时，又伴随着一些附加信息的产生，例如：从天线接收到数据后，会产生消息，但是任务的事件处理函数在处理这个事件的时候，还需要得到天线接收到的数据。

因此，这就需要将事件和数据封装成一个消息，将消息发送到消息队列，然后在事件处理函数中就可以使用 osal_msg_receive，从消息队列中得到该消息。如下代码可以从消息队列中得到一个消息。

```
pMsg = osal_msg_receive( simpleBLETaskId );
```

OSAL 维护了一个消息队列，每一个消息都会被放到这个消息队列中去，当任务接收到事件后，可以从消息队列中获取属于自己的消息，然后调用消息处理函数进行相应的处理即可。

OSAL 中消息队列如图 5-9 所示。

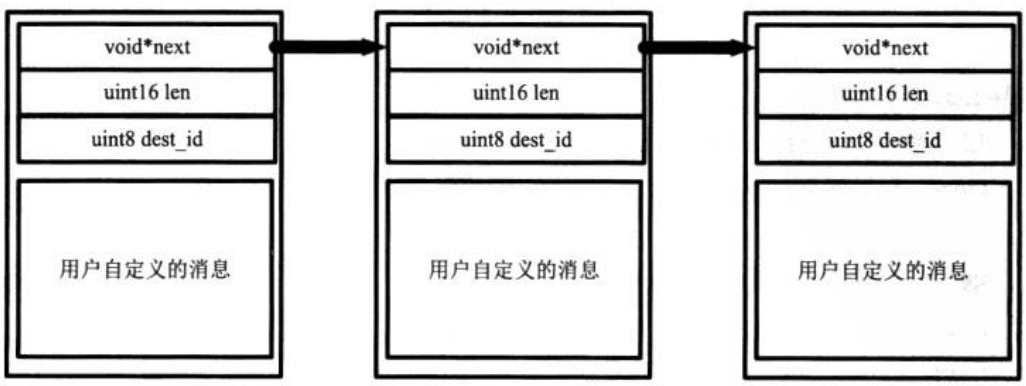


图 5-9 OSAL 中消息队列

每个消息都包含一个消息头 osal_msg_hdr_t 和自定义的消息，osal_msg_hdr_t 结构体的定义为：

```
typedef struct
```

```
{
  void        *next;
  uint16      len;
  uint8       dest_id;
} osal_msg_hdr_t;
```

5.3.4 OSAL 添加新任务

在使用蓝牙 4.0 BLE 协议栈进行应用程序开发时，如何在应用程序中添加一个新任务。下面结合示例工程 impleBLEPeripheral 来讲解，打开 OSAL_SimpleBLEPeripheral.c 文件可以找到数组 tasksArr[] 和函数 osalInitTasks( )。tasksArr[] 数组里存放了所有任务的事件处理函数的地址；osalInitTasks( ) 是 OSAL 的任务初始化函数，所有任务的初始化工作都在这里面完成，并且自动给每个任务分配一个任务 ID。

因此，要添加新任务，只需要编写两个函数：

▲ 新任务的初始化函数；

▲ 新任务的事件处理函数。

将新任务的初始化函数添加在 osalInitTasks（）函数的最后，如下代码所示。

```
const pTaskEventHandlerFn tasksArr[] =
{
  LL_ProcessEvent,      // task 0
  Hal_ProcessEvent,      // task 1
  HCI_ProcessEvent,      // task 2
#if defined ( OSAL_CBTIMER_NUM_TASKS )
  OSAL_CBTIMER_PROCESS_EVENT( osal_CbTimerProcessEvent ),
// task 3
#endif
  L2CAP_ProcessEvent,      // task 4
  GAP_ProcessEvent,      // task 5
  GATT_ProcessEvent,      // task 6
  SM_ProcessEvent,      // task 7
  GAPRole_ProcessEvent,      // task 8
  GAPBondMgr_ProcessEvent,      // task 9
  GATTServApp_ProcessEvent,      // task 10
  SimpleBLEPeripheral_ProcessEvent      // task 11
};
```

将事件处理函数的地址加入 tasksArr[] 数组，如下代码所示。

```
void osalInitTasks( void )
```

```
{
  uint8 taskID = 0;
  tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16
) * tasksCnt);
  osal_memset( tasksEvents, 0, (sizeof( uint16 ) *
tasksCnt));
  LL_Init( taskID++ );
  Hal_Init( taskID++ );
  HCI_Init( taskID++ );
#if defined ( OSAL_CBTIMER_NUM_TASKS )
  osal_CbTimerInit( taskID );
  taskID += OSAL_CBTIMER_NUM_TASKS;
#endif
  L2CAP_Init( taskID++ );
  GAP_Init( taskID++ );
  GATT_Init( taskID++ );
  SM_Init( taskID++ );
  GAPRole_Init( taskID++ );
  GAPBondMgr_Init( taskID++ );
  GATTServApp_Init( taskID++ );
  SimpleBLEPeripheral_Init( taskID );
}
```

在此例中，将 SimpleBLEPeripheral_ProcessEvent 函数添加到了数组的末尾，将 SimpleBLEPeripheral_Init( ) 函数添加到 osalInitTasks( ) 任务初始化函数中。

需要注意两点：

▲ tasskArr[] 数组里各事件处理函数的排列顺序要与 osalInitTasks( ) 函数中调用各任务初始化函数的顺序保持一致，只有这样才能够保证每个任务的事件处理函数能够接收到正确的任务 ID（在 osalInitTasks( ) 函数中分配）；

▲ 为了保存 osalInitTasks( ) 函数所分配的任务 ID，需要给每一个任务定义一个全局变量来保存这个 ID。如在 SimpleBLEPeripheral.c 中定义了一个全局变量 simpleBLEPeripheral_TaskID，并且在 SimpleBLEPeripheral_Init 函数中对其赋值。

5.3.5 OSAL 应用编程接口

既然蓝牙 4.0 BLE 协议栈内嵌了操作系统来支持多任务运行，那么任务间同步、互斥等都需要相应的 API 来支持，下面讲解一下 OSAL 提供的 API（应用编程接口，

Application Programming Interface)这些API允许协议栈以独立于特定操作系统、内核或任务环境的方式编写。

> **注意 ▶▶▶**
>
> 阅读本节时，读者只需要了解OSAL提供了哪些API函数即可，暂时不需要关注该函数的具体使用方法，很多函数使用几次后，自然就熟悉了。

总体而言，OSAL提供了以下8个方面的API：

- ▲ 消息管理；
- ▲ 任务同步；
- ▲ 时间管理；
- ▲ 中断管理；
- ▲ 任务管理；
- ▲ 内存管理；
- ▲ 电源管理；
- ▲ 非易失性闪存管理。

下面，选取部分典型的API进行讲解。

（1）消息管理API

消息管理有关的API主要用于处理任务间消息的交换，主要包括为任务分配消息缓存、释放消息缓存、接收消息和发送消息等API函数。

osal_msg_allocate( )

函数原型：uint8 *osal_msg_allocate(uint16 len)

功能描述：为消息分配缓存空间。

osal_msg_deallocate( )

函数原型：uint8 osal_msg_deallocate(uint8 *msg_ptr)

功能描述：释放消息的缓存空间。

osal_msg_send( )

函数原型：uint8 osal_msg_send(uint8 destination_task, uint8 *msg_ptr)

功能描述：一个任务发送消息到消息队列。

osal_msg_receive( )

函数原型：uint8 *osal_msg_receive(uint8 task_id)

功能描述：一个任务从消息队列接收属于自己的消息。

（2）任务同步API

任务同步API主要用于任务间的同步，允许一个任务等待某个事件的发生。

osal_set_event( )

函数原型：uint8 osal_set_event(uint8 task_id, uint16 event_flag)

功能描述：运行一个任务设置某一事件。

（3）时间管理 API

时间管理 API 用于开启和关闭定时器，定时时间一般为毫秒级定时，使用该 API，读者不必关心底层定时器是如何初始化的，只需要调用即可，在蓝牙 4.0 BLE 协议栈物理层已经完成了定时器的初始化。

osal_start_timerEx()

函数原型：uint8 osal_start_timerEx(uint8 task_id, uint16 event_id, uint16 timeout_value)

功能描述：设置一个定时时间，定时时间到后，相应的事件被设置。

osal_stop_timerEx()

函数原型：uint8 osal_stop_timerEx(uint8 task_id, uint16 event_id)

功能描述：停止已经启动的定时器。

（4）中断管理 API

中断管理 API 主要用于控制中断的开启与关闭。一般很少使用，所以在此不做赘述，如果读者需要使用相关的函数，请参考《OS Abstraction Layer Application Programming Interface》。

（5）任务管理 API

任务管理 API 主要功能是对 OSAL 进行初始化和启动，主要包括 osal_init_system() 和 osal_start_system()。

osal_init_system()

函数原型： uint8 osal_init_system(void)

功能描述：初始化 OSAL，该函数是第一个被调用的 OSAL 函数。

osal_start_system()

函数原型： uint8 osal_start_system(void)

功能描述：该函数包含一个无限循环，它将查询所有的任务事件，如果有事件发生，则调用相应的事件处理函数，处理完该事件后，返回主循环继续检测是否有事件发生，如果开启了节能模式，则没有事件发生时，该函数将使处理器进入休眠模式，以降低系统功耗。

（6）内存管理 API

内存管理 API 用于在堆上分配缓冲区，这里需要注意以下两个 API 函数一定要成对使用，防止产生内存泄露。

osal_mem_alloc()

函数原型： void *osal_mem_alloc(uint16 size)

功能描述：在堆上分配指定大小的缓冲区。

osal_mem_free()

函数原型： void osal_mem_free(void *ptr)

功能描述：释放使用 osal_mem_alloc() 分配的缓冲区。

（7）电源管理 API

电源管理 API 主要是用于电池供电的蓝牙 4.0 BLE 设备节点，在此暂不做讨论。

（8）非易失性闪存管理 API

非易失性闪存（Non-Volatile Memory，NV）管理 API 主要是添加了对非易失性闪存的管理函数，一般这里的非易失性闪存指的是系统的 Flash 存储器（也可以是 E2PROM），每个 NV 条目分配唯一的 ID 号。

osal_nv_item_init()

函数原型： byte osal_nv_item_init(uint16 id, uint16 len, void *buf);

功能描述：初始化 NV 条目，该函数检查是否存在 NV 条目，如果不存在，它将创建并初始化该条目。如果该条目存在，每次调用 osal_nv_read() 或 osal_nv_write() 函数对该条目进行读、写之前，都要调用该函数。

osal_nv_read()

函数原型： byte osal_nv_read(uint16 id, uint16 offset, uint16 len, void *buf);

功能描述：从 NV 条目中读取数据，可以读取整个条目的数据，也可以读取部分数据。

osal_nv_write()

函数原型： byte osal_nv_write(uint16 id, uint16 offset, uint16 len, void *buf);

功能描述：写数据到 NV 条目。

5.4 硬件抽象层

硬件抽象层（HAL）是提供硬件服务而又不涉及太多硬件细节的层，为应用提供访问 GPIO、UART、ADC 等硬件的接口。使用户可以专注于应用开发。

5.4.1 硬件抽象层文件目录

硬件抽象层文件目录包括 HAL 驱动和 HAL 相关文件的头文件，HAL 文件存放在

三个不同目录之下，如图 5-10 所示。

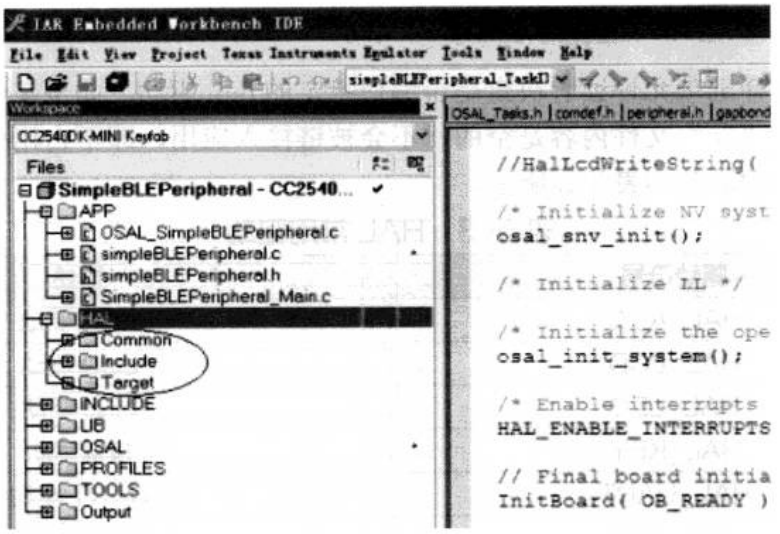


图 5-10　HAL 文件目录

HAL Common 文件夹包含协议栈、MAC 和驱动中使用到相关配置文件。其中“hal_drivers.c”文件包含所有驱动初始化及相关事件处理机制。主要包括以下三个函数:

▲ Hal_Init() 它由 osalTaskAdd 调用，以便在 OSAL 中注册 HAL 的驱动;

▲ HalDriverInit() 它由 main（）函数调用来初始化硬件驱动。读者也可以根据需要在这里添加硬件相关的初始化，所有这些初始化工作会在协议栈开始运行时完成;

▲ Hal_ProcessEvent（） 这个函数用来处理 OSAL 中 HAL 相关的驱动事件，例如按键、LED、LCD、蜂鸣器、休眠定时等，读者也可以根据需要在这里自定义 HAL 事件，需要注意的是，事件 ID 必须是唯一的，并且要在“hal_driver.h”文件中定义。

HAL Include 文件夹包含 HAL 驱动及 HAL 相关文件的头文件。如表 5-2 所示。

表 5-2　Include 文件下头文件

头文件	描述
hal_adc.h	ADC 驱动头文件
hal_key.h	按键驱动头文件
hal_lcd.h	LCD 驱动头文件
hal_led.h	LED 驱动头文件
hal_uart.h	串口驱动头文件
hal_timer.h	定时器驱动头文件

HAL Target 文件夹下的 Drivers 包含所有 HAL 驱动的“.c”文件。可以根据

需要修改“.c”文件中的内容来适应不同的硬件平台。

5.4.2 硬件抽象层驱动编译

下面的编译选项允许读者决定是否编译特定的HAL驱动，如果某个驱动不被编入，其对应的“.c”文件内容是空的，不会被链接入应用，即不占用代码空间。读者可选择编译的驱动如表5-3所示。

表5-3 HAL常用驱动

驱动符号	描述
HAL_ADC	ADC驱动
HAL_LCD	LCD驱动
HAL_LED	LED驱动
HAL_KEY	按键驱动
HAL_UART	UART驱动
HAL_DMA	DMA驱动
HAL_FLASH	FLASH驱动
HAL_AES	安全驱动
HAL_SPI	SPI驱动
HAL_TIMER	定时器驱动

读者可以在工程选项设置中编入需要的驱动，下面以SimpleBLEPeripheral工程为例进行讲解。Option → C/C++ compiler → Preprocessor选项卡中可以看到Defined symbol，里面已有一些预编译内容。通过键入HAL_XXX=TRUE，可以将对应的驱动加入编译。其中，XXX可以是ADC、KEY、UART、LED、LCD等，如图5-11所示。

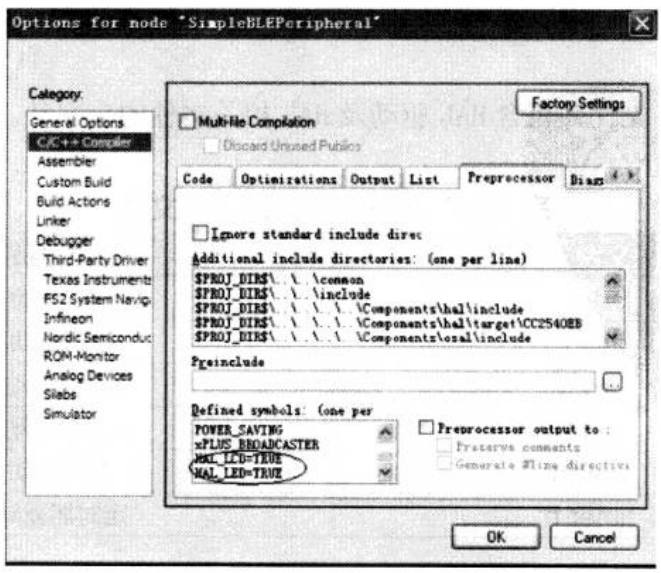


图5-11 将驱动加入编译

同样，读者可以将不需要的驱动编出。方法是在 Defined symbol 中，通过键入 HAL_XXX=FALSE，将对应的驱动编出，如图 5-12 所示。

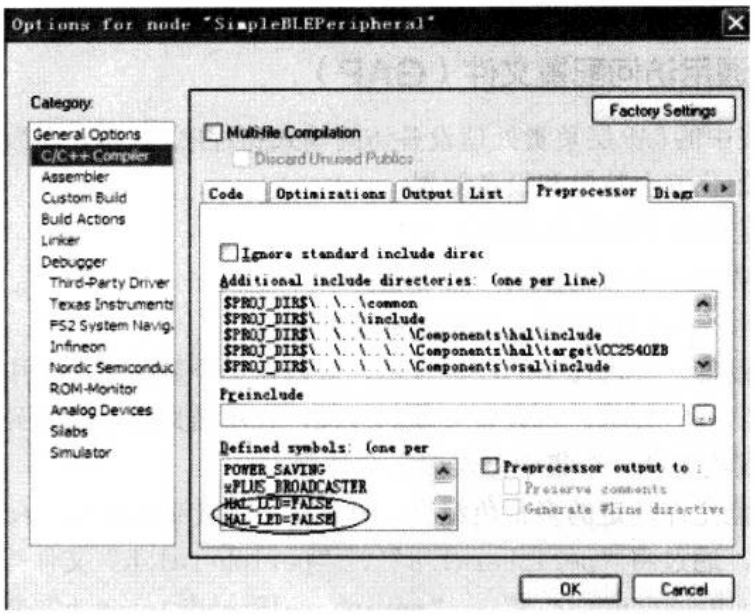


图 5-12　将驱动加入编译

5.4.3　用户自定义驱动

HAL 驱动具有很大的灵活性，用户可以根据自己的需要修改现有的驱动以适应特定的需求。当需要修改现有的驱动时，需要遵循以下规则：

▲　所有的 Include 中的头文件要保持不变，因为这些原型被程序其他部分使用，不能随意改变；

▲　每个驱动程序的内容通过修改 Drivers 文件夹下面的“.c”文件来改变；

▲　驱动硬件配置需要修改“hal_board_cfg.h”。

用户也可以增加新的驱动以适应不同的硬件平台，需要按以下步骤添加：

▲　在 Include 文件夹中增加新的头文件；

▲　将新增的驱动内容添加到 Drivers 文件夹下面的“.c”文件中；

▲　检查“hal_board_cfg.h”，查看 GPIO 是否正确设置，如果 GPIO 管脚冲突或不需要，可以按照上一节讲述的方法，将不需要的驱动编出。

5.5　配置文件层

蓝牙 4.0 BLE 协议栈的配置文件层包括 GAP 角色 / 安全配置文件、GATT 配置文

件两部分，处于协议栈的顶层，配置文件层将协议栈和应用紧密地联系在一起，这使得开发者即使对协议栈底层的原理没有深入的了解，也可以方便地进行应用的开发。

5.5.1 通用访问配置文件（GAP）

BLE协议栈中的GAP层负责处理设备访问模式和程序，包括设备发现、建立连接、终止连接、初始化安全特色和设备配置。

GAP层总是作为下面四种角色之一：

▲ 广播者——不可连接的广告设备；

▲ 观测者——扫描广播，但不发起建立连接；

▲ 外部设备——可连接的广告设备，可以在单个链路层连接中作为从机；

▲ 集中器——扫描广告设备并发起连接，在单链路层或多链路层作为主机。目前，BLE协议栈支持一个集中器连接三个外设。

低功耗规范允许特定的多角色组合。默认的SimpleBLEPeripheral示例工程只支持外设角色，通过将“peripheral.c”、“peripheral.h”文件排除编译，并添加“peripheralBroadcaster.c”、“peripheralBroadcaster.h”文件加入编译，可以使工程同时支持外设和广播者角色，如图5-13和图5-14所示。

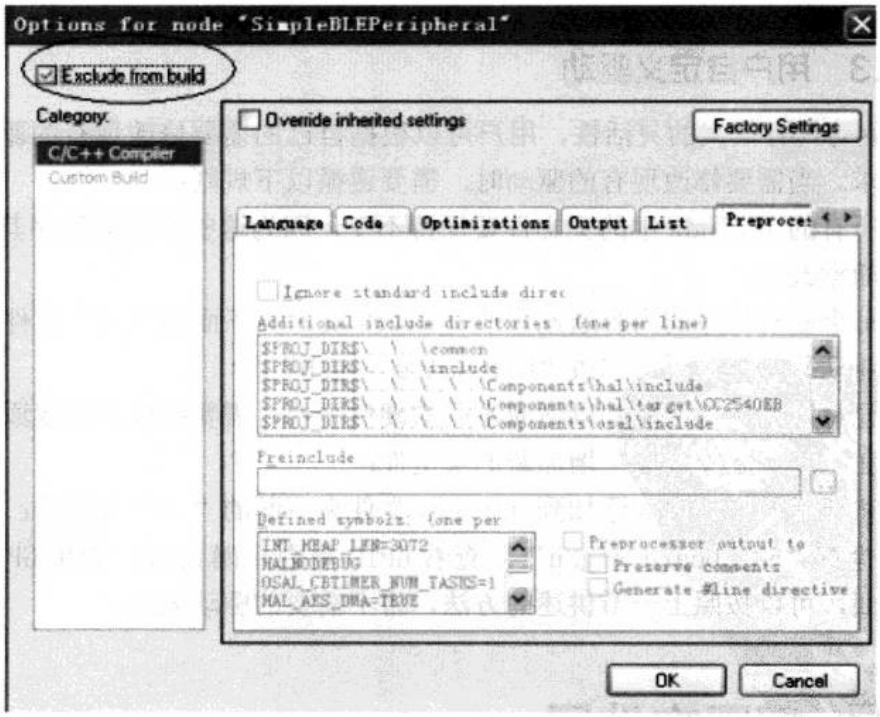


图5-13 勾选后将文件排除编译

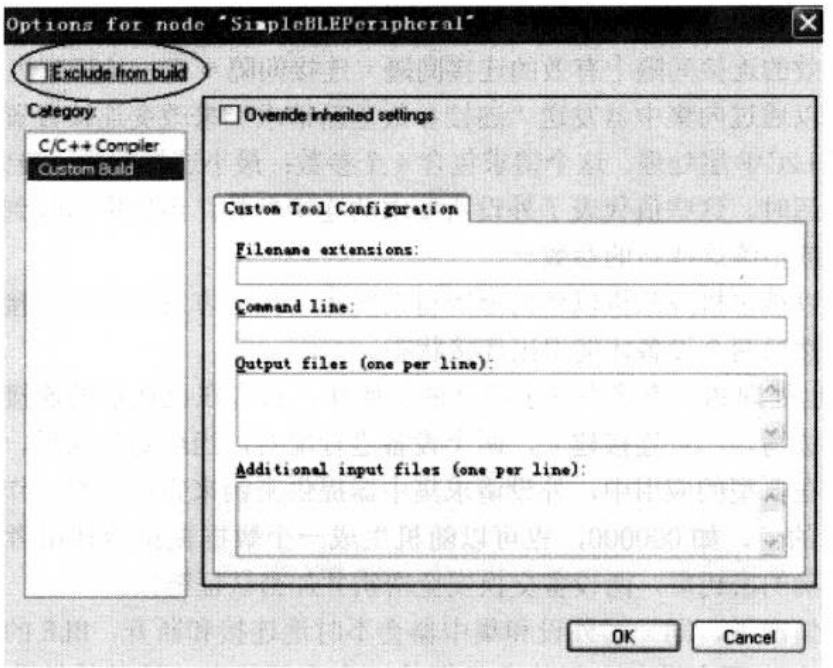


图 5-14　不勾选则将文件加入编译

在典型的蓝牙低功耗系统中，外部设备广告特定的数据，来使集中器知道它是一个可以连接的设备。广告内容包括设备地址以及一些额外的数据，如设备名等。集中器收到广告数据后，向外部设备发送扫描请求，然后外部设备将特定的数据回应给集中器，称为扫描回应。集中器收到扫描回应后，便知道这是一个可以建立连接的外部设备，这就是设备发现的全过程。此时，集中器可以向外部设备发起建立连接的请求。连接请求包括下面一些连接参数。

▲　连接间隔——在两个 BLE 设备的连接中使用跳频机制，两个设备使用特定的信道收发数据，然后过一段时间后再使用新的信道（链路层处理信道切换），两设备在信道切换后收发数据称之为连接事件。即使没有应用数据的收发，两设备仍然会通过交换链路层数据来维持连接。连接间隔就是两个连接事件之间的时间间隔，连接间隔以 1.25 为单位，连接间隔的值为 6（7.5ms）～3200（4s）。

不同的应用可能要求不同的时间间隔。长的时间间隔的优势是显著地节省功耗，因为设备可以在连接事件之间有较长时间的休眠，坏处是当设备有应用数据需要发送时，必须要等到下一个连接事件；短的时间间隔优势是两设备连接频发，可以更快地收发数据，不利之处是设备因连接事件的到来而被频繁地唤醒，会有较多功耗。

▲　从机延迟——这个参数的设置可以使从机（外部设备）跳过若干连接事件，这给了外设更多的灵活度，如果它没有数据发送时，可以选择跳过连接事件继续休眠，以节省功耗。

▲　管理超时——这是两个成功连接事件之间的最大允许间隔。如果超过了这个时间而没有成功的连接事件，设备被认为是丢失连接，返回到未连接状态。这个

值的单位是10ms，管理超时的范围是10（100ms）～3200（32s）。另外，超时值必须大于有效的连接间隔［有效的连接间隔=连接间隔*（1+从机延迟）］。

外设可以通过向集中器发送“连接参数更新请求”来改变连接设置，这个请求由协议栈的L2CAP层处理。这个请求包含4个参数：最小连接间隔、最大连接间隔、从机延迟、超时。这些值代表了外设所要求的连接参数。当集中器收到请求后，可以选择接受或拒绝这些新的参数。

连接可以被主机或从机以任何原因自动终止。当一方发起终止连接时，另一方必须响应。然后两个设备才能退出连接状态。

GAP层也处理BLE连接中安全特征的初始化。只有在已认证的连接中，特定的数据才能被读写。一旦连接建立，两个设备进行配对，当配对完成后，形成加密链接的密钥。在典型的应用中，外设请求集中器提供密钥来完成配对工作，密钥可以是一个固定的值，如000000，也可以随机生成一个数据提供给使用者，当集中器设备发送正确的密钥后，两设备交换安全密钥并加密认证链接。

在许多情况下，同一对外设和集中器会不时地连接和断开，BLE的安全机制中有一项特性允许两个设备之间建立长期的安全密钥信息，这种特性称为绑定，它允许两设备重新连接时快速地完成加密认证，而不需要每次连接时执行配对的完整过程。

5.5.2 通用属性配置文件（GATT）

两个设备应用数据的通信是通过协议栈的GATT层实现。从GATT角度来看，当两个设备建立连接后，他们处于下面两种角色之一：

▲ GATT服务器——它是为GATT客户端提供数据服务的设备；

▲ GATT客户端——它是从GATT服务器读写应用数据的的设备。

需要特别注意的是，GATT角色中的客户端和服务器的概念与链路层的主机和从机概念完全独立，与GAP角色中的外设和集中器概念也是完全独立的。主机既可以是GATT客户端也可以是GATT服务器；从机既可以是GATT客户端也可以是GATT服务器。

一个GATT服务器中可包含一个或多个GATT服务，GATT服务是完成特定功能的一系列数据的集合。在示例工程SimpleBLEPeripheral 应用中，有三个GATT服务。

▲ 强制的GAP服务——这一服务包含了设备和访问信息。例如，设备名、设备供应商和产品标识。它是协议栈的一部分，是BLE规范对每一个BLE设备的强制要求。这部分源代码并没有提供，而是编译到协议栈库文件中了。

▲ 强制的GATT服务——这一服务包含了GATT服务器的信息，是协议栈的一部分，是BLE规范对每一个BLE设备的要求。这部分源代码编译到协议栈库文件中了。

▲ SimpeGATTProfile服务——这个服务包含应用数据的信息，与应用数据的传递密切相关，读者也可以按照特定的格式编写自己的GATT服务。

“特性”(Characteristics)是服务用到的值，以及其内容和配置信息。GATT

定义了在BLE连接中发现、读取和写入属性的子过程。GATT服务器上的特性值，及其内容和配置信息（称为描述符）存储于属性表中。属性表是一个数据库，包含了称为属性的小块数据，除了值本身，每个属性都包含下列属性：

▲ 句柄——属性在表中的地址，每个属性有唯一的句柄；

▲ 类型——表示数据代表的事务，通常是蓝牙技术联盟规定或由用户自定义的UUID(Universally Unique Identifier)；

▲ 权限——规定了GATT客户端设备对属性的访问权限，包括是否能访问和怎样访问。

GATT定义了若干在GATT服务器和客户端之间的通信的子过程。

下面是一些子过程。

▲ 读特性值——客户端设备请求读取句柄处的特性值，服务器将此值回应给客户端（假定属性有读权限）。

▲ 使用特性的UUID读——客户端请求读基于一个特定类型的所有特性值，服务器将所有与指定类型匹配的特性的句柄和值回应给客户端设备（假设属性有读权限）。

▲ 读多个特性值——客户端一次请求中读取几个句柄的特性值，服务器将这些特性值回应给客户端（假设属性有读权限）。客户端需要知道如何解析这些不同特性值的数据。

▲ 读特性描述符——客户端请求读特定句柄处的特性描述符，服务器将特性描述符的值回应给客户端设备（假定属性有读权限）。

▲ 使用UUID发现特性——客户端通过发送“特性”的类型（UUID）来请求发现这个“特性”的句柄。服务器将这个“特性”的声明回应给客户端设备，其中包括特性值的句柄以及“特性”的权限。

▲ 写特性值——客户端设备请求向服务器特定的句柄处写入特性值，服务器将数据是否写入成功的信息反馈给客户端。

▲ 写特性描述符——客户端设备请求向服务器特定的句柄处写入特性描述符，服务器将特性描述符是否写入成功的信息反馈给客户端。

▲ 特性值通知——服务器将一个特性值通知给客户端。客户端设备不需要向服务器请求这个数据，当客户端收到这个数据时，也不需要回应服务器。但需要注意的是，要想使能服务器通知，首先要配置好特性，profile中定义了什么时候服务器应该发送这个数据。

每个Profile初始化其相应的服务并内在地通过设备的GATT服务器来注册服务。GATT服务器将整个服务加到属性表中，并为每个属性分配唯一句柄。

GATT属性表中有一些特殊的属性类型，其值由蓝牙技术联盟定义：

▲ GATT_PRIMARY_SERVICE_UUID——表示新服务的起始和提供的服务类型；

▲ GATT_CHARACTER_UUID——称为“特性声明”，紧随其后的是GATT特性值；

▲ GATT_CLIENT_CHAR_CFG_UUID——这一属性代表特性描述符，它与属性表

中它前面最近的句柄处的特性值相关，它允许GATT客户端设备使能特性值通知；

▲ GATT_CHAR_USER_DESC_UUID——这一属性代表特性描述符，它与属性表中它前面最近的句柄处的特性值相关，包含一个ASCII字符串，是对相关的特性的描述。

5.5.3 使用GAP和GATT的堆栈API

应用程序和配置文件直接调用GAP和GATT API功能来实现BLE相关功能，例如广告、连接、读写“特性”。

使用GAP或GATT的API功能的一般步骤如下：

▲ 使用适当的参数调用API 功能；

▲ 协议栈进行特定的操作并返回。；

▲ 操作完成后，或者堆栈有信息需要报告给调用API功能的任务，堆栈发送一条OSAL消息给调用API的任务；

▲ 调用API的任务接收消息并处理；

▲ 调用API的任务释放消息。

从GAP外设角色profile可以看到这一过程的例子（peripheral.c）如下。

▲ Profile调用GAP_DeviceInit来初始化设备。

▲ GAP进行初始化并返回SUCCESS（0x00）。

▲ 初始化完成后，BLE协议栈发送OSAL消息，头事件值为GAP_MSG_EVENT，操作码值为GAP_DEVICE_INIT_DONE_EVENT。

▲ Profile任务接收到SYS_EVENT_MSG事件，表明它有一个消息。Profile收到这个消息并查看事件头和操作码的值，profile基于这些值来将消息数据归到相应的类型并处理。在这种情况下，profile 将生成的密钥存储到flash存储器，并在本地保存设备地址。

▲ Profile释放消息并返回。

考虑到过多的文字性叙述而没有结合具体实验来讲解的话，会使读者感觉到一头雾水，本节对配置文件层仅做了概括性讲述，以使读者有一个初步认识。具体内容以及使用方法会在后续实验中陆续介绍给读者。

5.6 蓝牙4.0 BLE协议栈中串口应用详解

串口是开发板和电脑交互的一种工具，正确地使用串口对于蓝牙4.0 BLE的学习具有较大的促进作用，使用串口的基本步骤：

▲ 初始化串口，包括设置波特率、中断等；

▲ 向发送缓冲区发送数据或者从接收缓冲区读取数据。

上述方法是使用串口的常用方法，在本书的基础实验中已作了详细讲解。但是由于蓝牙 4.0 BLE 协议栈的存在，使得串口的使用略有不同，在蓝牙 4.0 BLE 协议栈中已经对串口初始化所需要的函数进行了实现，读者只需要传递几个参数就可以使用串口，此外，蓝牙 4.0 BLE 协议栈还实现了串口的读取函数和写入函数。

因此，读者在使用串口时，只需要掌握蓝牙 4.0 BLE 协议栈提供的串口操作相关的三个函数即可。蓝牙 4.0 BLE 协议栈中提供的与串口操作有关的三个函数为：

▲ uint8 HalUARTOpen(uint8 port, halUARTCfg_t *config);

▲ uint16 HalUARTRead(uint8 port, uint8 *buf, uint16 len);

▲ uint16 HalUARTWrite(uint8 port, uint8 *buf, uint16 len)。

在此先不对上述函数进行原理性的介绍[●]，先通过一个具体的例子展示一下上述函数的使用方法，或许使用过这些函数后，自然而然地就理解了。

5.6.1 串口收发实验

本节实验是建立在 SimpleBLEPeripheral 示例工程基础上，对“SimpleBLEPeripheral.c”文件进行一下改动就可以演示串口数据的收发实验。

修改“SimpleBLEPeripheral.c”文件，添加的内容如下（新增加的部分以加粗字体显示）：

```
#include "hal_uart.h"
unsigned char uartbuf[152] ;
static void rxCB(uint8 port,uint8 event)
{
1  HalUARTRead(0, uartbuf,19) ;
2  if(osal_memcmp(uartbuf,"BLE_stack_uart_test",19))
   {
3  HalUARTWrite(0, uartbuf,19) ;
4  osal_memset( uartbuf,0,19);
  }
}
```

向 SimpleBLEPeripheral_Init( ) 函数中添加：

```
5  halUARTCfg_t uartConfig;
6  uartConfig.configured      = TRUE;
7  uartConfig.baudRate      = HAL_UART_BR_115200;
8  uartConfig.flowControl   = FALSE;
9  uartConfig.callBackFunc = rxCB ;
```

● 这是笔者推荐的学习方法，即遇到新的函数、新的知识点，先尝试着去使用，或者查找相关的例子去看别人是如何使用的，这样通过实验，建立了感性的认识，然后再尝试着去看这些函数的源代码，根据实验现象去观察每个参数的具体含义。

```
10  HalUARTOpen (0, &uartConfig);
```

蓝牙4.0 BLE协议栈中对串口的配置是使用一个结构体来实现的，该结构体为halUARTCfg_t，在此不必关心该结构体的具体定义形式，只需要对其功能有个了解，该结构体将串口初始化有关的参数集合在了一起，例如：波特率、是否打开串口、是否使用流控等，读者只需要将各个参数初始化就可以。

最后使用HalUARTOpen()函数对串口进行初始化，注意，该函数将halUARTCfg_t类型的结构体变量作为参数，因为halUARTCfg_t类型的结构体变量已经包含了串口初始化相关的参数，所以，将这些参数传递给HalUARTOpen()函数，HalUARTOpen()函数使用这些参数对串口进行了初始化。

```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
uint16 events )
{
}
```

因本实验中并没有进行事件处理，SimpleBLEPeripheral_ProcessEvent（）函数中不需要任何实现代码，该函数是一个空函数。

下面进行代码分析：

第1行，调用HalUARTRead()函数，从串口读取数据并将其存放在uartbuf数组中。

第2行，使用osal_memcmp()函数判断接收到的数据是否是字符串"BLE_stack_uart_test"，如果是该字符串，在osal_memcmp()函数返回TURE，则执行第3行。注意，osal_memcmp()函数经常会使用到，读者需要掌握这一函数的使用技巧。

第3行，调用HalUARTWrite()函数将接收到的字符输出到串口。

第4行，将发送缓冲区中的内容清空，防止串口不停的收到数据。

第5～8行，配置串口参数。

第9行，获取回调函数的值。

第10行，打开串口。

rxCB()函数是一个回调函数，什么是回调函数呢？

回调函数就是一个通过函数指针（函数地址）调用的函数。如果把函数的指针（也即函数的地址）作为参数传递给另一个函数，当通过这个指针调用它所指向的函数时，称为函数的回调。

在第9行代码处，将rxCB()传递给了uartConfig的成员函数callBackFunc，其中callBackFunc的定义为：

```
halUARTCBack_t        callBackFunc;
```

而halUARTCBack_t的定义为：

```
typedef void (*halUARTCBack_t) (uint8 port, uint8 event);
```

这就是定义了一个函数指针！

小技巧 ▶▶▶

部分读者可能对函数指针的定义形式不熟悉，可以尝试着下面的方式理解。常用的定义形式如下：

typedef uint unsigned int

则如下两种定义变量的方式等价；

unit num ;

unsigned int num ;

按照这种理解方式，或许函数指针的定义形式改为如下形式更好理解：

typedef halUARTCBack_t void (*) (uint8 port, uint8 event);

当然这只是帮助读者理解的一种方式而已。

因此，第 9 行代码处，将 rxCB() 传递给了 uartConfig 变量的 callBackFunc 成员函数，这样就实现了“把函数的指针（也即函数的地址）作为参数传递给另一个函数”，这样就可以通过 callBackFunc 成员函数来调用 rxCB() 函数了。

此外，回调函数不是由该函数的实现方直接调用的，而是在特定的事件或条件发生时，由另外的一方调用的，用于对该事件或条件进行响应。

回调函数机制提供了系统对异步事件的处理的能力。首先将异步事件发生时需要执行的代码编写成一个函数，并将该函数注册成为回调函数，这样当该异步事件发生时，系统会自动调用事先注册好的回调函数，回调函数的注册实际上就是将回调函数的信息填写到一个用于注册回调函数的结构体变量中。

在程序中使用回调函数有以下几个步骤：

▲ 定义一个回调函数；

▲ 在初始化时，提供函数实现的一方将回调函数的函数指针传递给调用者；

▲ 当特定的事件或条件发生的时候，调用者使用函数指针调用回调函数对事件进行处理。

回调函数，顾名思义需要调用者对函数进行回调，到底什么时候回调的呢？

先把此问题放一下，先看一下上述代码的执行情况，然后再对回调进行讲解。顺便说一句，只要将函数的回调机制理解清楚，可以说蓝牙 4.0 BLE 协议栈的开发将变得简单了，因为串口操作有回调函数，按键操作也有回调函数，GATT 数据服务也有回调函数……

5.6.2 实例测试

将程序编译后，下载到 CC2540 开发板，设置串口调试助手，如图 5-15 所示。

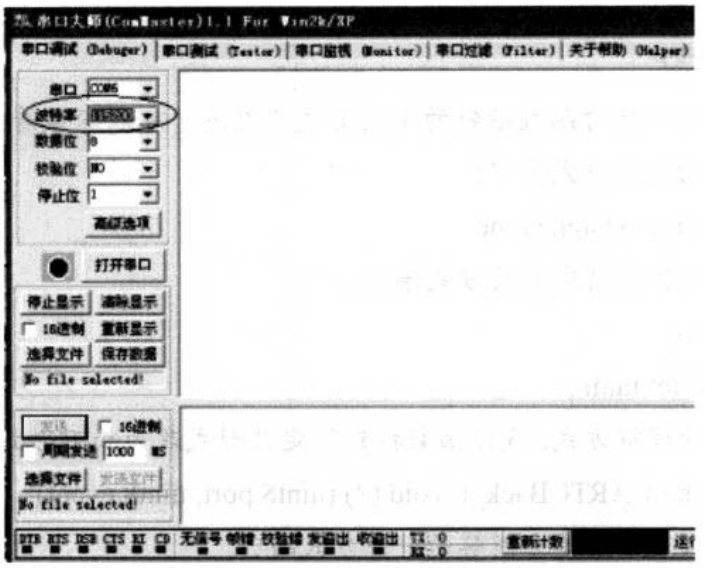


图 5-15 设置串口调试助手

串口调试助手主界面如图5-16所示，在输入栏中输入字符串“BLE_stack_uart_test”，单击“发送”按钮，咦？接收栏并没有显示任何字符！

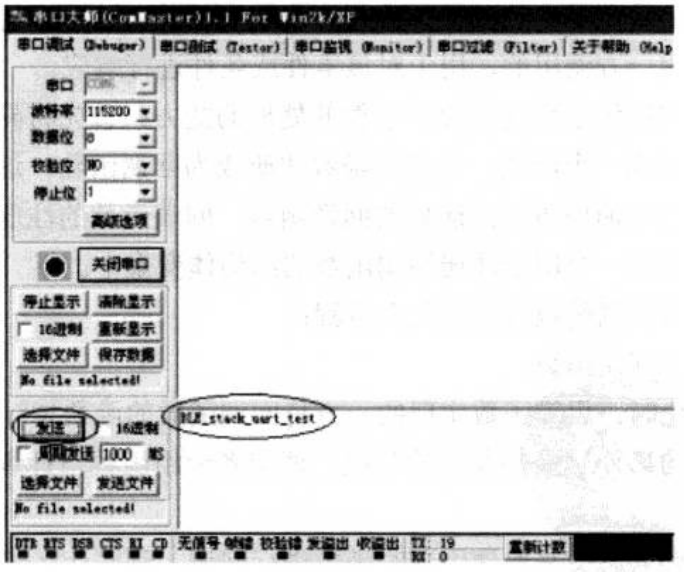


图 5-16 串口调试助手主界面

什么原因呢？程序有问题吗？经过前文的讲解，程序应该是没有问题的，读者通过串口输入数据后，读取串口的数据，然后将其发送到PC机的串口，那为什么接收不到数据呢？

这是由于蓝牙4.0 BLE协议栈使用了条件编译，打开“SimpleBLEPeripheral_Main.c”文件，找到main()函数，main()函数原型如下：

```
int main(void)
{
  HAL_BOARD_INIT();
  InitBoard( OB_COLD );
  HalDriverInit();
  osal_snv_init();
  osal_init_system();
  HAL_ENABLE_INTERRUPTS();
  InitBoard( OB_READY );
  #if defined ( POWER_SAVING )
    osal_pwrmgr_device( PWRMGR_BATTERY );
  #endif
  osal_start_system();
  return 0;
}
```

在main()函数中可以找到对HalDriverInit()函数的调用，右键单击HalDriverInit()函数，在弹出的下拉菜单中选择“Go to definition of HalDriverInit”，如图5-17所示，即可跳转到HalDriverInit()函数的定义处。

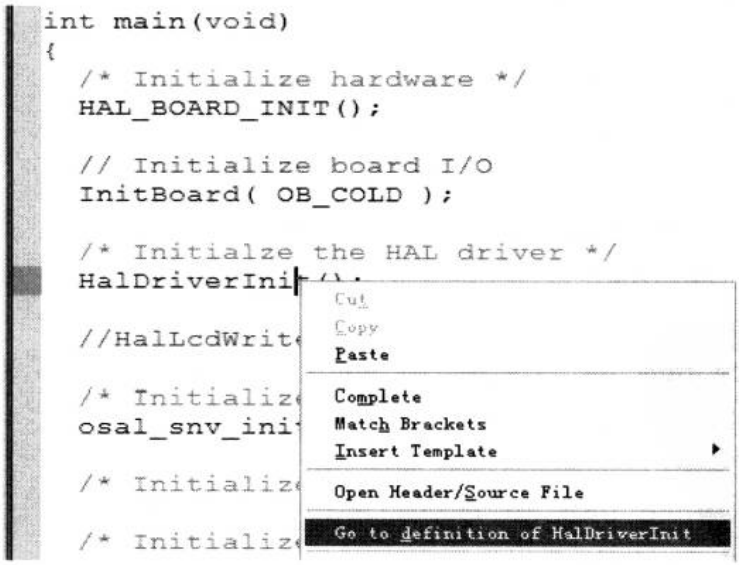


图 5-17　在弹出的下拉菜单中选择“Go to definition of HalDriverInit”

HalDriverInit()函数的定义如下：

```
void HalDriverInit (void)
{
  #if (defined HAL_TIMER) && (HAL_TIMER == TRUE)
```

```
    HalTimerInit();
  #endif
  #if (defined HAL_ADC) && (HAL_ADC == TRUE)
    HalAdcInit();
  #endif
  #if (defined HAL_DMA) && (HAL_DMA == TRUE)
    HalDmaInit();
  #endif
  #if (defined HAL_AES) && (HAL_AES == TRUE)
    HalAesInit();
  #endif
  #if (defined HAL_LCD) && (HAL_LCD == TRUE)
    HalLcdInit();
  #endif
  #if (defined HAL_LED) && (HAL_LED == TRUE)
    HalLedInit();
  #endif
  #if (defined HAL_UART) && (HAL_UART == TRUE)
    HalUARTInit();
  #endif
  #if (defined HAL_KEY) && (HAL_KEY == TRUE)
    HalKeyInit();
  #endif
  #if (defined HAL_IR) && (HAL_IR == TRUE)
    HalIrInit();
  #endif
  #if (defined HAL_DS18B20) && (HAL_DS18B20 == TRUE)
    HalDS18B20Init();
  #endif
  #if (defined HAL_SPI) && (HAL_SPI == TRUE)
    HalSpiInit();
  #endif
  #if (defined HAL_HID) && (HAL_HID == TRUE)
    usbHidInit();
  #endif
}
```

可见，使用UART时需要定义HAL_UART宏，并且将其值赋值为TRUE，在IAR

开发环境中，可以使用如下方法打开对 UART 的宏定义。

在 SimpleBLEPeripheral 工程上右键单击，在弹出的下拉菜单中选择“Options”如图 5-18 所示。

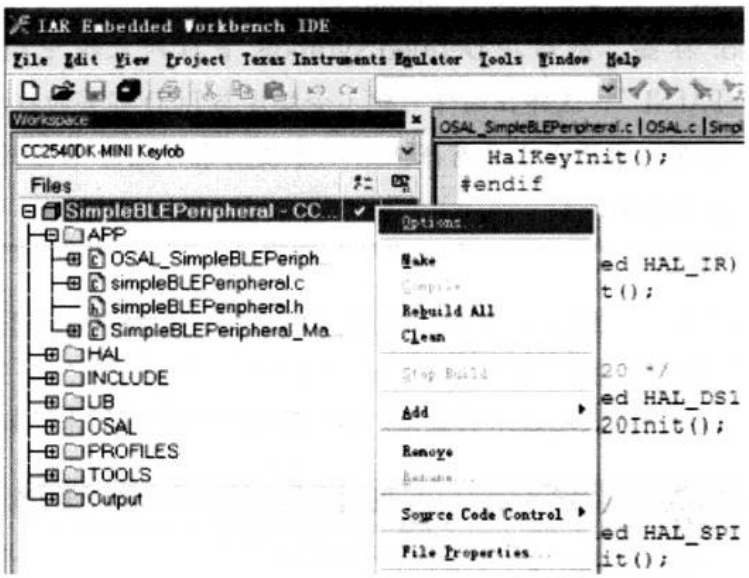


图 5-18　在弹出的下拉菜单中选择“Options”

此时会弹出 Options for node“SimpleBLEPeripheral”主窗口，如图 5-19 所示。

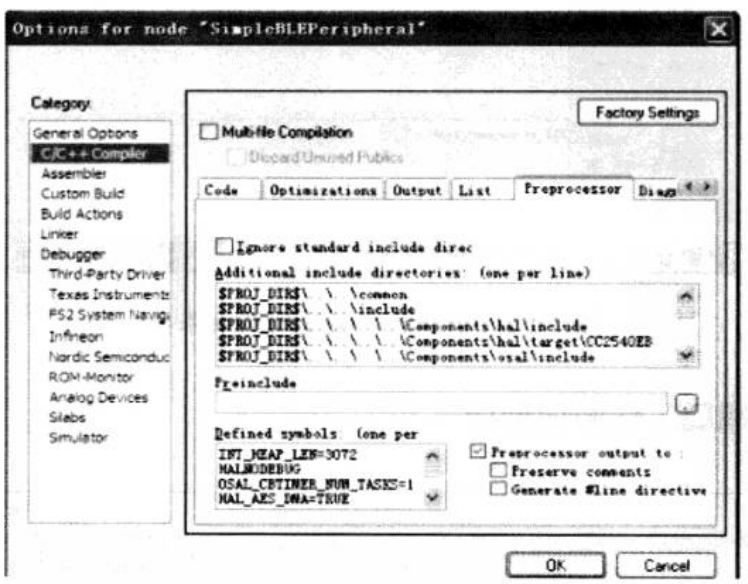


图 5-19　弹出 Options for node “SimpleBLEPeripheral”窗口

选择 C/C++ Compiler 标签，在窗口右边选择 Preprocessor 标签，然后在 Defined symbols 下拉列表框中输入“HAL_UART=TRUE”，最后单击 OK 按钮即可。

> **注意 ▶▶▶**
>
> 上述方法适用于其他模块，如LCD模块，如果不需要LCD显示数据，则可以选择C/C++ Compiler标签，在窗口右边选择Preprocessor标签，然后在Defined symbols下拉列表框中输入“HAL_LCD=FALSE”，这样在编译时就不会编译与LCD有关的程序。因为单片机的存储器资源十分有限，所以才使用条件编译来控制不同的模块是否参与编译。

此时，将程序编译下载到CC2540开发板，按照前文讲解的方法正确设置串口调试助手，从发送栏输入字符串“BLE_stack_uart_test”，然后单击“发送”按钮，此时，在接收栏接收到了开发板发送过来的数据，串口收发实验测试效果图如图5-20所示。

图5-20 串口收发实验测试效果图

5.6.3 串口工作原理剖析

> **注意 ▶▶▶**
>
> 本节内容读者可以有选择的阅读，内容涉及到DMA部分的知识。

上述实验只是向读者展示了在蓝牙4.0 BLE协议栈中串口的应用（或者说串口的配置），但是还有一些细节的问题没有讲解（虽然初学时可以将这些问题忽略，

也需要理解这些问题的工作原理），例如下述几个问题：

▲ 在蓝牙 4.0 BLE 协议栈中，halUARTCfg_t 结构体是如何定义的；

▲ 串口是如何初始化的；

▲ 发送给串口的数据是如何接收的；

▲ 串口是如何向 PC 机发送数据的。

这些问题涉及到上述实验中使用到的三个函数：HalUARTOpen()、HalUARTRead()、HalUARTWrite()，将这三个函数理解透彻，蓝牙 4.0 BLE 协议栈中的串口应用将很简单。

首先需要理解的问题是串口是什么时候被初始化的。下面看一下 HalUARTOpen() 函数做了哪些事情，HalUARTOpen() 函数原型如下：

```
uint8 HalUARTOpen(uint8 port, halUARTCfg_t *config)
{
#if (HAL_UART_DMA == 1)
  if (port == HAL_UART_PORT_0)  HalUARTOpenDMA(config);
#endif
#if (HAL_UART_DMA == 2)
  if (port == HAL_UART_PORT_1)  HalUARTOpenDMA(config);
#endif
#if (HAL_UART_ISR == 1)
  if (port == HAL_UART_PORT_0)  HalUARTOpenISR(config);
#endif
#if (HAL_UART_ISR == 2)
  if (port == HAL_UART_PORT_1)  HalUARTOpenISR(config);
#endif
#if (HAL_UART_USB)
  HalUARTOpenUSB(config);
#endif
#if (HAL_UART_DMA == 0) && (HAL_UART_ISR == 0)
  (void) port;
  (void) config;
#endif
  return HAL_UART_SUCCESS;
}
```

加粗部分实际上是调用了 HalUARTOpenDMA 函数，HalUARTOpenDMA 函数原型如下（省略了部分无关的代码）：

```
static void HalUARTOpenDMA(halUARTCfg_t *config)
{
```

```
  dmaCfg.uartCB = config->callBackFunc;
  ……
  if (config->baudRate == HAL_UART_BR_57600 ||
    config->baudRate == HAL_UART_BR_115200)
{
  UxBAUD = 216;
}
else
{
  UxBAUD = 59;
}
switch (config->baudRate)
{
  case HAL_UART_BR_9600:
    UxGCR = 8;
    dmaCfg.txTick = 35;
    break;
  case HAL_UART_BR_19200:
    UxGCR = 9;
    dmaCfg.txTick = 18;
    break;
  case HAL_UART_BR_38400:
    UxGCR = 10;
    dmaCfg.txTick = 9;
    break;
  case HAL_UART_BR_57600:
    UxGCR = 10;
    dmaCfg.txTick = 6;
    break;
  default:      //串口默认波特率是115200
    UxGCR = 11;
    dmaCfg.txTick = 3;
    break;
}
……
dmaCfg.rxBuf[0] = *(volatile uint8 *)DMA_UDBUF;
HAL_DMA_CLEAR_IRQ(HAL_DMA_CH_RX);
HAL_DMA_ARM_CH(HAL_DMA_CH_RX);
```

```
osal_memset(dmaCfg.rxBuf, (DMA_PAD ^ 0xFF),HAL_UART_DMA_
RX_MAX*2);
.....
}
```

需要注意的是，在蓝牙 4.0 BLE 协议栈中，TI 采用的方法是将串口和 DMA 结合起来使用，这样可以减轻 CPU 的负担。

该函数有个 halUARTCfg_t 类型的参数，halUARTCfg_t 的定义如下：

```
typedef struct
{
  bool                  configured;
  uint8                 baudRate;
  bool                  flowControl;
  uint16                flowControlThreshold;
  uint8                 idleTimeout;
  halUARTBufControl_t   rx;
  halUARTBufControl_t   tx;
  bool                  intEnable;
  uint32                rxChRvdTime;
  halUARTCBack_t     callBackFunc;
}halUARTCfg_t;
```

其中，halUARTCBack_t 为：

```
typedef void (*halUARTCBack_t) (uint8 port, uint8 event);
```

这显然是一个函数指针。

halUARTCfg_t 结构体较为复杂，一般不需要使用串口的硬件流控，所以很多与流控相关的不需要关注，一般的应用，只需要关注加粗字体部分的三个参数即可。

在 HalUARTOpenDMA() 函数中对串口的波特率进行了初始化，同时对 DMA 接收缓冲区进行了初始化。

在波特率初始化过程中，UxBAUD 和 UxGCR 的值可以从 CC2540 数据手册中查找到对应的初始化值，常用波特率设置（系统时钟为 32MHz）如表 5-4 所示。

表 5-4　常用波特率设置（系统时钟为 32MHz）

波特率	UxBAUD	UxGCR
9600	59	8
19200	59	9
38400	59	10
57600	216	10
115200	216	11

根据halUARTCfg_t结构体中的成员变量baudRate在初始化时设定的波特率，参考表5-4中UxBAUD和UxGCR的值，使用switch-case语句就可以完成串口波特率的初始化。

当然在程序的最后还完成了DMA的部分初始化工作，正因为有了TI的蓝牙4.0 BLE协议栈将串口和DMA结合在了一起，所以使读者理解起来较为困难，下面对DMA的实现机理进行讲解。

在蓝牙4.0 BLE协议栈中，开辟了DMA发送缓冲区和接收缓冲区，通过串口调试助手向串口发送数据时，数据首先存放在DMA接收缓冲区，然后调用HalUARTRead()函数进行读取时，实际上是读取的DMA缓冲区中的数据，HalUARTRead()函数原型如下：

```
uint16 HalUARTRead(uint8 port, uint8 *buf, uint16 len)
{
    (void)port;
    (void)buf;
    (void)len;
    #if (HAL_UART_DMA == 1)
        if (port == HAL_UART_PORT_0)   return
HalUARTReadDMA(buf,len);
    #endif
    #if (HAL_UART_DMA == 2)
        if (port == HAL_UART_PORT_1)   return
HalUARTReadDMA(buf, len);
    #endif
    #if (HAL_UART_ISR == 1)
        if (port == HAL_UART_PORT_0)   return
HalUARTReadISR(buf, len);
    #endif
    #if (HAL_UART_ISR == 2)
        if (port == HAL_UART_PORT_1)   return
HalUARTReadISR(buf, len);
    #endif
    #if HAL_UART_USB
       return HalUARTRx(buf, len);
    #else
       return 0;
    #endif
}
```

该函数实际上是调用了 HalUARTReadDMA() 函数，如加粗字体部分所示。

当调用 HalUARTWrite() 函数发送数据时，实际上是将数据写入到了 DMA 发送缓冲区，然后 DMA 自动将发送缓冲区中的数据通过串口发送给 PC 机，HalUARTWrite() 函数原型如下：

```
uint16 HalUARTWrite(uint8 port, uint8 *buf, uint16 len)
{
(void)port;
(void)buf;
(void)len;
#if (HAL_UART_DMA == 1)
if (port == HAL_UART_PORT_0)  return HalUARTWriteDMA(buf,
len);
#endif
#if (HAL_UART_DMA == 2)
if (port == HAL_UART_PORT_1)  return HalUARTWriteDMA(buf,
len);
#endif
#if (HAL_UART_ISR == 1)
if (port == HAL_UART_PORT_0)  return HalUARTWriteISR(buf,
len);
#endif
#if (HAL_UART_ISR == 2)
if (port == HAL_UART_PORT_1)  return HalUARTWriteISR(buf,
len);
#endif
#if HAL_UART_USB
HalUARTTx(buf, len);
return len;
#else
return 0;
#endif
}
```

该函数实际上是调用了 HalUARTWriteDMA() 函数，如加粗字体部分所示。

注意 ▶▶▶

上述只是从原理上对串口DMA机制进行的分析，具体的实现细节请读者参考协议栈源代码进行理解，在使用DMA时开辟了接收缓冲区和发送缓冲区，因此还涉及到数据缓冲区数据满、缓冲区数据空、缓冲区是否有新数据等检测，读者可以从这些方面去理解代码的实现细节。

5.7 本章小结

本章讲述了蓝牙4.0 BLE协议栈的分层结构，并重点对操作系统抽象层、硬件抽象层、配置文件层进行了讲解，同时对蓝牙4.0 BLE协议栈中的串口配置以及使用方法进行了初步的讨论，读者可以自行修改工程文件的内容以满足自身项目的需求。

[扩展阅读] OSI七层模型及协议栈分层思想

OSI（Open System Interconnection）七层网络模型称为开放式系统互联参考模型，是一个逻辑上的规范。它把网络从逻辑上分为七层，是一种框架性的设计方法，通过七个层次化的结构模型使不同的系统、不同的网络之间实现可靠的通信。OSI七层模型参考如图5-21所示。

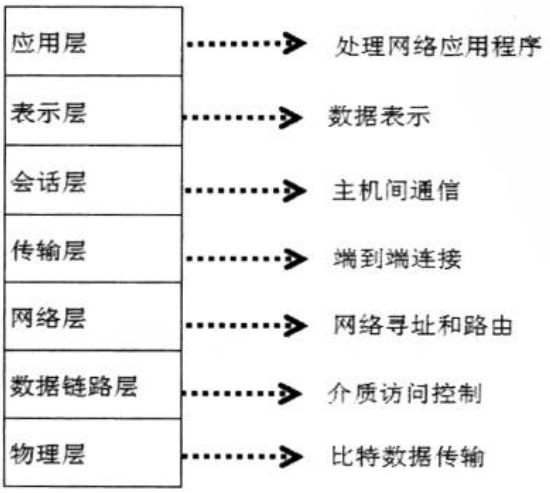


图5-21 OSI七层参考模型

OSI 七层参考模型，从最底层到最上层分别为物理层、数据链路层、网络层、传输层、会话层、表示层以及应用层。

▲ 物理层——负责原始比特流的传输，电子信号传输和硬件接口；

▲ 数据链路层——负责保障信息可靠地在物理链路上传输，控制网络层与物理层之间的通信。与这层相关的有物理地址、网络拓扑结构、网络存取、错误通报、数据包顺序、流量控制；

▲ 网络层——主要功能是将网络地址翻译成对应的物理地址，并决定如何将数据从发送方路由到接收方，提供了两个网络系统之间的连接和路径选择；

▲ 传输层——负责流量控制并基于接收方可接收数据的快慢程度规定适当的发送速率，根据网络处理能力把要传输的信息分成小的分段，把收到的分段整合成原信息；

▲ 会话层——负责建立、管理、结束两个网络节点之间的通信。会话层为表示层提供服务，它也负责同步两个网络节点的表示层和管理它们的信息交换；

表示层——充当应用程序和网络之间的翻译。在表示层，数据将按照网络能理解的格式进行转换；

▲ 应用层——最接近用户的一层，与其他六层的不同是，它不提供服务给另一层，而是为用户应用程序提供网络服务。

OSI 参考模型的分层思想是为了使网络的不同分层负责不同的功能，数据只在相邻的分层之间流动，从而带来如下便利：

▲ 有效降低了网络复杂程度，一旦系统发生故障，可以迅速定位故障所处的分层，便于查找和纠错；

▲ 在各层分别定义标准接口，使处于对等分层的不同网络设备能实现交互操作，各层之间则相对独立，一种高层协议可放在多种低层协议上运行；

▲ 能有效推动网络技术的革新，因为每次更新都可以只在单独的分层进行，不需对整个网络进行重新设计。

蓝牙 4.0 BLE 协议栈也是在 OSI 参考模型的基础上，结合无线网络的特点，采用分层的思想实现的。蓝牙 4.0 BLE 协议栈分层如图 5-22 所示。

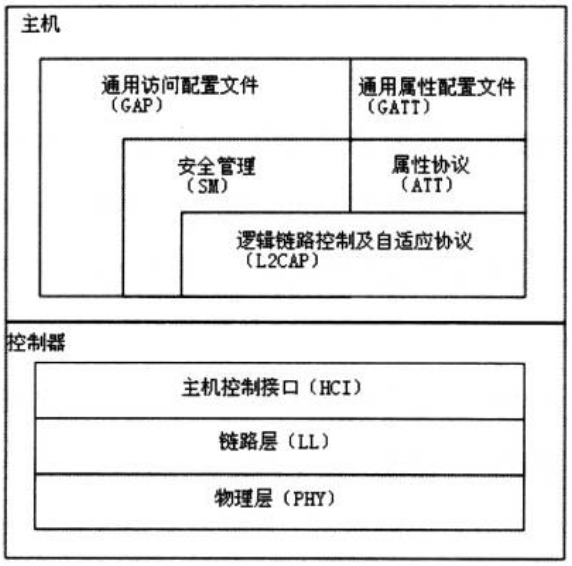


图 5-22　BLE 协议栈分层

蓝牙 4.0 BLE 协议栈采用分层思想的最大优点是：将服务、接口和协议这三个概念明确地区分开来。服务说明某一层为上一层提供一些什么样的功能；接口说明上一层如何使用下层的服务；而协议涉及到如何实现本层的服务。这样各层之间具有很强的独立性，当协议的一部分发生变化时，只需对与此相关的分层进行修改即可，其他分层不需要改变。

第6章

蓝牙4.0 BLE无线网络技术

前一章着重讨论了蓝牙 4.0 BLE 协议栈的构成以及协议栈各层的工作机理，读者应该已经对蓝牙 4.0 BLE 协议栈有了初步的认识，本章将通过几个具体的示例工程对蓝牙 4.0 BLE 应用程序的开发展开进一步讲解。

6.1 蓝牙无线网络节点设备示例工程

这个示例工程实现了一个简单 BLE 外设功能，使用单一设备堆栈配置构建，协议栈、Profile、应用程序都运行在 CC2540 设备上面。关于单一设备堆栈配置请读者参考第 5.2.2 节讲述的 BLE 协议栈开发平台配置。

6.1.1 工程概述

在 IAR 窗口左边，“workspace”区域列出了工程中用到的所有文件，如图 6-1 所示。

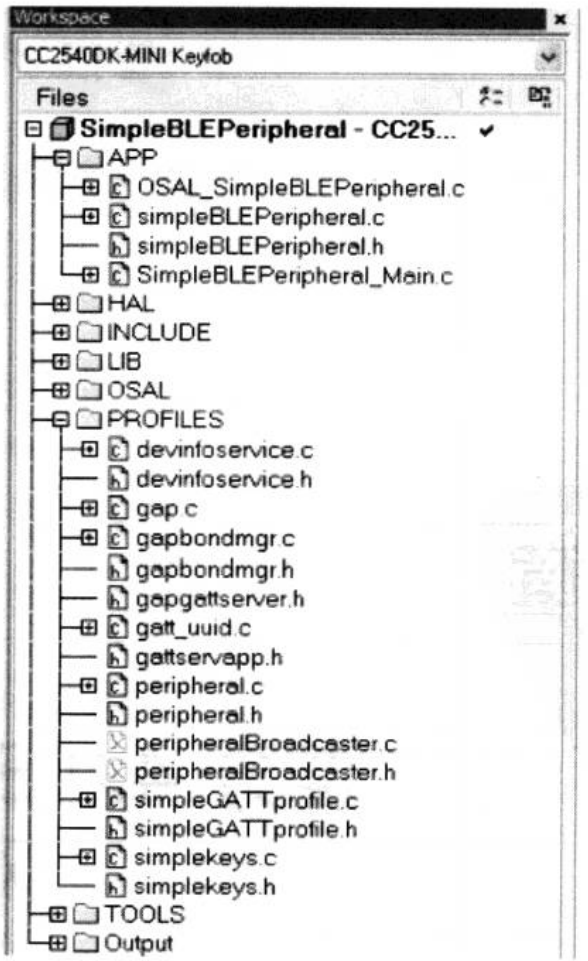


图 6-1 工程文件

文件列表分成如下几个文件夹：

▲ APP——包含应用程序源代码和头文件；

▲ HAL——包含硬件抽象层源代码和头文件；

▲ INCLUDE——这个组包含所有的 BLE 协议栈 API 的头文件；

▲ LIB——协议栈库文件；

▲ OSAL——包含操作系统抽象层源代码和头文件；

▲ PROFILE——包含 GAP 角色 Profile、GAP 安全 Profile、GATT Profile 的源代码和头文件；

▲ TOOLS——包含 buildConfig.cfg、buildComponents.cfg。也包含"OnBoard.c"和"OnBoard.h"，处理用户接口功能；

▲ OUTPUT——IAR 输出的内容。

6.1.2 初始化

应用程序初始化分为两个阶段：首先，OSAL 调用 SimpleBLEPeripheral_Init 函数，这一初始化函数设置 GAP 任务配置文件参数、GAP 特性、GAP 绑定管理参数、GATT 配置文件参数，通过设置一个操作系统虚拟层的 SBP_START_DEVIC_EVT 事件[●]来触发第二阶段的初始化，调用 SimpleBLEPeripheral_ProcessEvent（）事件处理函数来处理第二阶段的初始化工作。

```
void SimpleBLEPeripheral_Init( uint8 task_id )
{
......
osal_set_event( simpleBLEPeripheral_TaskID, SBP_START_
DEVICE_EVT );
}
```

如上程序所示，第一阶段的初始化的最后使用 osal_set_event(uint8 task_id, uint16 event_flag) 来设置了 SBP_START_DEVICE_EVT 事件，来触发第二阶段的初始化工作。

SimpleBLEPeripheral_ProcessEvent（）事件处理函数如下：

```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
uint16 events )
{
  VOID task_id;
  if ( events & SYS_EVENT_MSG )
  {
```

● 本书第 5 章详细讲解了操作系统虚拟层处理任务事件的机制。

```
        uint8 *pMsg;
        if ( (pMsg = osal_msg_receive( simpleBLEPeripheral_
TaskID )) != NULL )
        {
          simpleBLEPeripheral_ProcessOSALMsg( (osal_event_hdr_
t *)pMsg );
          VOID osal_msg_deallocate( pMsg );
        }
        return (events ^ SYS_EVENT_MSG);
      }
      1  if ( events & SBP_START_DEVICE_EVT )
      {
      2   VOID GAPRole_StartDevice( &simpleBLEPeripheral_
PeripheralCBs );
      3   VOID GAPBondMgr_Register( &simpleBLEPeripheral_
BondMgrCBs );
      4  osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_
PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
      5  return ( events ^ SBP_START_DEVICE_EVT );
      }
      ……
    return 0;
    }
```

第1行，因为初始化的第一步设置了一个操作系统虚拟层的SBP_START_DEVICE_EVT事件，在事件处理函数中检测是否有启动设备（SBP_START_DEVICE_EVT）事件发生。

第2行，启动设备，设置应用的GAP功能。

第3行，启动绑定管理。

第4行，设置了产生第一次周期事件的定时时间，定时函数原型参考本书第5.3.5章节。

第5行，将已处理的事件标志位清零，未处理的事件标志返回。

初始化完成后，设备处于可发现状态，未指定广告接收者。此时，集中器设备可以通过扫描来发现这个外设设备。如果集中器设备向这个外设发送连接请求，则外设接受请求并作为从机进入连接状态。外设发送广告并处于可发现状态时，有两种模式：一种是受限的发现模式；另一种是不受限制的发现模式。

其中，受限制的发现模式指的是外设在发送广告时，如果没收到集中器设备发来的建立连接请求，则只保持30s的可发现状态，然后转为不可被发现的待机状态。

而不受限制的发现模式则是外设在没有收到集中器设备的连接请求时，一直发送广告，处于可被发现的状态。

那么，外设的两种发现模式怎样选择呢？在示例工程的“SimpleBLEPeripheral.c”文件中有如下代码：

```
#if defined ( CC2540_MINIDK )
#define DEFAULT_DISCOVERABLE_MODE    GAP_ADTYPE_FLAGS_LIMITED
#else
#define DEFAULT_DISCOVERABLE_MODE    GAP_ADTYPE_FLAGS_GENERAL
#endif
```

GAP_ADTYPE_FLAGS_LIMITED 为受限的可发现模式；

GAP_ADTYPE_FLAGS_GENERAL 为不受限的可发现模式；

当程序中有宏定义 CC2540_MINIDK 时，将设备设置为受限的可发现模式，反之，则为不受限的可发现模式。在预编译选项中输入宏 CC2540_MINIDK，如图 6-2 所示。

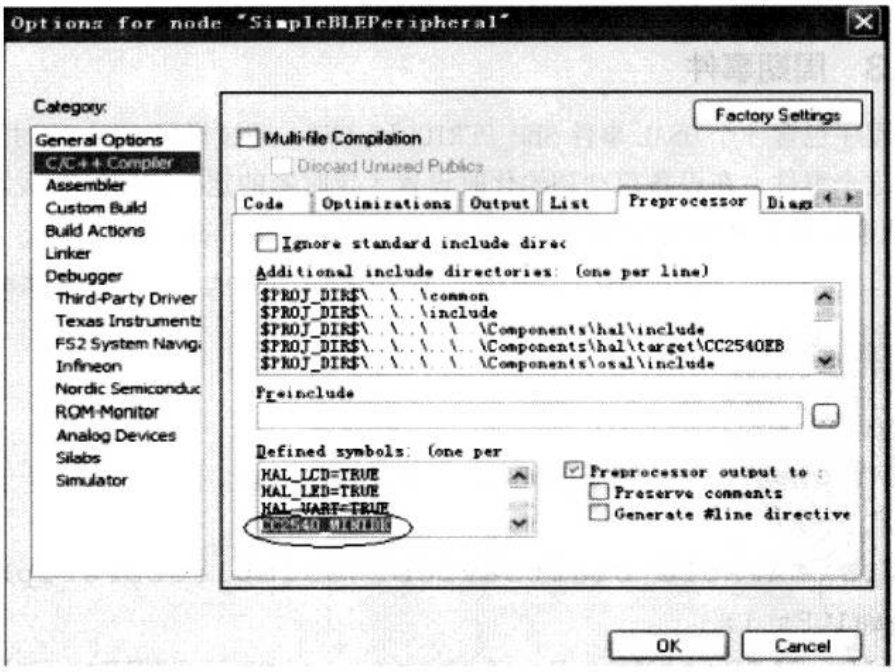


图 6-2　预编译选项中设置为受限的可发现模式

向预编译选项中输入宏 xCC2540_MINIDK，即设为不受限的可发现模式，如图 6-3 所示。

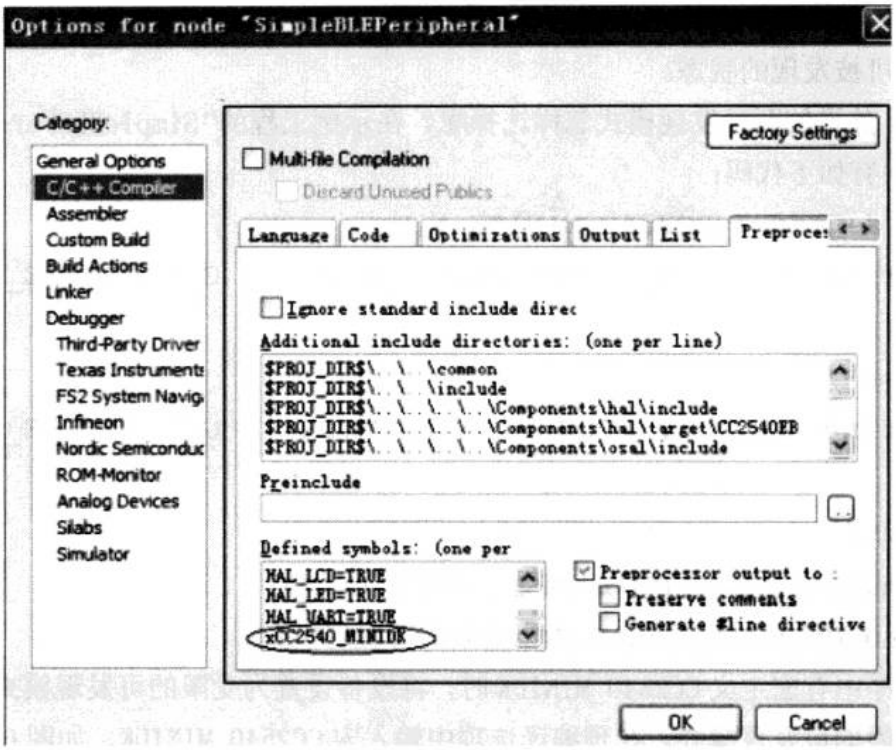


图 6-3　预编译选项中设置为不受限的可发现模式

6.1.3　周期事件

应用程序包含一个 OSAL 事件 SBP_PERIODIC_EVT，通过设置 OSAL 定时器，周期性地发生这个事件。在设备启动初始化时设置了定时器的定时时间（默认为 5s），每 5s 周期事件发生并调用周期任务处理函数 performPeriodicTask（）。

```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
uint16 events )
{
  ......
  if ( events & SBP_START_DEVICE_EVT )
  {
  VOID GAPRole_StartDevice( &simpleBLEPeripheral_
PeripheralCBs );
  VOID GAPBondMgr_Register( &simpleBLEPeripheral_
BondMgrCBs );
1 osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_
PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
  return ( events ^ SBP_START_DEVICE_EVT );
```

```
    }
2   if ( events & SBP_PERIODIC_EVT )
    {
3    if ( SBP_PERIODIC_EVT_PERIOD )
     {
4     osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_
PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
     }
5    performPeriodicTask();
6    return (events ^ SBP_PERIODIC_EVT);
    }
    ......
     return 0;
}
```

第 1 行，设备初始化事件处理过程中设置了周期事件的第一次定时时间。

第 2 ～ 4 行，周期事件发生时，首先重新设定下一次周期事件的定时时间。

第 5 行，调用周期事件的处理函数，即周期性地执行某项任务，具体执行的任务用户可以自己定义。

第 6 行，返回未被处理的事件。

下面来看一下示例工程中周期任务performPeriodicTask中要执行的具体任务。

```
static void performPeriodicTask( void )
{
  uint8 valueToCopy;
  uint8 stat;
7  stat = SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR3,
&valueToCopy);
     if( stat == SUCCESS )
     {
8    SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR4,
sizeof(uint8),&valueToCopy);
     }
}
```

第 7 行，复制 profile 中第三个“特性”的值。

第 8 行，将 profile 中第三个“特性”的值存到第四个“特性”中。

在这里读者只需了解应用数据是存放在“特性”之中，应用数据的读写都是通过“特性”来完成。

6.1.4 外设状态通知回调

应用程序中包含了称之为外设状态通知回调的函数(peripheralStateNotificationCB)，这个函数在调用GAPRole_StartDevice之后被注册。之后的任何时候当外设GAP角色改变时，回调函数被调用，更新应用的局部变量gapProfileState。回调函数允许应用进行基于设备状态的操作，相应的操作在回调函数中执行。

加粗部分中对回调函数进行了注册：

```
if ( events & SBP_START_DEVICE_EVT )
  {
    VOID GAPRole_StartDevice( &simpleBLEPeripheral_
PeripheralCBs );
    VOID GAPBondMgr_Register( &simpleBLEPeripheral_
BondMgrCBs );
      osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_
PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
    return ( events ^ SBP_START_DEVICE_EVT );
  }
```

下面以设备状态回调函数为例进行分析：

```
static gapRolesCBs_t simpleBLEPeripheral_PeripheralCBs =
{
  peripheralStateNotificationCB,
  NULL
};
```

外设状态回调函数执行内容如下代码所示：

```
static void peripheralStateNotificationCB( gaprole_States_
t newState )
{
  switch ( newState )
  {
    case GAPROLE_STARTED:
      {
        uint8 ownAddress[B_ADDR_LEN];
        uint8 systemId[DEVINFO_SYSTEM_ID_LEN];
        GAPRole_GetParameter(GAPROLE_BD_ADDR, ownAddress);
        ......
        #if (defined HAL_LCD) && (HAL_LCD == TRUE)
          HalLcdWriteString( bdAddr2Str( ownAddress ),
```

```
HAL_LCD_LINE_2 );
            HalLcdWriteString( "Initialized",   HAL_LCD_
LINE_3 );
        #endif
      }
      break;
    case GAPROLE_ADVERTISING:
      {
        #if (defined HAL_LCD) && (HAL_LCD == TRUE)
        HalLcdWriteString( "Advertising",  HAL_LCD_LINE_3 );
        #endif
      }
      break;
    case GAPROLE_CONNECTED:
      {
        #if (defined HAL_LCD) && (HAL_LCD == TRUE)
          HalLcdWriteString( "Connected",  HAL_LCD_LINE_3 );
        #endif
      }
      break;
    case GAPROLE_WAITING:
      {
        #if (defined HAL_LCD) && (HAL_LCD == TRUE)
          HalLcdWriteString( "Disconnected",  HAL_LCD_LINE_3 );
        #endif
      }
      break;
    case GAPROLE_WAITING_AFTER_TIMEOUT:
      {
        #if (defined HAL_LCD) && (HAL_LCD == TRUE)
          HalLcdWriteString( "Timed Out",  HAL_LCD_LINE_3 );
        #endif
      }
          break;
      case GAPROLE_ERROR:
      {
        #if (defined HAL_LCD) && (HAL_LCD == TRUE)
```

```
        HalLcdWriteString( "Error",  HAL_LCD_LINE_3 );
      #endif
    }
    break;
  default:
    {
      #if (defined HAL_LCD) && (HAL_LCD == TRUE)
        HalLcdWriteString( "",  HAL_LCD_LINE_3 );
      #endif
    }
    break;
  }
  gapProfileState = newState;
#if !defined( CC2540_MINIDK )
  VOID gapProfileState;
#endif
}
```

针对外设的不同状态，液晶上面显示不同的字符串来表示当前所处的状态。

外设不同状态的赋值如下：

```
typedef enum
{
  GAPROLE_INIT = 0,
  GAPROLE_STARTED,
  GAPROLE_ADVERTISING
  GAPROLE_WAITING
  GAPROLE_WAITING_AFTER_TIMEOUT
  GAPROLE_CONNECTED,
  GAPROLE_ERROR
} gaprole_States_t;
```

enum 为枚举类型，定义了第一个值为 0，从第二个数据开始，值依次加 1。

6.1.5 按键服务

示例工程的应用程序中包含按键相关的代码，需要按照前面章节所述的方法在预编译内容中定义一个宏“CC2540_MINIDK”。这给 GATT 服务器增加了简单的按键服务，用来处理用户定义的按键事件。每一次按键按下或释放，HAL 层发送一个 OSAL 消息给应用，产生一个 SYS_EVENT_MSG 事件，通过 simpleBLEperipheral_ProcessOSALMsg 来处理这个消息事件。

```
static void simpleBLEPeripheral_ProcessOSALMsg( osal_
event_hdr_t *pMsg )
{
  switch ( pMsg->event )
  {
  #if defined( CC2540_MINIDK )
    case KEY_CHANGE:
      simpleBLEPeripheral_HandleKeys( ((keyChange_t *)
pMsg)->state, ((keyChange_t*)pMsg)->keys );
    break;
  #endif
  default:
    break;
  }
}
```

若收到的 OSAL 消息为按键事件，则调用按键处理函数如下：

```
static void simpleBLEPeripheral_HandleKeys( uint8 shift,
uint8 keys )
{
  uint8 SK_Keys = 0;
  VOID shift;
  if ( keys & HAL_KEY_SW_1 )
  {
    SK_Keys |= SK_KEY_LEFT;
  }
  if ( keys & HAL_KEY_SW_2 )
  {
    SK_Keys |= SK_KEY_RIGHT;
  1  if( gapProfileState != GAPROLE_CONNECTED )
    {
      uint8 current_adv_enabled_status;
      uint8 new_adv_enabled_status;
   2  GAPRole_GetParameter(GAPROLE_ADVERT_
ENABLED,&current_adv_enabled_status );
  3  if( current_adv_enabled_status == FALSE )
  {
  new_adv_enabled_status = TRUE;
```

```
    }
  else
    {
    4  new_adv_enabled_status = FALSE;
    }
    5  GAPRole_SetParameter(GAPROLE_ADVERT_
ENABLED,sizeof(uint8),&new_adv_enabled_status );
    }
  }
    6  SK_SetParameter(SK_KEY_ATTR,sizeof(uint8),&SK_Keys );
  }
```

第1行，判断设备gap状态是否为连接状态，如果未连接，则执行相应的处理程序。

第2行，读取设备当前的广告状态。

第3行，如果设备当前的广告状态为FALSE，即设备未广告时，参数更改为设备开启广告。

第4行，如果设备当前的广告状态为TRUE，即设备广告时，参数更改为设备关闭广告，即每一次按键按下，都会改变设备的广告状态。

第5行，将新的广告状态写入设备。

第6行，将按键状态值写入按键Profile的“特性”中，如果设备处于连接状态并且使能了按键通知，改变后的按键值将会通知给集中器设备。关于通知功能的使用将在第7章讲解。

6.1.6 LCD显示

应用程序也包含了LCD的代码，只需要在预编译中设置“HAL_LCD=TRUE”即可使用，如图6-4所示。

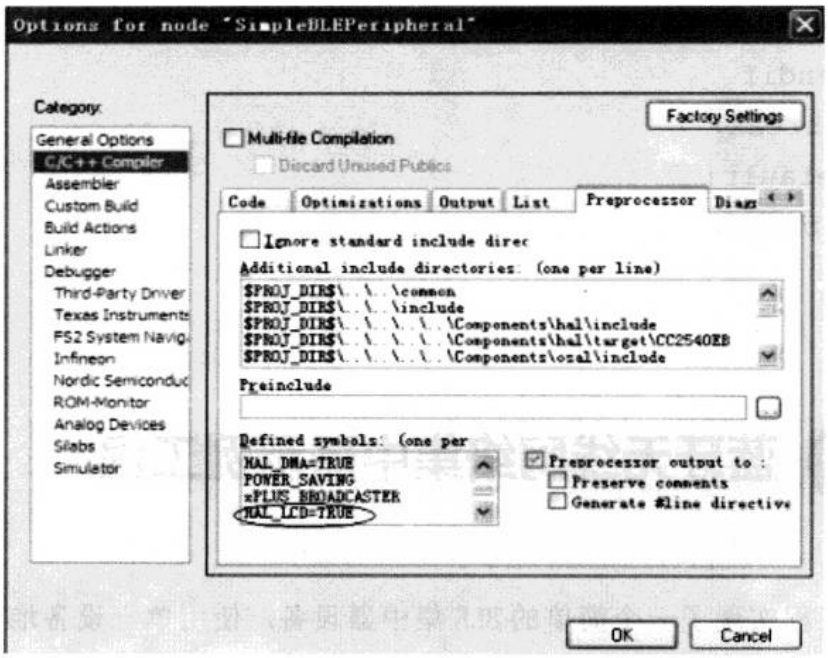


图 6-4 设置 LCD 预编译项

LCD 可用来显示设备地址、设备 GAP 连接状态以及特性值等信息。下面程序为特性值改变时的回调函数，加粗部分为 LCD 显示“特性”中的值。

```
static void simpleProfileChangeCB( uint8 paramID )
{
  uint8 newValue;
  switch( paramID )
  {
  case SIMPLEPROFILE_CHAR1:
    SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR1,
&newValue );
    #if (defined HAL_LCD) && (HAL_LCD == TRUE)
     HalLcdWriteStringValue("Char1:",(uint16)
(newValue),10, HAL_LCD_LINE_3 );
    #endif
     break;
    case SIMPLEPROFILE_CHAR3:
     SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR3,
&newValue );
    #if (defined HAL_LCD) && (HAL_LCD == TRUE)
     HalLcdWriteStringValue("Char3:",(uint16)
```

```
(newValue),10, HAL_LCD_LINE_3 );
        #endif
          break;
        default:
          break;
      }
    }
```

6.2 蓝牙无线网络集中器示例工程

这个工程实现了一个简单的BLE集中器设备，使用单一设备堆栈配置，协议栈、Profile、应用程序都运行在CC2540设备上面。关于单一设备堆栈配置请读者参考第5.2.2节。使用SimpleBLECentral工程的集中器设备可以与SimpleBLEPeripherial设备建立连接，来演示协议栈基本的GAP和GATT工作过程。

6.2.1 工程概述

SimpleBLECentral工程结构非常类似于SimpleBLEPeripherial工程，如图6-5所示。

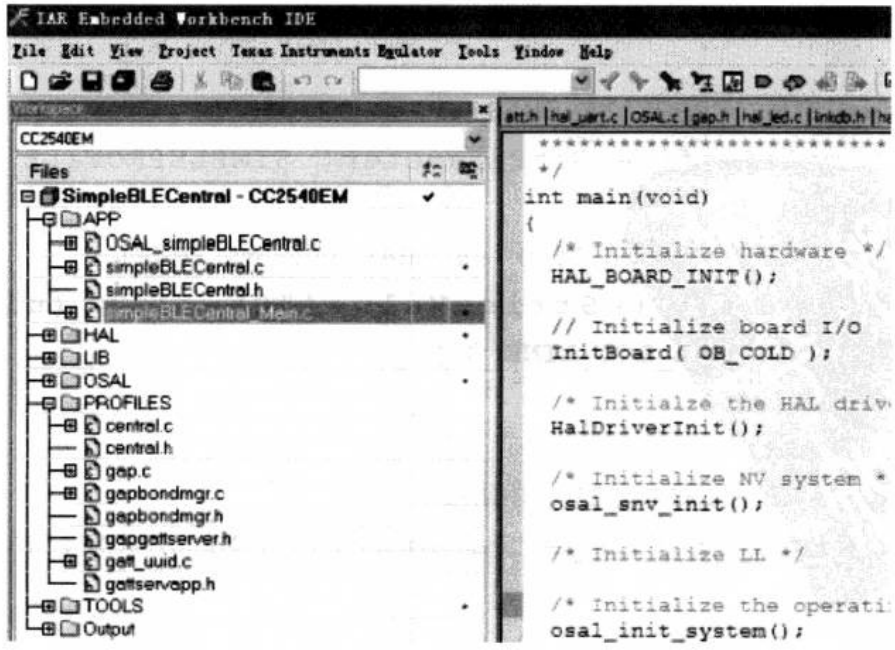


图6-5 SimpleBLECentral工程主界面

6.2.2 初始化

应用程序初始化分为两个阶段：首先，OSAL 调用 SimpleBLECentral_Init 函数，这一初始化函数配置集中器的参数、GAP 特性、GAP 绑定管理参数、初始化 GATT 配置文件，在属性服务器中设置标准的 GAP 和 GATT 服务。然后设置一个操作系统虚拟层的事件 START_DEVIC_EVT 来触发第二阶段的初始化，调用 SimpleBLECentral_ProcessEvent 事件处理函数来处理第二阶段的初始化工作，这一阶段使用 GAP_CentralRole_StartDevice 函数来设置应用的 GAP 功能，然后调用 GAPBondMgr_Register 函数来注册绑定管理。

SimpleBLECentral_Init 初始化函数如下：

```
void SimpleBLECentral_Init( uint8 task_id )
{
   simpleBLETaskId = task_id;
1  uint8 scanRes = DEFAULT_MAX_SCAN_RES;
2  GAPCentralRole_SetParameter ( GAPCENTRALROLE_MAX_SCAN_
RES, sizeof( uint8 ), &scanRes );
3  GAP_SetParamValue( TGAP_GEN_DISC_SCAN, DEFAULT_SCAN_
DURATION );
4  GAP_SetParamValue( TGAP_LIM_DISC_SCAN, DEFAULT_SCAN_
DURATION );
5  GGS_SetParameter( GGS_DEVICE_NAME_ATT, GAP_DEVICE_
NAME_LEN, (uint8 *) simpleBLEDeviceName );
6  uint32 passkey = DEFAULT_PASSCODE;
7  uint8 pairMode = DEFAULT_PAIRING_MODE;
8  uint8 mitm = DEFAULT_MITM_MODE;
9  uint8 ioCap = DEFAULT_IO_CAPABILITIES;
10  uint8 bonding = DEFAULT_BONDING_MODE;
11  GAPBondMgr_SetParameter( GAPBOND_DEFAULT_PASSCODE,
sizeof( uint32 ), &passkey );
12  GAPBondMgr_SetParameter( GAPBOND_PAIRING_MODE,
sizeof( uint8 ), &pairMode );
13  GAPBondMgr_SetParameter( GAPBOND_MITM_PROTECTION,
sizeof( uint8 ), &mitm );
14  GAPBondMgr_SetParameter( GAPBOND_IO_CAPABILITIES,
sizeof( uint8 ), &ioCap );
15  GAPBondMgr_SetParameter( GAPBOND_BONDING_ENABLED,
sizeof( uint8 ), &bonding );
```

```
16  VOID GATT_InitClient();
17  GATT_RegisterForInd( simpleBLETaskId );
18  GGS_AddService( GATT_ALL_SERVICES );
19  GATTServApp_AddService( GATT_ALL_SERVICES );
20  RegisterForKeys( simpleBLETaskId );
21  HalLedSet((HAL_LED_1 | HAL_LED_2), HAL_LED_MODE_OFF );
22  osal_set_event( simpleBLETaskId, START_DEVICE_EVT );
}
```

第1、2行，设置集中器Profile，设置扫描可发现的最大设备数量。

第3～5行，设置GAP，包括普通扫描模式和受限扫描模式的最小持续时间等参数。

第6～15行，设置GAP绑定管理。包括初始密钥、绑定模式、配对时的绑定请求等参数。

第16行，初始化GATT客户端。

第17行，注册来接收传入的属性指示(Indications)或通知(Notifications)。

第18、19行，初始化GATT属性。

第20行，注册所有的按键事件。

第21行，熄灭所有LED灯。

第22行，设置一个OSAL事件START_DEVICE_EVT。

设置OSAL事件的函数如下所示：

```
uint8 osal_set_event( uint8 task_id, uint16 event_flag )
{
  if ( task_id < tasksCnt )
  {
    halIntState_t   intState;
  1 HAL_ENTER_CRITICAL_SECTION(intState);
  2 tasksEvents[task_id] |= event_flag;
  3 HAL_EXIT_CRITICAL_SECTION(intState);
    return ( SUCCESS );
  }
  else
  {
    return ( INVALID_TASK );
  }
}
```

第1行，关闭所有中断，避免同时有其他任务和中断服务进入临界段代码。

第2行，采用“按位或”的方式将事件标志位置1。

第 3 行，开启中断。

6.2.3 事件处理

应用程序有两个主要的事件处理函数， SimpleBLECentral_ProcessEvent 和 leBLECentral_ProcessOSALMsg。

SimpleBLECentral_ProcessEvent 函数处理如下事件：

▲ SYS_EVENT_MSG：为 OSAL 队列服务，处理 OSAL 消息；

▲ START_DEVICE_EVT：启动设备；

▲ START_DISCOVERY_EVT：启动设备服务发现；

▲ simpleBLECentral_ProcessOSALMsg 函数处理如下两种 OSAL 消息：

▲ KEY_CHANGE 消息：调用按键处理函数 simpleBLECentral_HandleKeys 来按键任务；

▲ GATT_MSG_EVENT 消息：调用消息处理函数 simpleBLECentralProcessGATT 来处理来自 GATT 的消息。

OSAL 消息 START_DEVIC_EVT 触发了第二阶段的初始化，调用 leBLECentral_ProcessEvent 事件处理函数如下：

```
uint16 SimpleBLECentral_ProcessEvent( uint8 task_id,
:16 events )
{
  VOID task_id;
1  if ( events & SYS_EVENT_MSG )
  {
2  uint8 *pMsg;
3  if ( (pMsg = osal_msg_receive( simpleBLETaskId )) !=
; )
  {
4  simpleBLECentral_ProcessOSALMsg( (osal_event_hdr_t *)
g );
5  VOID osal_msg_deallocate( pMsg );
  }
6  return (events ^ SYS_EVENT_MSG);
  }
  if ( events & START_DEVICE_EVT )
  {
7  VOID GAPCentralRole_StartDevice( (gapCentralRoleCB_t *)
mpleBLERoleCB );
8  GAPBondMgr_Register( (gapBondCBs_t *) &simpleBLEBondCB );
```

```
9  return ( events ^ START_DEVICE_EVT );
   }
10  if ( events & START_DISCOVERY_EVT )
   {
     simpleBLECentralStartDiscovery( );
     return ( events ^ START_DISCOVERY_EVT );
   }
   return 0;
}
```

第1～6行，用来处理OSAL的按键事件和GATT消息，后面再详细讨论。

第7行，启动设备事件发生时，注册GAP状态回调函数，每当GAP状态改变时，调用回调函数进行相应的操作。

第8行，启动设备事件发生时，注册绑定管理回调函数，用于处理绑定管理相关的操作。

第9行，返回未处理的事件标志，至此，设备初始化工作完成。

第10行，处理设备服务发现事件，集中器设备通过GATT服务UUID来发现外设的数据服务。

设备初始化完成后，按下开发板上Key1键，集中器扫描是否有处于广告状态的外设。程序执行流程如下：按键按下后，HAL层发送一个OSAL消息给应用，产生一个SYS_EVENT_MSG事件，通过simpleBLECentral_ProcessOSALMsg来处理这个消息事件。如果为按键事件，则调用按键处理函数simpleBLECentral_HandleKeys来处理按键任务。

simpleBLECentral_ProcessOSALMsg消息处理函数如下：

```
static void simpleBLECentral_ProcessOSALMsg( osal_event_
hdr_t *pMsg )
{
  switch ( pMsg->event )
  {
    case KEY_CHANGE:
      simpleBLECentral_HandleKeys( ((keyChange_t *)pMsg)-
>state, ((keyChange_t*)pMsg)->keys );
    break;
  case GATT_MSG_EVENT:
    simpleBLECentralProcessGATTMsg( (gattMsgEvent_t *)
pMsg );
    break;
  }
```

```
}
```

前文中提到过 simpleBLECentral_ProcessOSALMsg 用来处理按键消息和 GATT 数据服务的消息。当消息为按键事件时，调用按键处理函数如下（省略部分无关代码）：

```
static void simpleBLECentral_HandleKeys( uint8 shift, uint8 keys )
{
  (void)shift;
1 if ( keys & HAL_KEY_SW_1 )
  {
2 if ( simpleBLEState != BLE_STATE_CONNECTED )
   {
3 if ( !simpleBLEScanning )
   {
4   simpleBLEScanning = TRUE;
5     simpleBLEScanRes = 0;
6   GAPCentralRole_StartDiscovery( DEFAULT_DISCOVERY_MODE, DEFAULT_DISCOVERY_ACTIVE_SCAN, DEFAULT_DISCOVERY_WHITE_LIST );
7   LCD_WRITE_STRING( "Discovering...", HAL_LCD_LINE_1 );
8   LCD_WRITE_STRING( "", HAL_LCD_LINE_2 );
   }
   else
   {
9   GAPCentralRole_CancelDiscovery();
   }
  }
  ......
}
  ......
}
```

第 1、2 行，Key1 键按下后，判断当前设备是否处于连接状态，如果未连接，则执行 if 里面的程序。

第 3 ～ 6 行，若集中器设备当前没有进行外设扫描，则调用外设扫描处理函数，启动扫描。

第 7、8 行，LCD 在对应的行显示“Discovering”字符串。

第 9 行，若集中器设备当前正在进行外设扫描，则停止扫描。

外设扫描处理函数如下：

```
bStatus_t GAPCentralRole_StartDiscovery( uint8 mode,
uint8 activeScan, uint8 whiteList )
{
1  gapDevDiscReq_t params;
2  params.taskID = gapCentralRoleTaskId;
3  params.mode = mode;
4  params.activeScan = activeScan;
5  params.whiteList = whiteList;
6  return GAP_DeviceDiscoveryRequest( &params );
}
```

第1行，定义params为结构体类型，gapDevDiscReq_t结构体的定义如下：

```
typedef struct
{
  uint8 taskID;      // 请求应用的任务ID，用来返回结果
  uint8 mode;      // 扫描的模式
  uint8 activeScan;      // 主动扫描
  uint8 whiteList;      // 白名单
} gapDevDiscReq_t;
```

第2～5行，将任务ID、扫描模式、主动扫描、白名单信息赋值给params中的各参数。

第6行，集中器发起扫描请求，函数的源码封装在协议栈的库中，仅以API形式给出。

集中器发起扫描，GAP状态改变，调用GAP事件回调函数simpleBLECentralEventCB如下：

```
static void simpleBLECentralEventCB( gapCentralRoleEvent_
t *pEvent )
{
  switch ( pEvent->gap.opcode )
  {
  ......
  case GAP_DEVICE_INFO_EVENT:
    {
    1  if ( DEFAULT_DEV_DISC_BY_SVC_UUID == TRUE )
      {
    2  if ( simpleBLEFindSvcUuid( GAP_SERVICE_UUID,
      pEvent->deviceInfo.pEvtData, pEvent->deviceInfo.
```

```
dataLen ) )
          {
   3      simpleBLEAddDeviceInfo( pEvent->deviceInfo.addr,
   pEvent->deviceInfo.addrType );
          }
        }
      }
      break;
      ......
    }
  }
```

第 1、2 行，扫描时通过指定的 GAP 服务 UUID 来过滤 UUID 不符合的广告设备。

第 3 行，向设备发现列表中添加一个设备。

在设备发现过程中，当一个设备被发现时，产生一个 GAP_DEVICE_INFO_EVENT 事件， 事件发送一个 OSAL 消息，包含了外设扫描回应的信息如下：

```
typedef struct
{
  osal_event_hdr_t  hdr;      // GAP_MSG_EVENT 和状态
  uint8 opcode;      // 操作码 GAP_DEVICE_INFO_EVENT
  uint8 eventType;      // 设备广告类型
  uint8 addrType;      // 地址类型
  uint8 addr[B_ADDR_LEN];    // 外设广告中的或扫描返回的外设地址
  int8 rssi;      // 接收的信号强度指示，用来判断链路质量
  uint8 dataLen;      // 数据长度
  uint8 *pEvtData;      // 外设广告中的或扫描返回的数据段
} gapDeviceInfoEvent_t;
```

设备扫描完成时，GAP 状态回调函数进行相应处理如下（部分无关代码省略）：

```
static void simpleBLECentralEventCB( gapCentralRoleEvent_
t *pEvent )
{
  switch ( pEvent->gap.opcode )
  {
    ......
  case GAP_DEVICE_DISCOVERY_EVENT:
    {
      simpleBLEScanning = FALSE;
1   LCD_WRITE_STRING_VALUE( "Devices Found",
```

```
simpleBLEScanRes, 10,
   HAL_LCD_LINE_1 );
   2  if ( simpleBLEScanRes > 0 )
     {
   3   LCD_WRITE_STRING( "<- To Select", HAL_LCD_LINE_2 );
     }
   4  simpleBLEScanIdx = simpleBLEScanRes;
   }
   break;
      ......
     }
   }
```

第1行，扫描完所有设备后，LCD第一行显示字符串“Devices Found”，以及扫描到的广告设备的个数，如果没有扫描到外设，则显示“Devices Found 0”。

第2、3行，LCD第二行显示字符串“<- To Select”，提示用户选择想要与之建立连接的设备。

第4行，初始化扫描索引到下一个设备。

```
typedef struct
{
  osal_event_hdr_t  hdr;      // GAP_MSG_EVENT 和状态
  uint8 opcode;      //操作码 GAP_DEVICE_DISCOVERY_EVENT
  uint8 numDevs;      //扫描期间发现的设备
  gapDevRec_t *pDevList;      //设备记录的数组，包括事件类型、地址类型和设备地址
} gapDevDiscEvent_t;
```

设备扫描完成后，集中器设备的LCD上显示了扫描到的广告设备(外设)的个数，并提示用户选择想要与之建立连接的外设。按下Key2键，集中器发起建立连接请求，程序执行过程如下所示：

```
uint16 SimpleBLECentral_ProcessEvent( uint8 task_id, uint16 events )
{
1 if ( events & SYS_EVENT_MSG )
  {
2   uint8 *pMsg;
3   if ( (pMsg = osal_msg_receive( simpleBLETaskId )) != NULL )
      {
```

```
4       simpleBLECentral_ProcessOSALMsg( (osal_event_hdr_t
*)pMsg );
5     VOID osal_msg_deallocate( pMsg );
    }
6   return (events ^ SYS_EVENT_MSG);
  }
......
}
```

第 1 ～ 4 行，按键按下后，HAL 层产生一个 OSAL 消息，调用消息事件处理函数 simpleBLECentral_ProcessOSALMsg 进行处理。

第 5 行，释放 OSAL 消息。

第 6 行，将已处理的事件标志位清零，返回未处理的事件标志。

消息事件处理函数 simpleBLECentral_ProcessOSALMsg 结构如下：

```
static void simpleBLECentral_ProcessOSALMsg( osal_event_
hdr_t *pMsg )
{
  switch ( pMsg->event )
  {
    case KEY_CHANGE:
      simpleBLECentral_HandleKeys( ((keyChange_t *)pMsg)-
>state, ((keyChange_t*)pMsg)->keys );
      break;
    case GATT_MSG_EVENT:
      simpleBLECentralProcessGATTMsg( (gattMsgEvent_t *)
pMsg );
      break;
  }
}
```

检测到为按键事件，调用按键事件处理函数 simpleBLECentral_HandleKeys 如下：

```
static void simpleBLECentral_HandleKeys( uint8 shift,
uint8 keys )
{
if ( keys & HAL_KEY_SW_2  )
  {
1  if ( !simpleBLEScanning && simpleBLEScanRes > 0 )
    {
```

```
2      simpleBLEScanIdx++;
3       if ( simpleBLEScanIdx >= simpleBLEScanRes )
       {
4         simpleBLEScanIdx = 0;
       }
5       LCD_WRITE_STRING_VALUE( "Device", simpleBLEScanIdx
+ 1, 10,HAL_LCD_LINE_1 );
6  LCD_WRITE_STRING( bdAddr2Str( simpleBLEDevList[simpleB
LEScanIdx].addr ),HAL_LCD_LINE_2 );
      }
    }
7  if ( keys & HAL_KEY_SW_2 )
    {
    uint8 addrType;
    uint8 *peerAddr;
8  if ( simpleBLEState == BLE_STATE_IDLE )
     {
9      if ( simpleBLEScanRes > 0 )
       {
10     peerAddr = simpleBLEDevList[simpleBLEScanIdx].addr;
11       addrType = simpleBLEDevList[simpleBLEScanIdx].
addrType;
12     simpleBLEState = BLE_STATE_CONNECTING;
13     GAPCentralRole_EstablishLink( DEFAULT_LINK_HIGH_
DUTY_CYCLE,DEFAULT_LINK_WHITE_LIST, addrType, peerAddr );
14    LCD_WRITE_STRING( "Connecting", HAL_LCD_LINE_1 );
15     LCD_WRITE_STRING( bdAddr2Str( peerAddr ), HAL_LCD_
LINE_2 );
      }
    }
16     else if ( simpleBLEState == BLE_STATE_CONNECTING
||simpleBLEState ==BLE_STATE_CONNECTED )
{
17  simpleBLEState = BLE_STATE_DISCONNECTING;
18  gStatus = GAPCentralRole_TerminateLink(simpleBLEConn
Handle);
19  LCD_WRITE_STRING( "Disconnecting", HAL_LCD_LINE_1 );
```

```
      }
    }
  }
```

第 1 ～ 6 行，显示扫描结果和设备地址。

第 7 ～ 12 行，如果当前状态为未连接状态，并且已经扫描到了广告设备，选定要连接的设备的设备地址和地址类型。

第 13 行，集中器向外设发起建立连接的请求。

第 14、15 行，集中器设备的 LCD 第一行显示字符串“Connecting”，表示正在发起连接请求，LCD 第二行显示外设的设备地址。

第 16 ～ 18 行，如果当前状态为已连接状态，则断开设备间连接。

第 19 行，集中器 LCD 显示字符串“Disconnecting”，表示正在断开设备连接。

连接请求函数如下：

```
bStatus_t GAPCentralRole_EstablishLink( uint8 highDutyCycle, uint8 whiteList,
uint8 addrTypePeer, uint8 *peerAddr )
{
  gapEstLinkReq_t params;
  params.taskID = gapCentralRoleTaskId;
  params.highDutyCycle = highDutyCycle;
  params.whiteList = whiteList;
  params.addrTypePeer = addrTypePeer;
  VOID osal_memcpy( params.peerAddr, peerAddr, B_ADDR_LEN );
  return GAP_EstablishLinkReq( &params );
}
```

其中，参数 Params 为结构体类型，gapEstLinkReq_t 结构体定义如下：

```
typedef struct
{
  uint8 taskID;        // 请求应用的任务 ID
  uint8 highDutyCycle;        // 高性能循环扫描，默认为禁用
  uint8 whiteList;        // 使用白名单
  uint8 addrTypePeer;        // 广告设备的地址类型
  uint8 peerAddr[B_ADDR_LEN];        // 广告设备的地址
} gapEstLinkReq_t;
```

发起建立连接的请求时，GAP 状态发生了改变，调用 GAP 状态回调函数进行相应处理如下（省略部分无关代码）：

```
static void simpleBLECentralEventCB( gapCentralRoleEvent_t *pEvent )
```

```
{
  switch ( pEvent->gap.opcode )
  {
    ......
case GAP_LINK_ESTABLISHED_EVENT:
    {
1     if ( pEvent->gap.hdr.status == SUCCESS )
      {
2       simpleBLEState = BLE_STATE_CONNECTED;
3       simpleBLEConnHandle = pEvent->linkCmpl.
connectionHandle;
4       simpleBLEProcedureInProgress = TRUE;
5       if ( simpleBLECharHdl == 0 )
        {
6       osal_start_timerEx( simpleBLETaskId, START_
DISCOVERY_EVT,DEFAULT_SVC_DISCOVERY_DELAY );
        }
7       LCD_WRITE_STRING( "Connected", HAL_LCD_LINE_1 );
8       LCD_WRITE_STRING( bdAddr2Str( pEvent->linkCmpl.
devAddr ),HAL_LCD_LINE_2 );
      }
9     else
      {
10      simpleBLEState = BLE_STATE_IDLE;
11      simpleBLEConnHandle = GAP_CONNHANDLE_INIT;
12      simpleBLERssi = FALSE;
13      simpleBLEDiscState = BLE_DISC_STATE_IDLE;
14      LCD_WRITE_STRING( "Connect Failed", HAL_LCD_
LINE_1 );
15      LCD_WRITE_STRING_VALUE( "Reason:", pEvent->gap.
hdr.status, 10,HAL_LCD_LINE_2 );
        }
      }
      break;
}
```

第1～4行，事件状态为SUCESS，将设备连接状态标志设为BLE_STATE_CONNECTED，处理进程标志设为TRUE，表示处理正在进行。

第 5 行，要读取的句柄为 0，表示还没有进行 GATT 服务发现。

第 6 行，使用 OSAL 定时器设置一个定时事件 START_DISCOVERY_EVT，即为服务发现事件。定时时间到达后，调用事件处理函数来处理这个事件。

第 7、8 行，集中器设备 LCD 显示字符串“Connected”以及所连接设备的地址。

第 9 ～ 15 行，若建立连接失败，将设备当前的发现状态设置为空闲，并在 LCD 上显示连接失败。

6.2.4 回调函数

示例工程 SimpleBLECentral 中应用使用到的的回调函数如下。

① **simpleBLECentralRssiCB** GAP RSSI 回调函数，用来显示最新的 RSSI 值。其中，RSSI（Received Signal Strength Indication）为接收的信号强度指示，是无线发送层的可选部分，用来判定链接质量，以及是否增大广播发送强度。GAP RSSI 回调函数代码内容如下：

```
static void simpleBLECentralRssiCB( uint16 connHandle,
int8 rssi )
{
LCD_WRITE_STRING_VALUE( "RSSI -dB:", (uint8) (-rssi), 10,
HAL_LCD_LINE_1 );
}
```

② **simpleBLECentralEventCB** GAP 事件回调函数，是在设备初始化阶段注册的，GAP 状态改变时调用，用来对 GAP 初始化、设备发现、设备连接、断开连接等事件进行处理。GAP 事件回调函数代码内容如下：

```
static void simpleBLECentralEventCB( gapCentralRoleEvent_
t *pEvent )
{
  switch ( pEvent->gap.opcode )
  {
    case GAP_DEVICE_INIT_DONE_EVENT:
      {
        LCD_WRITE_STRING( "BLE Central", HAL_LCD_LINE_1 );
        LCD_WRITE_STRING( bdAddr2Str( pEvent->initDone.
devAddr ),HAL_LCD_LINE_2 );
      }
      break;
  case GAP_DEVICE_INFO_EVENT:
    {
    if ( DEFAULT_DEV_DISC_BY_SVC_UUID == TRUE )
```

```
        {
          if ( simpleBLEFindSvcUuid( GAP_SERVICE_UUID,pEvent-
>deviceInfo.pEvtData, pEvent->deviceInfo.dataLen ) )
          {
            simpleBLEAddDeviceInfo( pEvent->deviceInfo.
addr,pEvent->deviceInfo.addrType );
          }
        }
      }
      break;
      case GAP_DEVICE_DISCOVERY_EVENT:
        {
          simpleBLEScanning = FALSE;
          if ( DEFAULT_DEV_DISC_BY_SVC_UUID == FALSE )
          {
            simpleBLEScanRes = pEvent->discCmpl.numDevs;
            osal_memcpy( simpleBLEDevList, pEvent->discCmpl.
pDevList,(sizeof( gapDevRec_t ) * pEvent->discCmpl.numDevs) );
          }
        LCD_WRITE_STRING_VALUE("DevicesFound",simpleBLEScanR
es,10, HAL_LCD_LINE_1 );
        if ( simpleBLEScanRes > 0 )
        {
          LCD_WRITE_STRING( "<- To Select", HAL_LCD_LINE_2 );
        }
        simpleBLEScanIdx = simpleBLEScanRes;
      }
    break;
    case GAP_LINK_ESTABLISHED_EVENT:
    {
      if ( pEvent->gap.hdr.status == SUCCESS )
      {
    simpleBLEState = BLE_STATE_CONNECTED;
    simpleBLEConnHandle = pEvent->linkCmpl.connectionHandle;
    simpleBLEProcedureInProgress = TRUE;
    if ( simpleBLECharHdl == 0 )
    {
```

```
    osal_start_timerEx( simpleBLETaskId, START_DISCOVERY_EVT,
    DEFAULT_SVC_DISCOVERY_DELAY );
    }
    LCD_WRITE_STRING( "Connected", HAL_LCD_LINE_1 );
    LCD_WRITE_STRING( bdAddr2Str( pEvent->linkCmpl.devAddr
),HAL_LCD_LINE_2 );
    }
    else
    {
    simpleBLEState = BLE_STATE_IDLE;
    simpleBLEConnHandle = GAP_CONNHANDLE_INIT;
    simpleBLERssi = FALSE;
    simpleBLEDiscState = BLE_DISC_STATE_IDLE;
          LCD_WRITE_STRING("Connect Failed",HAL_LCD_LINE_1 );
          LCD_WRITE_STRING_VALUE( "Reason:", pEvent->gap.
hdr.status, 10,HAL_LCD_LINE_2 );
        }
      }
      break;
    case GAP_LINK_TERMINATED_EVENT:
      {
        simpleBLEState = BLE_STATE_IDLE;
        simpleBLEConnHandle = GAP_CONNHANDLE_INIT;
        simpleBLERssi = FALSE;
        simpleBLEDiscState = BLE_DISC_STATE_IDLE;
        simpleBLECharHdl = 0;
        simpleBLEProcedureInProgress = FALSE;
        LCD_WRITE_STRING("Disconnected",HAL_LCD_LINE_1);
        LCD_WRITE_STRING_VALUE("Reason:",pEvent-
>linkTerminate.reason,10,HAL_LCD_LINE_2);
          }
          break;
        case GAP_LINK_PARAM_UPDATE_EVENT:
        {
          LCD_WRITE_STRING( "Param Update", HAL_LCD_LINE_1 );
        }
        break;
```

```
      default:
        break;
    }
  }
```

③ **simpleBLECentralPairStateCB** GAP绑定管理状态回调函数，用于显示配对和绑定操作的状态。GAP绑定管理状态回调函数代码内容如下：

```
static void simpleBLECentralPairStateCB( uint16
connHandle, uint8 state, uint8 status )
{
  if ( state == GAPBOND_PAIRING_STATE_STARTED )
  {
    LCD_WRITE_STRING( "Pairing started", HAL_LCD_LINE_1 );
  }
  else if ( state == GAPBOND_PAIRING_STATE_COMPLETE )
  {
    if ( status == SUCCESS )
    {
      LCD_WRITE_STRING("Pairing success",HAL_LCD_LINE_1);
    }
    else
    {
      LCD_WRITE_STRING_VALUE( "Pairing fail", status,
10,HAL_LCD_LINE_1 );
    }
  }
  else if ( state == GAPBOND_PAIRING_STATE_BONDED )
  {
    if ( status == SUCCESS )
    {
      LCD_WRITE_STRING("Bonding success",HAL_LCD_LINE_1 );
    }
  }
}
```

④ **simpleBLECentralPasscodeCB** GAP绑定管理密钥回调函数，用来生成和显示一个密钥。GAP绑定管理密钥回调函数代码内容如下：

```
static void simpleBLECentralPasscodeCB( uint8
*deviceAddr, uint16 connectionHandle, uint8 uiInputs, uint8
```

```
uiOutputs )
    {
    #if (HAL_LCD == TRUE)
      uint32  passcode;
      uint8   str[7];
      LL_Rand( ((uint8 *) &passcode), sizeof( uint32 ));   //
产生一个随机的密钥
      passcode %= 1000000;
      if ( uiOutputs != 0 )      // 向用户显示密钥
      {
      LCD_WRITE_STRING( "Passcode:",  HAL_LCD_LINE_1 );
      LCD_WRITE_STRING( (char *) _ltoa(passcode, str, 10),
HAL_LCD_LINE_2 );
      }
      GAPBondMgr_PasscodeRsp( connectionHandle, SUCCESS,
passcode );
    #endif
    }
```

6.2.5 服务发现

当集中器设备作为客户端，需要读取外设中提供的数据服务时，需要首先进行服务发现。客户端进行 GATT 服务发现时，需要给出要发现的服务的 UUID，只有服务 UUID 配对，才能获得 GATT 数据服务。前面章节提到，建立连接回调函数中使用 OSAL 定时器设置了一个定时事件 START_DISCOVERY_EVT，即为 GATT 服务发现事件。定时时间到达后，调用事件处理函数来处理这个事件。事件处理函数如下：

```
    uint16 SimpleBLECentral_ProcessEvent( uint8 task_id,
uint16 events )
    {
    ......
    1  if ( events & START_DISCOVERY_EVT )
      {
    2    simpleBLECentralStartDiscovery( );
    3    return ( events ^ START_DISCOVERY_EVT );
      }
      return 0;
    }
```

第 1 行，判断是否为 START_DISCOVERY_EVT 事件。

第2行，调用服务发现处理函数，进行GATT数据服务的发现。

第3行，将已处理的事件标志位清零，返回未处理的事件标志。

服务发现处理函数内容如下：

```
static void simpleBLECentralStartDiscovery( void )
{
1  uint8  uuid[ATT_BT_UUID_SIZE]  =  {  LO_UINT16(SIMPLEPROFILE_SERV_UUID),HI_UINT16(SIMPLEPROFILE_SERV_UUID) };
2  simpleBLESvcStartHdl  =  simpleBLESvcEndHdl  = simpleBLECharHdl = 0;
3  simpleBLEDiscState = BLE_DISC_STATE_SVC;
4  GATT_DiscPrimaryServiceByUUID( simpleBLEConnHandle, uuid,ATT_BT_UUID_SIZE, simpleBLETaskId );
}
```

第1行，指定GATT服务发现的UUID，共16bit，分为高8位和低8位。在“simpleGATTpeofile.h”文件中可以找到其定义。

第2行，将服务的起始句柄、结束句柄以及“特性”句柄清零，句柄为属性在表中的地址。

第3行，将当前发现状态标志设为服务发现。

第4行，通过指定的服务UUID发现基本服务，只发现UUID匹配的服务。这一函数处理上一次GATT过程的结果，并初始化下一步的发现操作。

接收到返回的GATT消息，首先调用事件处理函数：

```
uint16 SimpleBLECentral_ProcessEvent( uint8 task_id, uint16 events )
{
  VOID task_id;
1 if ( events & SYS_EVENT_MSG )
  {
2   uint8 *pMsg;
3   if ( (pMsg = osal_msg_receive( simpleBLETaskId )) != NULL )
    {
4     simpleBLECentral_ProcessOSALMsg( (osal_event_hdr_t *)pMsg );
5     VOID osal_msg_deallocate( pMsg );
    }
6   return (events ^ SYS_EVENT_MSG);
```

```
    }
  ......
  }
```

第 1 ～ 4 行，天线收到返回的数据后，产生一个 OSAL 消息，调用消息事件处理函数 simpleBLECentral_ProcessOSALMsg 进行处理。

第 5 行，释放 OSAL 消息。

第 6 行，将已处理的事件标志位清零，返回未处理的事件标志。

消息事件处理函数如下：

```
static void simpleBLECentral_ProcessOSALMsg( osal_event_
hdr_t *pMsg )
{
  switch ( pMsg->event )
  {
    case KEY_CHANGE:
      simpleBLECentral_HandleKeys( ((keyChange_t *)pMsg)-
>state, ((keyChange_t*)pMsg)->keys );
      break;
    case GATT_MSG_EVENT:
      simpleBLECentralProcessGATTMsg( (gattMsgEvent_t *)
pMsg );
      break;
  }
}
```

如果判断为 GATT 消息事件，则调用 GATT 事件处理函数：

```
static void simpleBLECentralProcessGATTMsg( gattMsgEvent_
t *pMsg )
{
1  if ( simpleBLEState != BLE_STATE_CONNECTED )
  {
2  return;
  }
  ......
  else if ( simpleBLEDiscState != BLE_DISC_STATE_IDLE )
  {
3  simpleBLEGATTDiscoveryEvent( pMsg );
  }
}
```

第1～2行，如果GATT消息到来时，设备已断开连接，则忽略这个消息。

第3行，进行应用数据的发现。

simpleBLEGATTDiscoveryEvent函数内容如下：

```
static void simpleBLEGATTDiscoveryEvent( gattMsgEvent_t
*pMsg )
{
  attReadByTypeReq_t req;
  if ( simpleBLEDiscState == BLE_DISC_STATE_SVC )
  {
    if(pMsg->method==ATT_FIND_BY_TYPE_VALUE_RSP&&pMsg-
>msg.findByTypeValueRsp.numInfo > 0 )
  {
1 simpleBLESvcStartHdl = pMsg->msg.findByTypeValueRsp.
handlesInfo[0].handle;
2 simpleBLESvcEndHdl = pMsg->msg.findByTypeValueRsp.
handlesInfo[0].grpEndHandle;
}
3 if((pMsg->method==ATT_FIND_BY_TYPE_VALUE_RSP&&pMsg->hdr.
status==bleProcedureComplete ) || ( pMsg->method == ATT_ERROR_
RSP ) )
{
4 if ( simpleBLESvcStartHdl != 0 )
  {
5 simpleBLEDiscState = BLE_DISC_STATE_CHAR;
6 req.startHandle = simpleBLESvcStartHdl;
7 req.endHandle = simpleBLESvcEndHdl;
8 req.type.len = ATT_BT_UUID_SIZE;
9 req.type.uuid[0] = LO_UINT16(SIMPLEPROFILE_CHAR1_
UUID);
10 req.type.uuid[1] = HI_UINT16(SIMPLEPROFILE_CHAR1_
UUID);
11 GATT_ReadUsingCharUUID( simpleBLEConnHandle, &req,
simpleBLETaskId );
      }
    }
  }
.....
```

}

第 1、2 行，获取返回的消息中的 GATT 服务的起始句柄和结束句柄，一个服务当中可能包含多个“特性”。

第 3 ～ 8 行，若从上一步骤获得服务的起始句柄不为 0，将这些参数赋值给 req 结构体，用来进行下一步的 GATT 特性发现操作。

第 9 ～ 10 行，要操作的“特性”的 UUID。

第 11 行，通过发送某个“特性”的 UUID，来获得这个特性值的句柄。

注意 ▶▶▶

“特性”(Characteristics) 是服务用到的值，以及其内容和配置信息。GATT 定义了在 BLE 连接中发现、读取和写入属性的子过程。GATT 服务器上的特性值及其内容和配置信息（称为描述符）存储于属性表中。所有应用数据的操作都需要通过“特性”进行，如果将“特性”比作一辆客车，应用数据便是乘客，“特性”的 UUID 便是车牌号，“特性”中应用数据的发现过程就相当于通过车牌号找到客车，然后才能找到客车上的乘客。将要操作的“特性”的 UUID 发送以后，再次从天线收到返回的消息，仍然通过 SimpleBLECentral_ProcessEvent->simpleBLECentralProcessGATTMsg->simpleBLEGATTDiscoveryEvent 过程进行 GATT 消息处理。

```
   static void simpleBLECentralProcessGATTMsg( gattMsgEvent_
t *pMsg )
   {
   1  if ( simpleBLEState != BLE_STATE_CONNECTED )
     {
   2  return;
     }
     ......
   3  else if ( simpleBLEDiscState != BLE_DISC_STATE_IDLE )
     {
   4    simpleBLEGATTDiscoveryEvent( pMsg );
     }
   }
```

第 1 ～ 2 行，如果 GATT 消息到来时，设备已断开连接，则忽略这个消息。

第 3 ～ 4 行，进行应用数据的发现。

再次调用 simpleBLEGATTDiscoveryEvent 处理返回的 GATT 消息：

```
static void simpleBLEGATTDiscoveryEvent( gattMsgEvent_t
*pMsg )
{
1  attReadByTypeReq_t req;
……
2  else if ( simpleBLEDiscState == BLE_DISC_STATE_CHAR )
  {
3  if(pMsg->method==ATT_READ_BY_TYPE_RSP&&pMsg->msg.
readByTypeRsp.numPairs > 0 )
    {
4    simpleBLECharHdl=BUILD_UINT16( pMsg->msg.
readByTypeRsp.dataList[0],pMsg->msg.readByTypeRsp.
dataList[1] );
5    LCD_WRITE_STRING("Simple Svc Found",HAL_LCD_LINE_1 );
6    simpleBLEProcedureInProgress = FALSE;
    }
7    simpleBLEDiscState = BLE_DISC_STATE_IDLE;
  }
}
```

第 1 行，定义一个结构体，用于处理消息中包含的数据。

第 2 ～ 4 行，从返回的数据中得到第一个特性值的句柄，因为前面已经发送了这个特性值的 UUID。

第 5 行，液晶显示字符串“Simple Svc Found”，表明已经找到要操作的“特性”的句柄，后面可以进行“特性”中数据的读写操作了。

第 6、7 行，将处理进程标志设为完成，发现状态标志设为空闲，这两个标志只是在程序的编写中起判断作用。

下面讲解使用 Key1 键，进行对第一个“特性”中值的读写操作。

Key1 按键按下后，HAL 层发送一个 OSAL 消息给应用，产生一个 SYS_EVENT_MSG 事件，通过调用 SimpleBLECentral_ProcessEvent-> simpleBLECentral_ProcessOSALMsg 函数来处理这个消息。若为按键事件，则进一步调用按键处理函数 simpleBLECentral_HandleKeys 来处理按键任务，进行读取特性值的操作如下：

```
static void simpleBLECentral_HandleKeys( uint8 shift,
uint8 keys )
{
  (void)shift;
  if ( keys & HAL_KEY_SW_1 )
```

```
    {
    ......
  1 else if ( simpleBLEState == BLE_STATE_CONNECTED
&&simpleBLECharHdl != 0&&simpleBLEProcedureInProgress ==
FALSE )
    {
       uint8 status;
  2   if ( simpleBLEDoWrite )
      {
  3      attWriteReq_t req;
  4      req.handle = simpleBLECharHdl;
  5      req.len = 1;
  6      req.value[0] = simpleBLECharVal;
  7      req.sig = 0;
  8      req.cmd = 0;
  9 status=GATT_WriteCharValue(simpleBLEConnHandle, &req,
simpleBLETaskId );
       }
       else
       {
  10      attReadReq_t req;
  11      req.handle = simpleBLECharHdl;
  12 status=GATT_ReadCharValue(simpleBLEConnHandle,&req,
simpleBLETaskId );
      }
      if ( status == SUCCESS )
      {
  13      simpleBLEProcedureInProgress = TRUE;
  14      simpleBLEDoWrite = !simpleBLEDoWrite;
      }
    }
  }
```

第 1 行，已建立连接，要操作的“特性”的句柄不为 0，且上一阶段处理进程已完成。

第 2 ～ 8 行，定义一个结构体变量 req，将句柄、要写入的应用数据等参数赋值给结构体变量。

第 9 行，调用协议栈提供的写入函数，向指定的句柄中写入数据，并返回状态

标志来判断是否正在进行写入数据的操作。

第10、11行，定义一个结构体变量req，将要读取的特性值的句柄赋值给结构体变量。

第12行，调用协议栈提供的读取函数，从指定的句柄中读取数据，并返回状态标志来判断是否正在进行读取数据的操作。

第13行，将处理过程标志赋值为TRUE，表示读取数据过程正在进行。

第14行，将写标志取反，这样在下次按键事件到来时，执行的为写入数据操作。

天线将会接收返回的数据，调用事件处理函数流程为SimpleBLECentral_ProcessEvent->simpleBLECentral_ProcessOSALMsg->simpleBLECentralProcessGATTMsg。

GATT消息处理函数simpleBLECentralProcessGATTMsg内容如下：

```
static void simpleBLECentralProcessGATTMsg( gattMsgEvent_
t *pMsg )
{
1  if ( simpleBLEState != BLE_STATE_CONNECTED )
   {
2     return;
   }
3   if ( ( pMsg->method == ATT_READ_RSP ) ||( ( pMsg-
>method == ATT_ERROR_RSP )
&&( pMsg->msg.errorRsp.reqOpcode == ATT_READ_REQ ) ) )
{
4   if ( pMsg->method == ATT_ERROR_RSP )
    {
5     uint8 status = pMsg->msg.errorRsp.errCode;
6      LCD_WRITE_STRING_VALUE( "Read Error", status, 10,
HAL_LCD_LINE_1 );
    }
    else
    {
7    uint8 valueRead = pMsg->msg.readRsp.value[0];
8    LCD_WRITE_STRING_VALUE("Read rsp:", valueRead, 10,
HAL_LCD_LINE_1 );
    }
9   simpleBLEProcedureInProgress = FALSE;
   }
   ......
```

```
}
```

第 1、2 行，如果 GATT 消息到来时，设备已断开连接，则忽略这个消息。

第 3 ～ 6 行，若返回读操作失败，获取操作的错误码，并在 LCD 上显示。错误码代表了出错的原因。

第 7、8 行，若返回读操作成功，在 LCD 上显示读取的特性值，即为应用数据。

第 9 行，将处理过程标志赋值为 FALSE，表示读操作已经完成。

再次按下 Key1 按键时，将 simpleBLECharVal 的值（初始值为 0）写入到第一个“特性”中。　处理流程仍然为：HAL 层发送一个 OSAL 消息给应用，产生一个 SYS_EVENT_MSG 事件，通过调用 SimpleBLECentral_ProcessEvent → simpleBLECentral_ProcessOSALMsg 函数来处理这个消息。若为按键事件，则进一步调用按键处理函数 simpleBLECentral_HandleKeys 来处理按键任务，进行写操作。

```
static void simpleBLECentral_HandleKeys( uint8 shift,
uint8 keys )
{
  (void)shift;
  if ( keys & HAL_KEY_SW_1 )
  {
  ......
1   else if ( simpleBLEState == BLE_STATE_CONNECTED
&&simpleBLECharHdl != 0
&&simpleBLEProcedureInProgress == FALSE )
    {
    uint8 status;
2   if ( simpleBLEDoWrite )
    {
3       attWriteReq_t req;
4       req.handle = simpleBLECharHdl;
5       req.len = 1;
6       req.value[0] = simpleBLECharVal;
7       req.sig = 0;
8       req.cmd = 0;
9 status=GATT_WriteCharValue(simpleBLEConnHandle, &req,
simpleBLETaskId );
    }
    ......
    if ( status == SUCCESS )
    {
```

```
10      simpleBLEProcedureInProgress = TRUE;
11      simpleBLEDoWrite = !simpleBLEDoWrite;
    }
  }
}
```

第1行，已建立连接，要操作的“特性”的句柄不为0，且上一阶段处理进程已完成。

第2～8行，定义一个结构体变量req，将句柄、要写入的应用数据等参数赋值给结构体变量。

第9行，调用协议栈提供的写入函数，向指定的句柄中写入数据，并返回状态标志来判断是否正在进行写入数据的操作。

第10行，将处理过程标志赋值为TRUE，表示写入数据过程正在进行。

第11行，将写标志取反，这样在下次按键事件到来时，执行的为读取数据操作。

天线将会接收返回的数据，调用事件处理函数流程为SimpleBLECentral_ProcessEvent->simpleBLECentral_ProcessOSALMsg->simpleBLECentralProcessGATTMsg。

GATT消息处理函数simpleBLECentralProcessGATTMsg内容如下（省略部分无关代码）：

```
static void simpleBLECentralProcessGATTMsg( gattMsgEvent_
t *pMsg )
{
  ......
  else if ((pMsg->method==ATT_WRITE_RSP)|| ( ( pMsg-
>method == ATT_ERROR_RSP )&& ( pMsg->msg.errorRsp.reqOpcode
== ATT_WRITE_REQ ) ) )
  {
1  if ( pMsg->method == ATT_ERROR_RSP == ATT_ERROR_RSP )
    {
2     uint8 status = pMsg->msg.errorRsp.errCode;
3     LCD_WRITE_STRING_VALUE( "Write Error", status, 10,
HAL_LCD_LINE_1 );
    }
    else
    {
4      LCD_WRITE_STRING_VALUE( "Write sent:",
simpleBLECharVal++, 10,HAL_LCD_LINE_1 );
    }
5   simpleBLEProcedureInProgress = FALSE;
```

```
    }
    ......
}
```

第 1 ～ 3 行，若返回写操作失败，获取操作的错误码，并在 LCD 上显示。错误码代表了出错的原因。

第 4 行，若返回写操作成功，在 LCD 上显示写入外部设备中的值，并将这个值加 1 来作为下一次要写入的值。

第 5 行，将处理过程标志赋值 FALSE，表示写操作已经完成。

下面总结一下这个示例工程实现的功能及操作流程。

▲ 首先集中器设备发起扫描请求，扫描正在发送广告的外部设备，若 GAP 服务的 UUID 相匹配，则可以建立连接。

▲ 集中器发起建立连接的请求，外部设备响应后，两设备进入连接状态。

▲ 集中器通过特定的 UUID 进行 GATT 服务发现。

▲ 发现 GATT 服务后，集中器发送要进行数据操作的“特性”的 UUID，得到这个“特性”的句柄，句柄即为在属性表中的地址。

▲ 通过句柄进行应用数据的读取和写入操作。上述示例工程是通过按键来触发数据的读写操作，当按键按下时，交替地进行读取特性值和写入特性值的操作，并且将每次写入的值增加 1。

6.3 网络处理器工程简介

在 TI 为开发者免费提供的软件开发套件中包含一个 HostTestRelease 网络处理器工程，它实现了在 CC2540 上的网络处理器配置，即控制器和主机部分在 CC2540 上执行，而应用程序和配置文件在另一个设备执行。应用程序通过厂商特定的 HCI 命令与 CC2540 通信。这一过程需要通过串口（或虚拟串口）来读写主机控制接口（HCI）命令。如果需要对设备进行任何操作，需要由外部控制器通过串口来发送控制命令，当设备收到任何消息或需要采取某种操作，它会将消息发送给外部控制器。

6.3.1 工程概述

HostTestRelease 网络处理器工程结构类似于外部节点设备示例工程，同样使用硬件抽象层、操作系统虚拟层，虽然它也包含一个称之为“APP”的工程文件夹，但这些文件并不是真正的应用程序，只是简化的代码层，用来将外部 PC 发来的消息转化为调用协议栈的 API 功能。任何从协议栈收到的消息都会发送给外部 PC。这些转换的所有源代码都包含在“hci_ext_app.c”文件中。HostTestRelease 网

络处理器工程如图6-6所示。

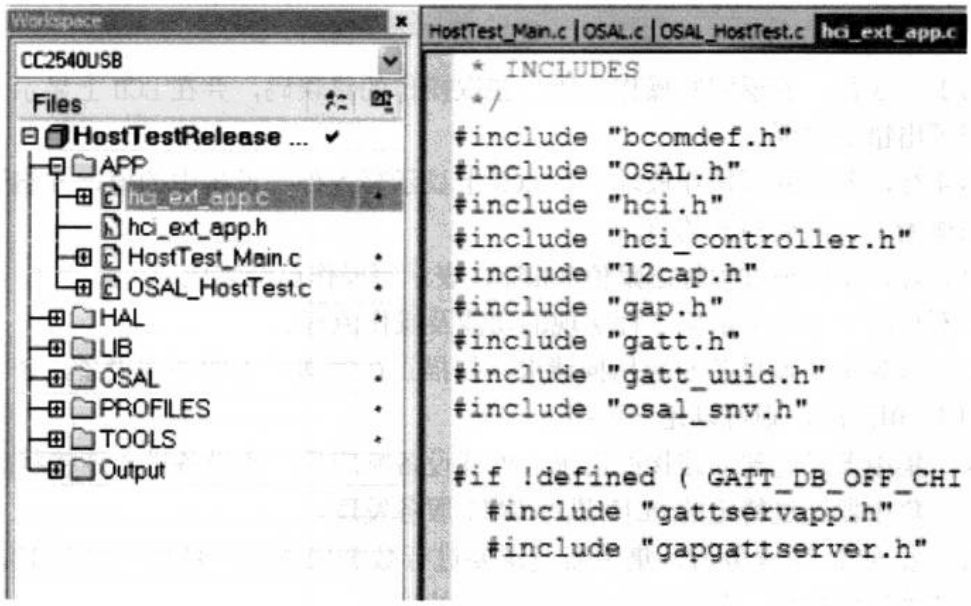


图6-6 HostTestRelease 网络处理器工程主界面

网络处理器配置适用于应用程序在另一个设备（外部微控制器或PC）上运行的情况。在这种情况下，应用程序可以在外部独立开发，而协议栈仍然在CC2540上运行。要使用网络处理器，就需要用到HostTestRelease工程。

6.3.2 BLE协议栈的外部设备控制

只需使用HostTestRelease网络处理器工程，外部控制器就可以通过串口命令来控制CC2540蓝牙设备。这样就可以使用一个外部设备，如微控制器或PC，来进行BLE协议栈的应用开发了。一个常见的例子就是在PC上使用BToll应用软件，BToll允许用户使用基本的BLE集中器设备功能，例如发现蓝牙外设或广播设备、建立与外设的连接、进行GATT应用数据的读写操作、绑定服务等。因此，可以在PC端使用BToll来进行蓝牙外设应用程序的开发调试。关于BToll工具的使用将在第7章进行详细阐述。

6.4 本章小结

本章通过对节点设备示例工程和集中器示例工程的讨论与分析，详细阐述了事

件处理、回调函数、周期事件、按键服务、数据读写等操作的的流程，尤其是针对初学者最为关心的应用数据读写问题进行了细致分析，并严格按照处理流程对代码展开讲解。相信读者通过本章的学习，已经对蓝牙 4.0 BLE 协议栈的机制以及应用数据的读写操作有了一定认识，可以通过简单的修改代码来实现自定义的数据操作。在本章的最后对网络处理器工程的应用作了简单介绍，初学者只需要简单了解其配置即可。

[扩展阅读] 探究接收数据的存放位置

经过前面的学习，读者可能仍然有以下疑问，当集中器设备接收到节点设备返回的与应用数据相关的消息后，是怎样找到数据存放的位置并进行相应处理的呢？只要理解清楚这个问题，相信很多困扰读者的问题就会迎刃而解了。

当天线接收到应用数据后，协议栈会将接收到的数据以及与该数据相关的一些信息打包，然后存储起来，读者找到这个数据包就可以找到所需要的信息。既然是将数据进行了打包，那么这个数据包是用什么样的数据结构来实现的呢？

观察 OSAL 消息处理函数 simpleBLECentral_ProcessOSALMsg()，函数原型如下：

```
static void simpleBLECentral_ProcessOSALMsg( osal_event_
hdr_t *pMsg )
{
  switch ( pMsg->event )
  {
    case KEY_CHANGE:
      simpleBLECentral_HandleKeys(((keyChange_t*)pMsg)-
>state,((keyChange_t*)pMsg)->keys );
      break;
    case GATT_MSG_EVENT:
      simpleBLECentralProcessGATTMsg( (gattMsgEvent_t *)
pMsg );
      break;
  }
}
```

可以看到在该函数中调用了 GATT 消息处理函数进行应用数据的处理，其中包含了一个 gattMsgEvent_t 类型的参数 pMsg，与数据的存放密切相关。

下面来看一下 gattMsgEvent_t 结构体中包含的内容：

```
typedef struct
```

```
{
1  osal_event_hdr_t hdr;
2  uint16 connHandle;
3  uint8 method;
4  gattMsg_t msg;
} gattMsgEvent_t;
```

第1行，GATT消息事件及状态。

第2行，收到的连接消息。

第3行，消息类型。

第4行，该成员变量为问题的关键所在，包含了大量信息，下面对gattMsg_t结构体进行剖析。

```
typedef union
{
  ……
  attErrorRsp_t errorRsp;      // 包含错误码
  attExchangeMTURsp_t exchangeMTURsp;
  attFindInfoRsp_t findInfoRsp;      // 包含句柄和UUID
  attFindByTypeValueRsp_t findByTypeValueRsp;
  attReadByTypeRsp_t readByTypeRsp;
  attReadRsp_t readRsp;      // 收到的特性值数据
  attReadBlobRsp_t readBlobRsp;
  attReadMultiRsp_t readMultiRsp;      // 多个特性值数据的集合
  attReadByGrpTypeRsp_t readByGrpTypeRsp;
  attPrepareWriteRsp_t prepareWriteRsp;
  attHandleValueNoti_t handleValueNoti;    // 特性值句柄通知
  attHandleValueInd_t handleValueInd;      // 特性值句柄指示
}  gattMsg_t;
```

gattMsg_t结构体中成员变量较多，不必强记，只需尝试着使用这些参数，自然就熟悉了。从上面结构体中可以发现readRsp这一成员变量，接收到的特性值数据就包含在其中。readRsp这一成员变量仍然是一个结构体类型，定义如下：

```
typedef struct
{
1  uint8 len;
2  uint8 value[ATT_MTU_SIZE-1];
} attReadRsp_t;
```

第1行，数据的长度。

第2行，给定句柄处的特性值，即为我们要找的应用数据。

到此为止，已找到了数据的存放位置，还需要提醒读者注意，从消息队列中接收到了包含应用数据的消息，才能调用 GATT 数据处理函数进行相应的处理。那么，接收到的消息又是如何包含数据信息的呢？

下面再来回顾一下在第 5.3.1 节中提到了 OSAL 的消息队列，OSAL 的消息队列如图 6-7 所示。

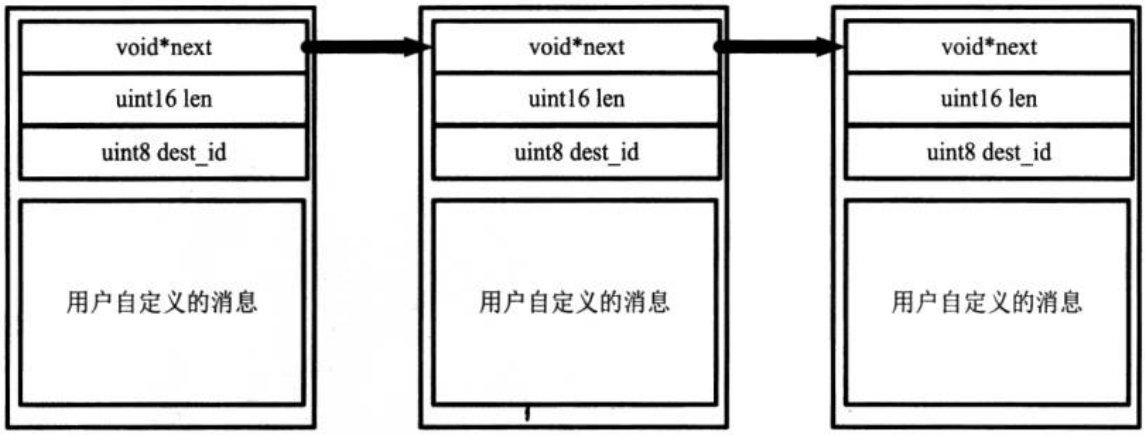
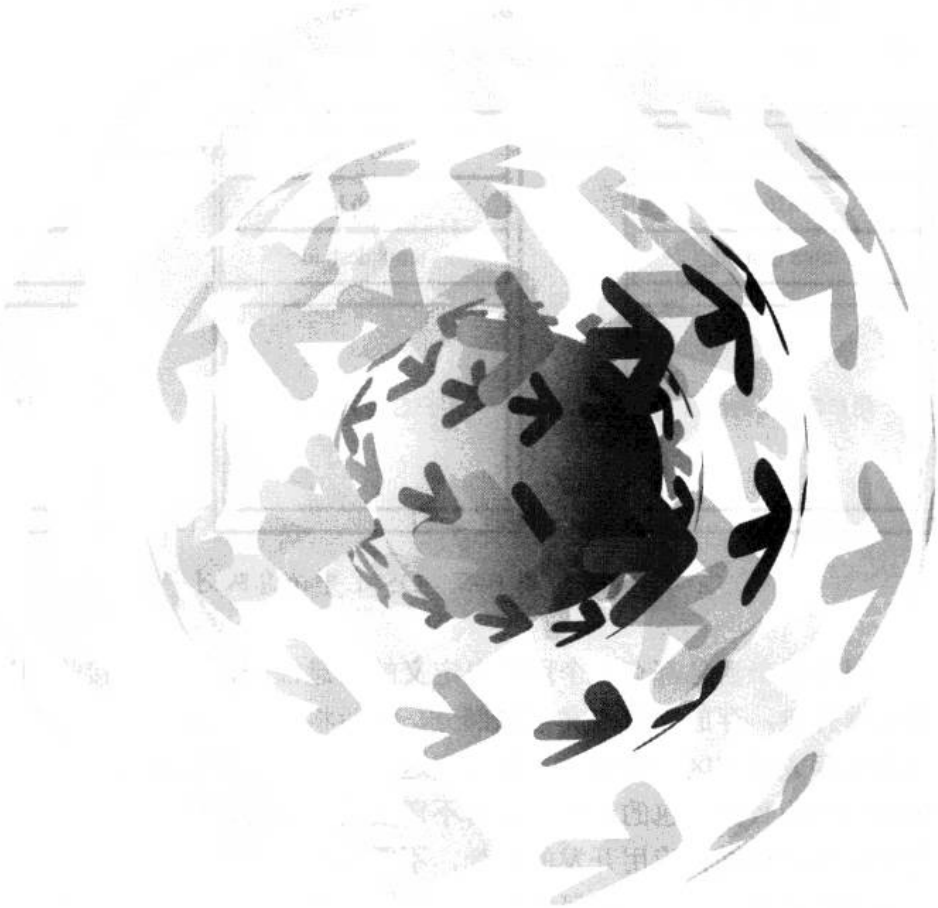


图 6-7　OSAL 中消息队列

因为每个消息都有一个用户自定义的消息部分，从天线接收到数据后，协议栈将数据打包，存放在自定义消息部分，然后将其插入消息队列。消息中存放的为指向数据存放缓冲区的指针，只要找到这个指针，就可以找到接收到的数据了。至于将接收到的数据打包的过程，读者不需要关心，协议栈已经实现了这些内容。这就是使用协议栈进行应用开发的好处，不需要过多地关注协议的实现细节，只需要掌握数据的传输流程以及如何使用协议栈提供的函数即可进行应用开发。



第7章

PC端调试开发工具

BToll 是一个 PC 端应用程序，通过使用主机控制接口命令的方式与蓝牙 BLE 外设通信。BToll 允许用户使用基本的 BLE 集中器设备功能，例如发现蓝牙外设或广播设备、建立与外设的连接、进行 GATT 应用数据的读写操作、绑定服务等。因此，可以在 PC 端使用 BToll 工具来进行蓝牙外设应用程序的开发调试。

7.1 启动 BToll 工具

首先向 USB Dongle 板（可用节点板替代）中写入固件“HostTestRelease 工程”，将 USB Dongle 与 PC 连接，这样 BToll 就可以与 USB Dongle 通过虚拟串口通信了。安装蓝牙 4.0 BLE 协议栈之后，在路径 C:\TexasInstruments\BLE-CC254x-1.2.1\Projects 目录下找到 BToll，双击打开应用程序。如图 7-1 所示。

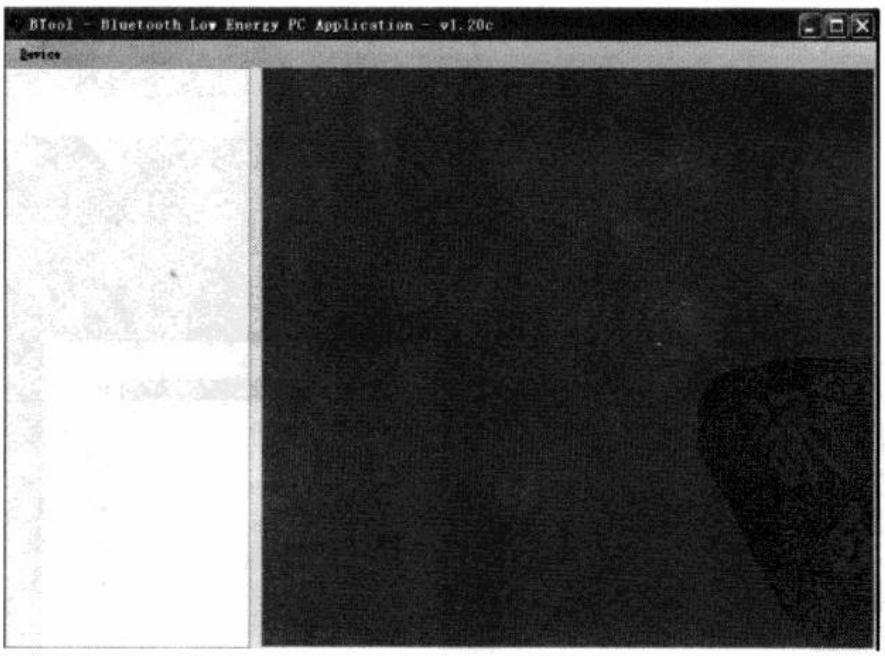


图 7-1　启动 BToll 工具

在界面的左上角单击“Device”，在弹出的下拉列表中选择“New Device”，如图 7-2 所示。

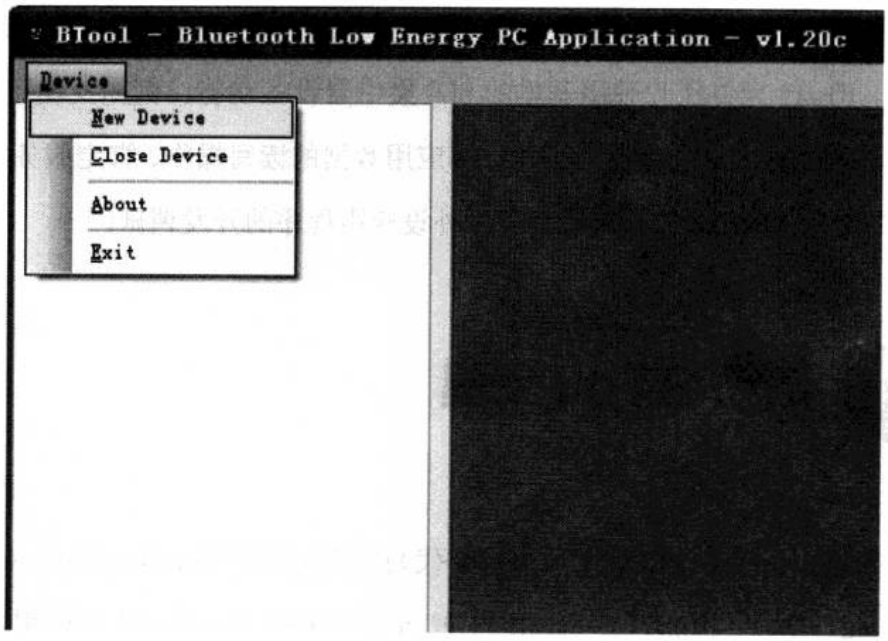


图 7-2　选择“New Device”

在新弹出的对话框中设置好端口及波特率，然后单击“OK”，如图 7-3 所示。

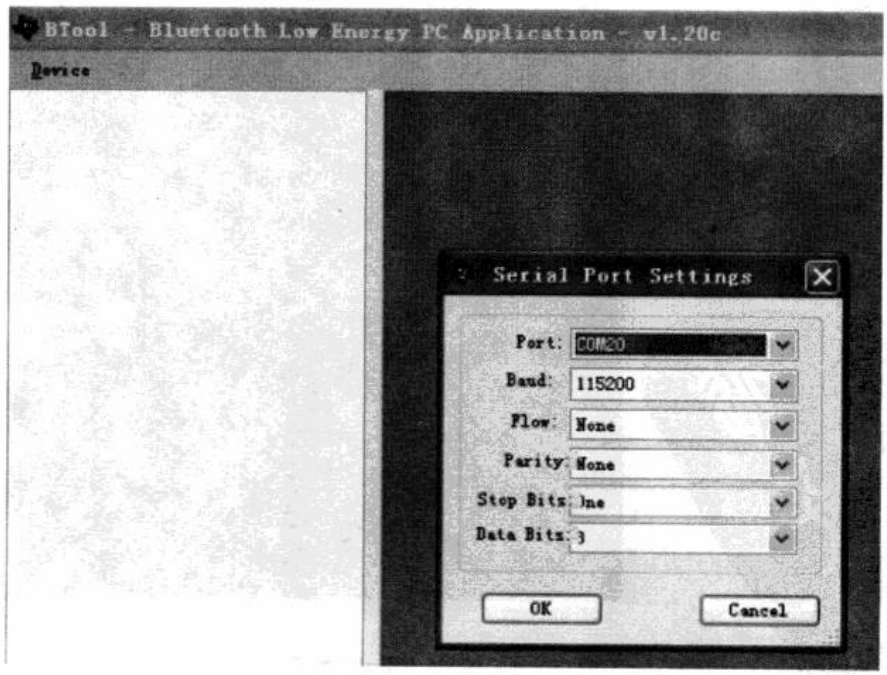


图 7-3　串口设置

建立 BTo11 与 USB Dongle 的连接如图 7-4 所示。

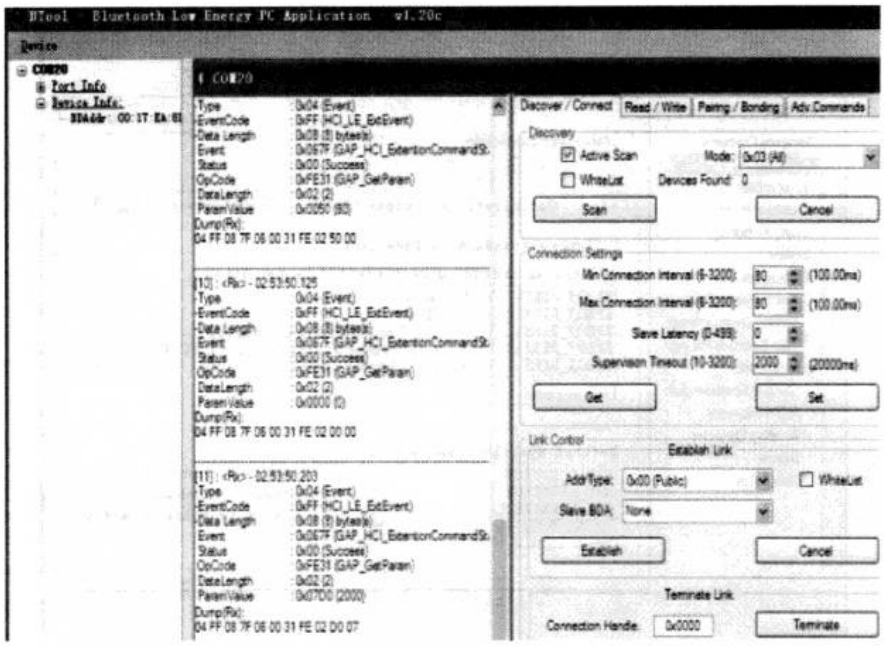


图 7-4　BToll 与 USB Dongle 的连接

BToll 与 USB Dongle 建立连接后的窗口分为三个区域：左边为端口和 USB Dongle 的信息，包括连接端口号、波特率、USB Dongle 的 IEEE 地址等信息；中间部分为 BToll 通过 USB Dongle 发送和接收的数据记录；右边部分为 HCI 命令，用于对 USB Dongle 的控制。

7.2 创建 BToll 与蓝牙节点设备的连接

BToll 可通过使用 USB Dongle 来与蓝牙节点设备进行通信，同样需要扫描广告设备、发起建立连接请求等操作来建立与节点设备的连接。BToll 在这里充当集中器设备的角色，可以同时与三个蓝牙节点设备保持连接状态。

7.2.1 设置节点设备的可发现状态

蓝牙节点设备上电后，可能不会立即处于可发现状态，这就需要设置节点设备的可发现模式。若想要蓝牙节点设备处于受限的可发现模式，即按下按键后才能发送广告，在接下来的 30s 内处于可发现状态，然后返回待机状态，可在预编译选项中输入宏 CC2540_MINIDK，将节点设备设置为受限的可发现模式，如图 7-5 所示。

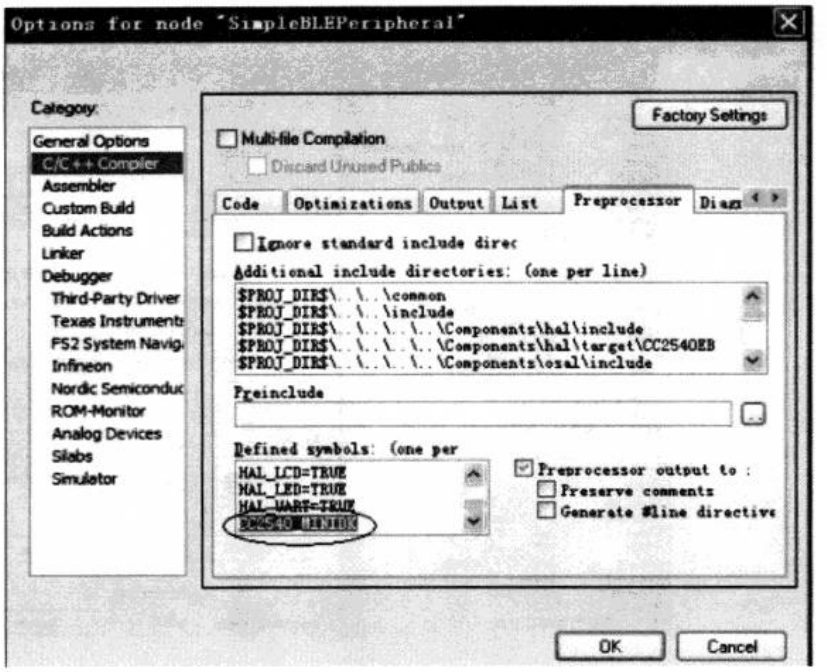


图 7-5　预编译选项中设置为受限的可发现模式

若想要蓝牙节点设备上电后，在未建立连接的情况下，一直处于可发现状态，向预编译选项中输入宏 xCC2540_MINIDK，即设为不受限的可发现模式，如图 7-6 所示。

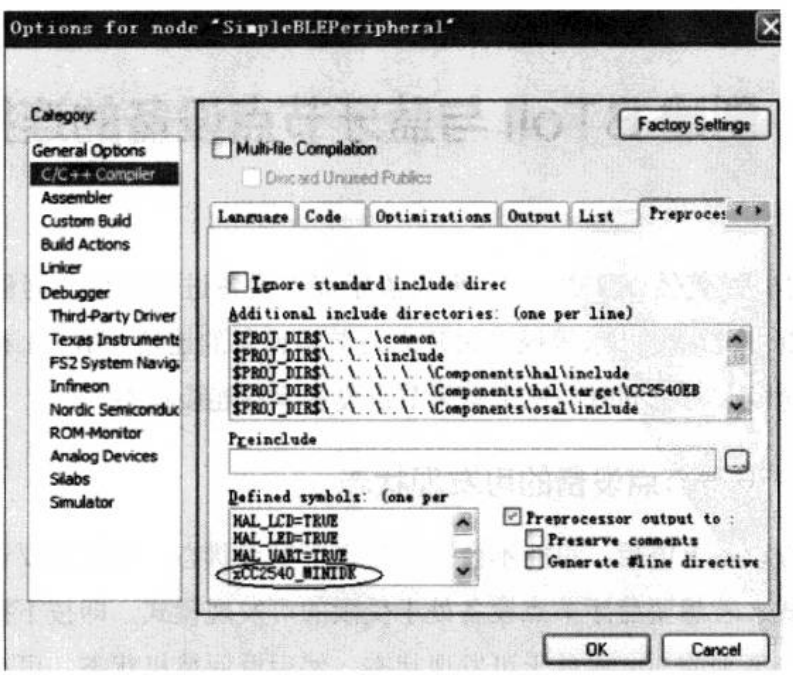


图 7-6　预编译选项中设置为不受限的可发现模式

7.2.2 扫描节点设备

在 BToll 工具的 Discover/Connect 选项卡中，单击“Scan”，对正在发送广告的节点设备进行扫描，如图 7-7 所示。

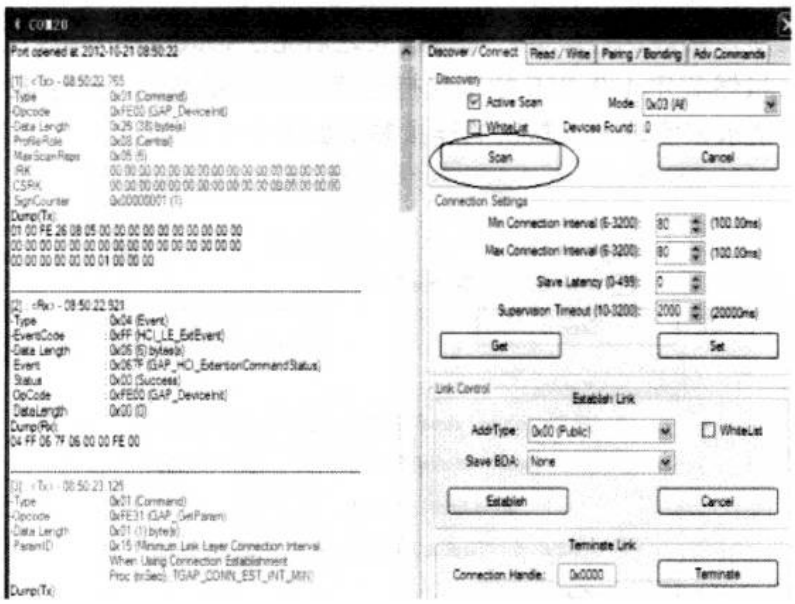


图 7-7　Discover/Connect 选项卡

默认的扫描时间为 10s，若在扫描过程中已经找到了想要与之建立连接的节点设备，也可以单击“Cancel”来停止扫描。扫描完成后，在右侧的记录窗口会显示扫描到的设备个数以及设备的地址，如图 7-8 所示。

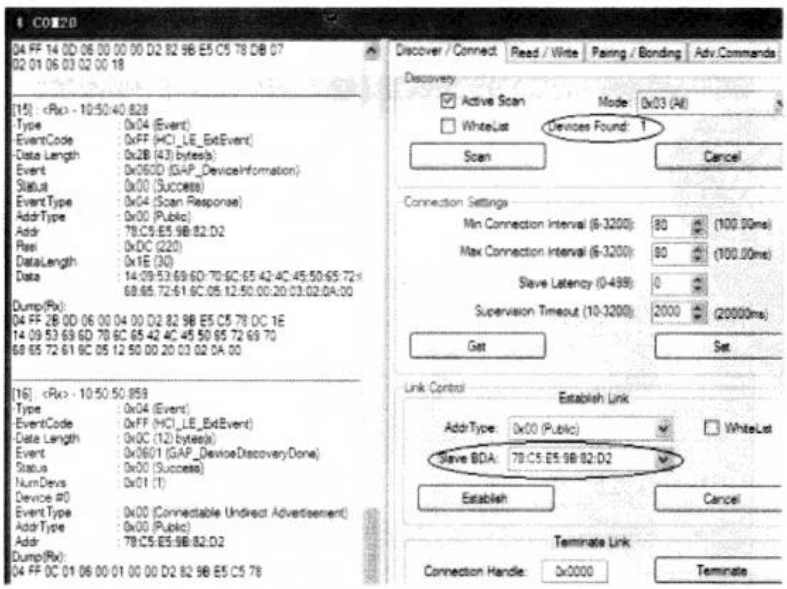


图 7-8　扫描结果

7.2.3 连接参数设置

在建立设备连接之前，需要设置连接参数，包括最小最大连接间隔、从机延时和管理超时。可以使用默认的参数，也可以针对不同应用来调整这些参数。

设置好连接参数值后，单击“Set”按钮进行确定，否则设置不会生效。另外，需要在连接之前设置好这些参数，连接建立之后无法修改这些参数，必须断开连接并重新设置（虽然蓝牙 4.0 BLE 规范支持连接状态更新参数，但这需要通过 L2CAP 连接参数更新请求来实现）。默认的连接参数如图 7-9 所示。

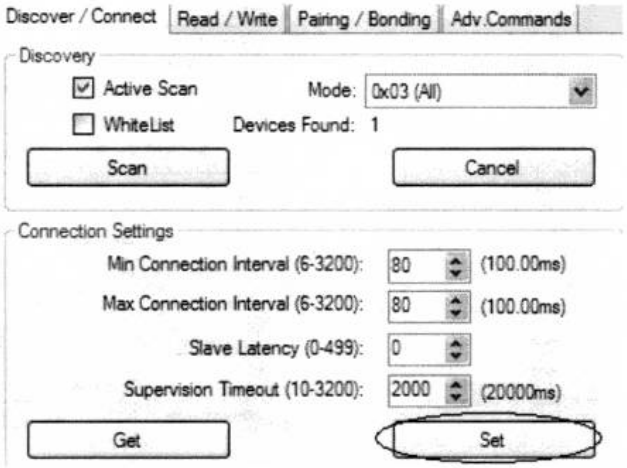


图 7-9 连接参数设置

7.2.4 建立连接

在“Slave BDA”一栏的下拉列表中选择要与之建立连接的节点设备的地址，并单击“Establish”按钮，如图 7-10 所示。

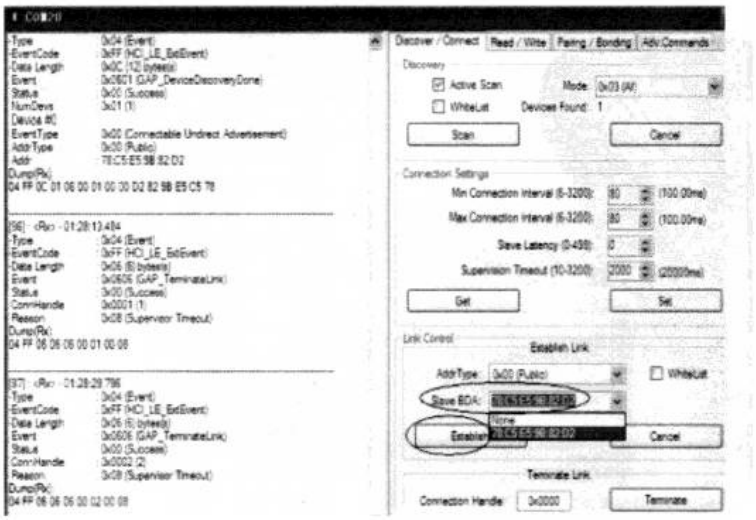


图 7-10 选择节点设备地址

如果连接参数有效并且此时节点设备仍然处于广告状态，则连接建立。此时，节点设备的信息会出现在窗口左侧，如图 7-11 所示。

图 7-11　建立连接

如果扫描到多个处于广告状态的节点设备，可以依次与它们建立连接。具体步骤为在“Slave BDA”一栏的下拉列表中选择其中一个节点设备地址，单击“Establish”按钮，建立连接，然后选择下一个设备地址，再次单击“Establish”按钮，依次类推。目前 CC2540 支持集中器设备同时与三个节点设备保持连接状态，BToll 在这里担任的是集中器的角色，与多个节点设备建立连接如图 7-12 所示。

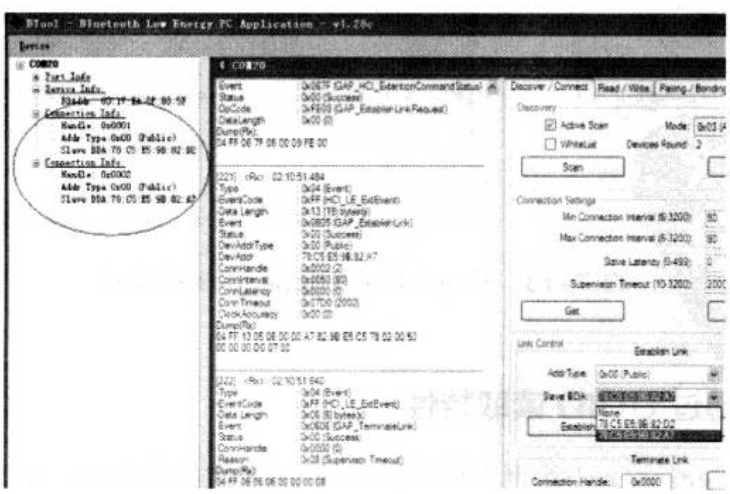


图 7-12　与多个节点设备建立连接

7.3 GATT 数据服务

在 SimpleBLEPeripheral 示例工程中提供了一个简单的 GATT 数据服务，GATT 服务包含一些称之为“特性”的数据，“特性”(Characteristics) 是服务用到的值，以及其内容和配置信息。GATT 定义了在 BLE 连接中发现、读取和写入属性的子过程。GATT 服务器上的特性值，及其内容和配置信息（称为描述符）存储于属性表中。所有应用数据的操作都需要通过“特性”进行，如果将“特性”比作一辆客车，应用数据便是乘客，“特性”的 UUID 便是车牌号，“特性”中应用数据的发现过程就相当于通过车牌号找到客车，然后才能找到客车上的乘客。在 SimpleBLEPeripheral 示例工程的 Profiles 文件夹中有两个源文件“simpleGATTprofile.c”和“simpleGATTprofile.c”，在这两个源文件里包含了所有 GATT 数据服务的相关内容。如图 7-13 所示。

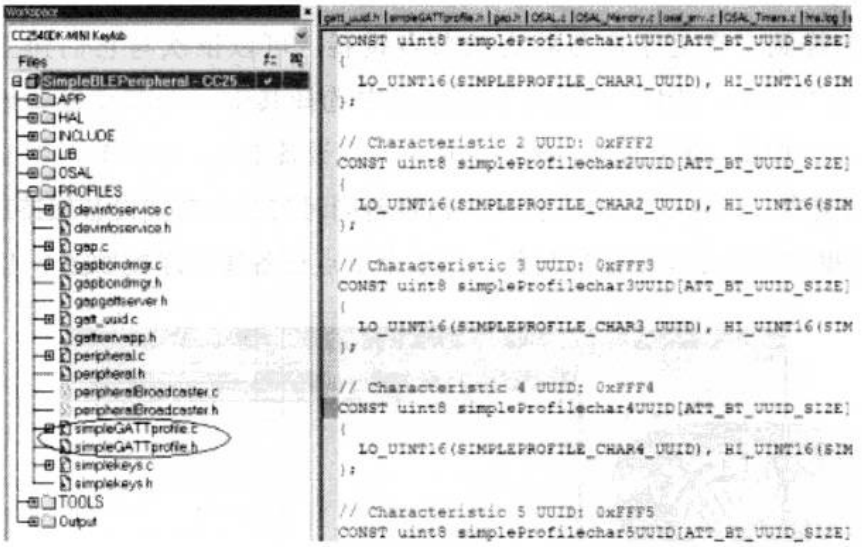


图 7-13　SimpleBLEPeripheral 示例工程

7.3.1　通过 UUID 读取特性

SimpleBLEPeripheral 示例工程中提供的 GATT 数据服务中，定义了 5 个特性值的 UUID，分别为 0xFFF1，0xFFF2，0xFFF3，0xFFF4，0xFFF5，以及主服务的 UUID（0xFFF0），如图 7-14 所示。这些并不属于蓝牙 SIG 技术联盟的规范，读者也可以根据需要自己定义。

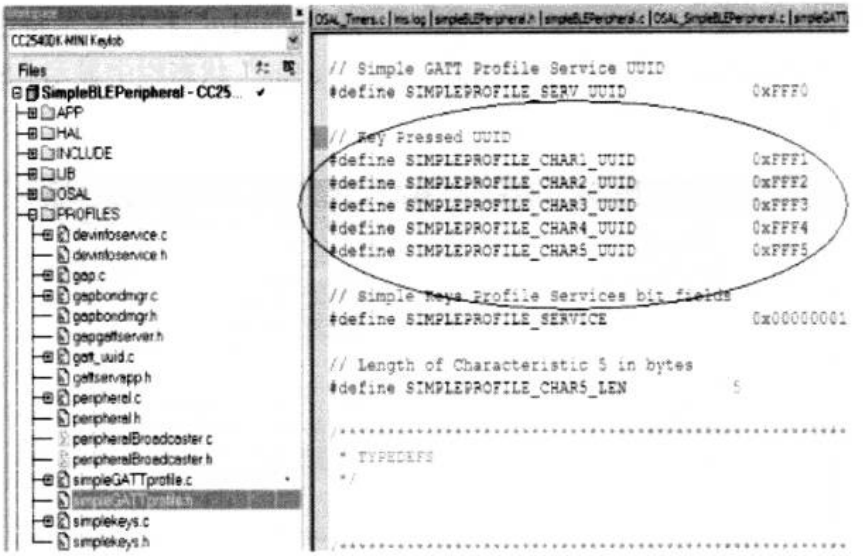


图 7-14　定义 5 个特性的 UUID

其中，第一个特性值具有读写属性，第二个特性值为只读属性，第三个特性值为只写属性，第四个特性值为只通告属性，第五个特性值为只读属性，并且只能在连接加密时读取。这些读写权限是在“simpleGATTprofile.c”文件的属性表中定义的，读者也可以自定义特性。

因为第一个特性值具有读写属性，UUID 为 0xFFF1，可以使用 UUID 来读取这个特性的信息。在 BToll 的“Read/Write”选项卡下的特性读取栏选择“Read Using Characteristic UUID”，向“Characteristic UUID”栏中写入“F1:FF”，并单击“Read”按钮，如图 7-15 所示。

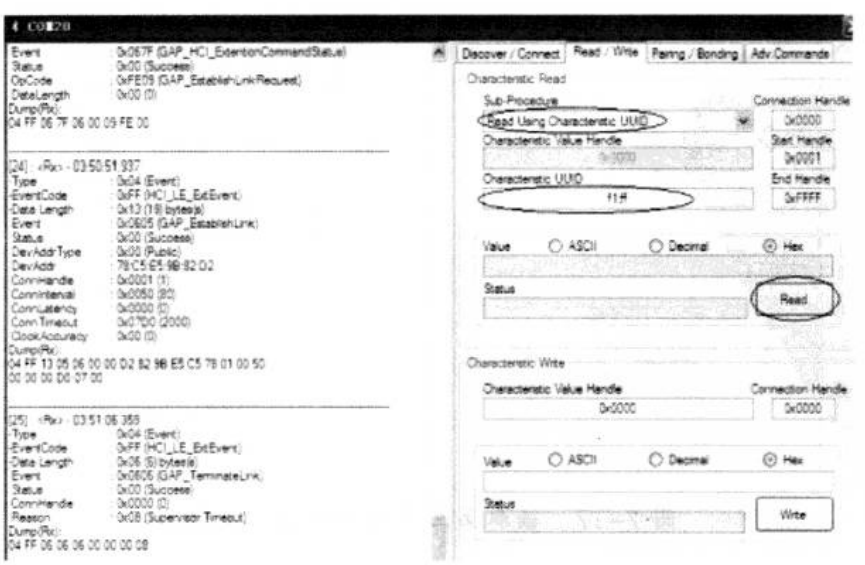


图 7-15　使用 UUID 读特性

读取成功后，会在“Characteristic Value Handle”栏中显示特性值的句柄“0x0025”，在“Value”栏中显示特性值，在“Status”状态栏中显示“Success”。如图7-16所示。

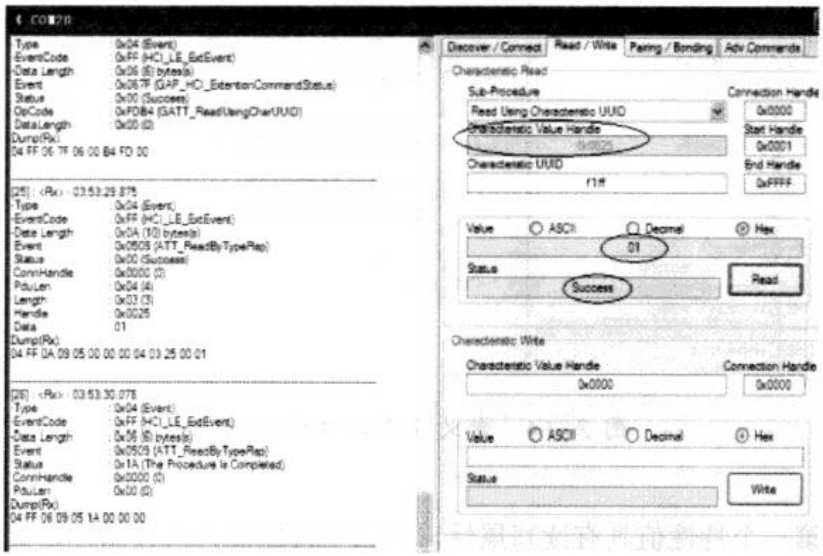


图7-16 成功读取特性信息

下面尝试一下使用UUID读取第三个特性的信息，向“Characteristic UUID”栏中写入“F3:FF”，并单击“Read”按钮，发现无法读取第三个特性的信息，结果如图7-17所示。

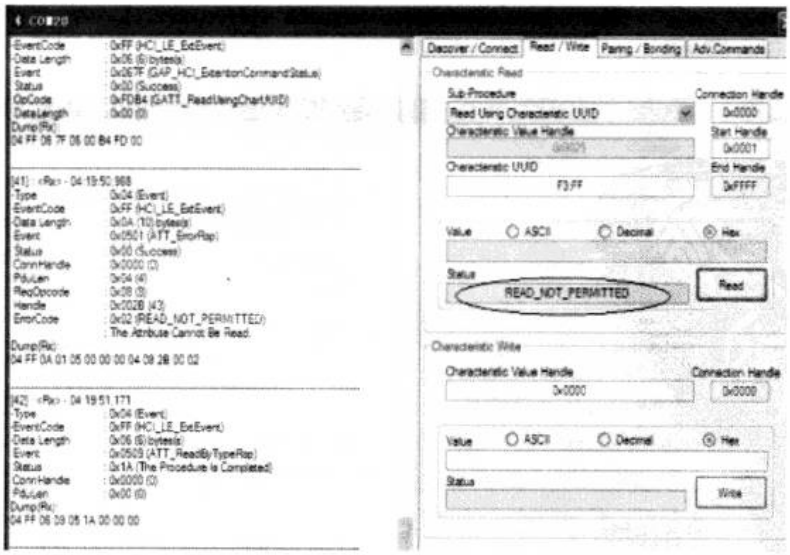


图7-17 读取第三个特性的信息失败

这是因为在属性表中将第三个特性值定义为只写属性，所以无法使用UUID读

取特性值。那么，如何获取这个只具有写属性的特性的句柄呢？这一部分将在后面第 7.3.4 节进行讲述。

7.3.2 通过句柄写入特性值

上一小节讲述了通过特性的UUID获取节点设备的GATT数据服务中特性的信息，这些特性的信息中包括了特性的句柄。本节讲述如何使用句柄来写入特性值。在上一节中使用第一个特性的UUID得到了特性值的句柄“0x0025”，向“Characteristic Write”区域中的“Characteristic Value Handle”一栏中输入句柄“0x0025”，向“Value”栏中输入数据，例如“99”，单击“Write”按钮，如图 7-18 所示。

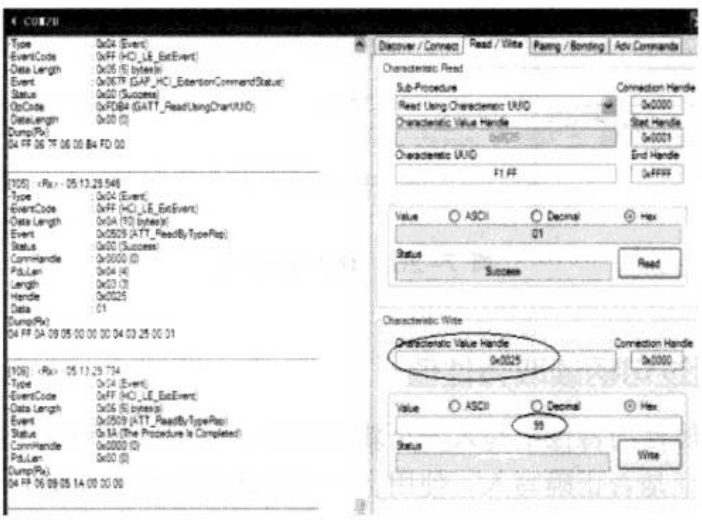


图 7-18 使用句柄写入特性值

新的值会被写入特性中，状态栏中显示成功，如图 7-19 所示。

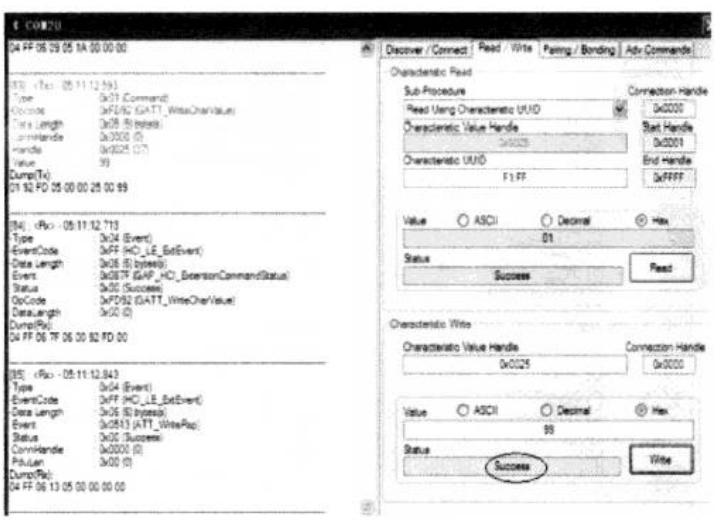


图 7-19 特性值写入成功

需要注意的是，写入的数据是有长度限制的，特性值的长度在“simpleGATTprofile.c”文件中定义，前四个特性值的长度为1字节，第五个特性值长度为5字节，如图7-20所示。

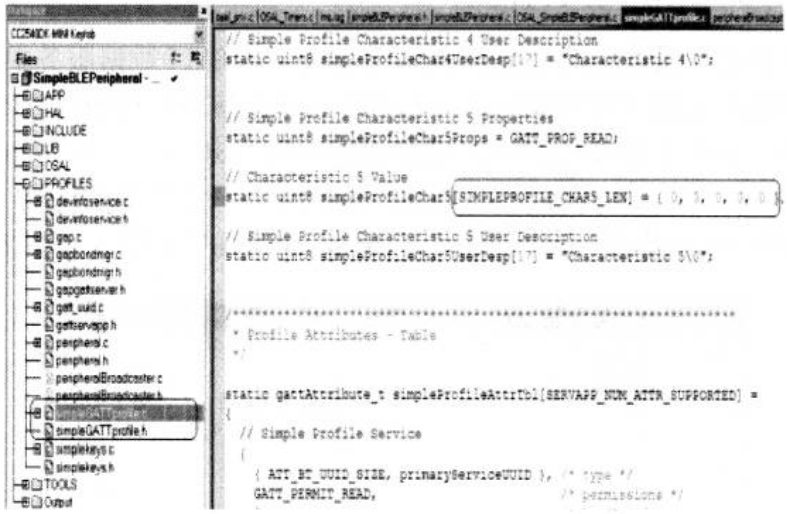


图7-20 特性值长度

7.3.3 通过句柄读取特性值

向具有读写属性的特性中写入一个新数据后，也可以使用这个特性值的句柄将数据读出，以验证是否正确写入。使用上一节讲述的方法先向第一个特性值中写入“99”，读取这个特性值的步骤为在“Read/Write”选项卡的“Characteristic Read”栏中选择“Read Characteristic Value/Descriptor ”，在“Characteristic Value Handle”栏中输入“0x0025”句柄，然后单击“Read”，此时读出了上一步写入的新数据“99”，如图7-21所示。

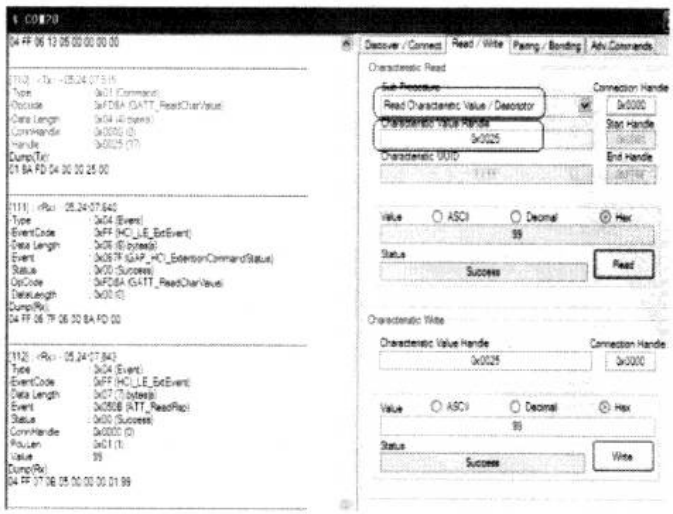


图7-21 通过句柄读特性值

7.3.4 通过 UUID 发现特性

在第 7.3.1 节中提到，属性表中将第三个特性值定义为只写属性，所以无法使用 UUID 读取特性值。本节讲述使用 UUID 发现的方式来获取这个只具有写属性的特性的句柄及其属性。在“Read/Write”选项卡的“Characteristic Read”栏中选择“Discover Characteristic by UUID”，在“Characteristic UUID”栏中输入“F3:FF”，然后单击“Read”，如图 7-22 所示。

图 7-22 使用 UUID 发现特性

此时，可以读取第三个特性值的句柄“0x002B”，如图 7-23 所示。

图 7-23 获取第三个特性值的句柄

在“Value”栏中读到了一组数据“08 2B 00 F3 FF”，这并不是特性值，因为第三个特性为只写属性，无法读取它的特性值。下面来分析一下得到的这一组

数据的含义。其中，第一个字节“08”，按照蓝牙协议，表示只写属性；第二、三字节“2B00”表示这个特性的句柄；最后两个字节“F3FF”为第三个特性的UUID。

使用上述方法，进行具有只读属性的属性表中第二个特性的发现，如图7-24所示。

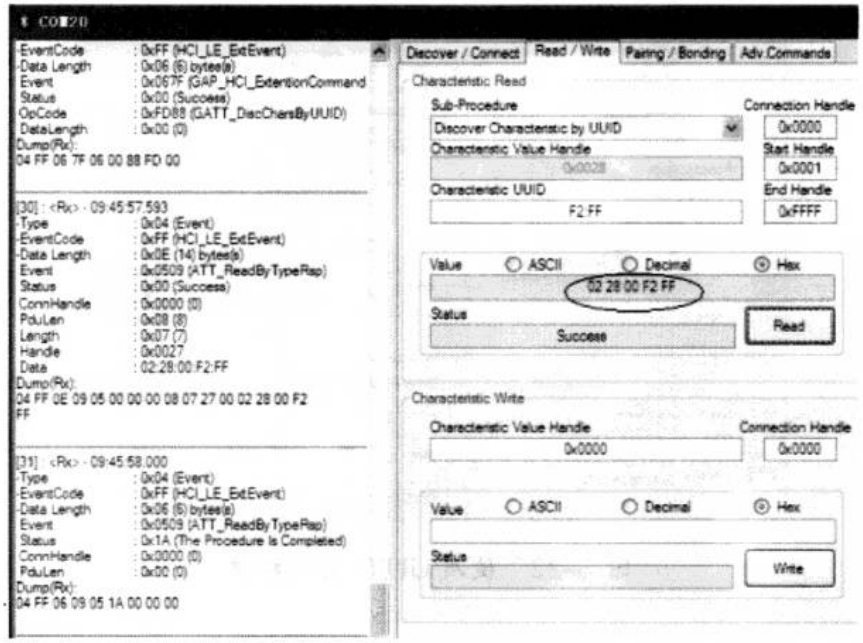


图7-24 第二个特性的发现结果

在“Value”栏中读到了一组数据“02 28 00 F2 FF”。下面来分析一下得到的这一组数据的含义。其中，第一个字节“02”，按照蓝牙协议，表示只读属性；第二、三字节“2800”表示这个特性的句柄；最后两个字节“F2FF”为第二个特性的UUID。

7.3.5 通过句柄读取多个特性值

通过一次读请求可以获得多个特性值，只要知道这些特性值的句柄，并且这些特性值具有可读属性。首先使用上一小节讲述的方法获取属性表中第一个特性值和第二个特性值的句柄，分别为“0x0025”和“0x0028”，在“Read/Write”选项卡的“Characteristic Read”栏中选择“Read Multiple Characteristic Values”，在“Characteristic Value Handle”栏中输入第一个特性值和第二个特性值的句柄“0x0025;0x0028”，然后单击“Read”按钮，如图7-25所示。

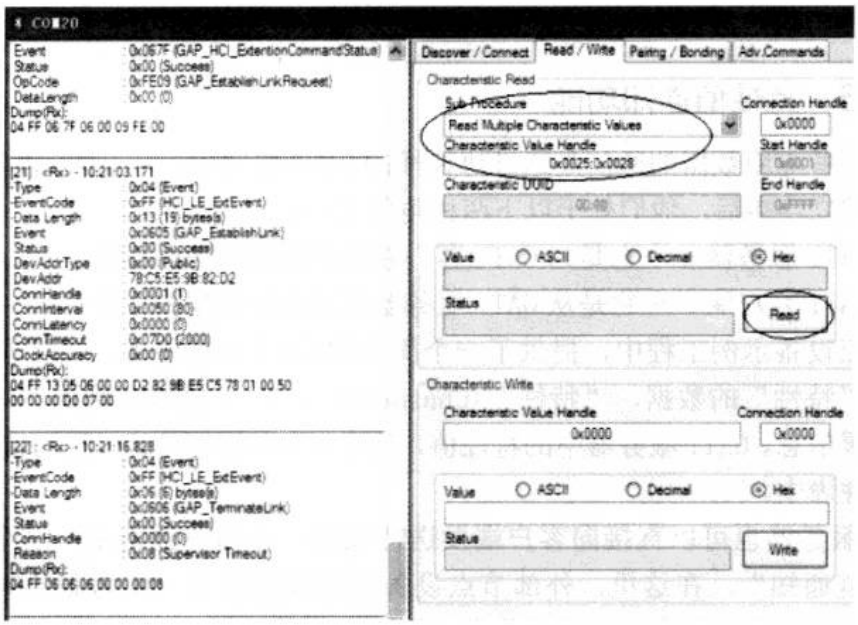


图 7-25　同时读取两个特性值

在“Value”栏中显示了两个特性值“01 02”。其中，“01”为第一个特性的值，“02”为第二个特性的值，如图 7-26 所示。

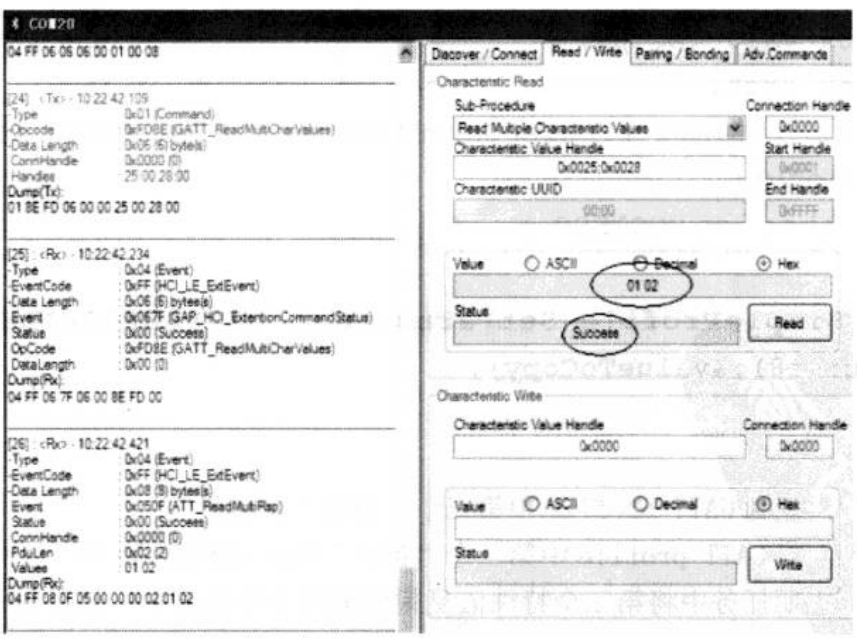


图 7-26　同时读取两个特性值

读取多个特性值的数据包会通过 USB Dongle 板发送到 CC2540 蓝牙节点设备，通过提供的句柄得到属性表中的特性值，返回的数据包大小必须是固定而且已知的，

以供客户端（此处为PC端）进行数据解析。

7.3.6 特性值通知功能

两个设备应用数据的通信是通过协议栈的GATT层实现。从GATT角度来看，当两个设备建立连接后，他们处于以下两种角色之一：

▲ GATT服务器——它是为GATT客户端提供数据服务的设备；

▲ GATT客户端——它是从GATT服务器读写应用数据的设备。

在节点设备示例工程中，提供了一个简单的GATT数据服务，GATT服务包含一些称之为“特性”的数据，“特性”(Characteristics)是服务用到的值，以及其内容和配置信息。GATT服务器中的特性值，及其内容和配置信息（称为特性描述符）存储于属性表中。

GATT服务器也可以直接向客户端发送特性值，不需要读请求，这个过程称之为“特性值通知”。在这里，外部节点设备提供数据服务，作为服务器；PC端通过BToll应用来读写节点设备中的数据，作为客户端。

节点设备示例工程的GATT服务中，定义了第三个特性值具有只写的属性，第四个特性值具有只通知的属性。下面先来回顾一下节点设备示例工程中周期任务performPeriodicTask要执行的具体任务。

```
static void performPeriodicTask( void )
{
 uint8 valueToCopy;
 uint8 stat;
1 stat = SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR3,
&valueToCopy);
 if( stat == SUCCESS )
 {
2   SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR4,
sizeof(uint8),&valueToCopy);
  }
}
```

第1行，复制GATT profile中第三个“特性”的值。

第2行，将GATT profile中第三个“特性”的值复制到第四个“特性”中。

在上述周期任务中将第三个特性值复制到第四个特性值中，因为第四个特性值具有通知的属性，所以每次这个特性值被更新时，应用都会检查它是否使能了通知功能，如果已经使能了通知功能，应用会向客户端设备发送一个通知，不需要客户端发送读请求，就可以将第四个特性值发送给客户端设备。

使能第四个特性值的通知功能，需要向第四个特性值通知功能的句柄写入“0x0001”，那么，第四个特性值通知功能的句柄怎样获得呢？首先要找到第四个

特性值的句柄，然后将这个句柄值加 1，即为第四个特性值通知功能的句柄。

在第 7.3.4 节讲述了使用 UUID 发现特性，获取特性值句柄的方法，使用 UUID 获取第四个特性值的句柄，如图 7-27 所示。

图 7-27　获取第四个特性值的句柄

得到第四个特性值的句柄“0x002E”，那么，第四个特性值通知功能的句柄即为“0x002F”。只需要向句柄“0x002F”中写入“0x0001”，就可以使能第四个特性值的通知功能。在 BToll 中的“Characteristic Write”区域可以找到“Characteristic Value Handle”一栏，向其中写入句柄“0x002F”，然后向“Value”栏中写入“01:00”，单击“Write”按钮，如图 7-28 所示。

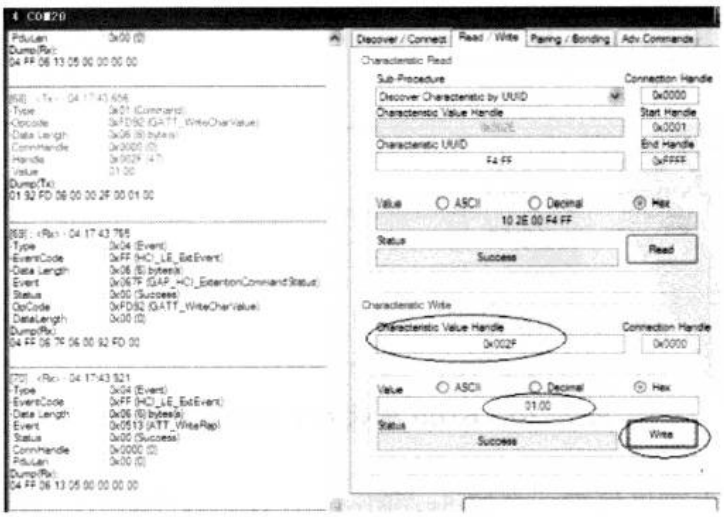


图 7-28　使能通知功能

此时，可从 BToll 软件界面观察到 BToll 发送给节点设备的数据和节点设备返

回的数据，如图 7-29 所示。

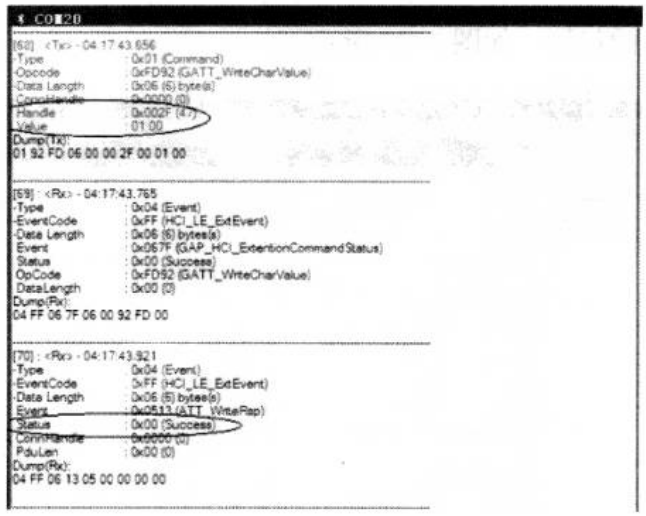


图 7-29　BToll 发送与接收的数据

使能通知之后，每当有数据写入第四个特性值中，应用都会发送通知给 BToll。前面提到，节点设备示例工程中的周期任务为：每隔 5s，将第三个特性值复制到第四个特性值中。那么，每隔 5s，BToll 将会收到节点设备发送来的通知，即为第四个特性值，如图 7-30 所示。

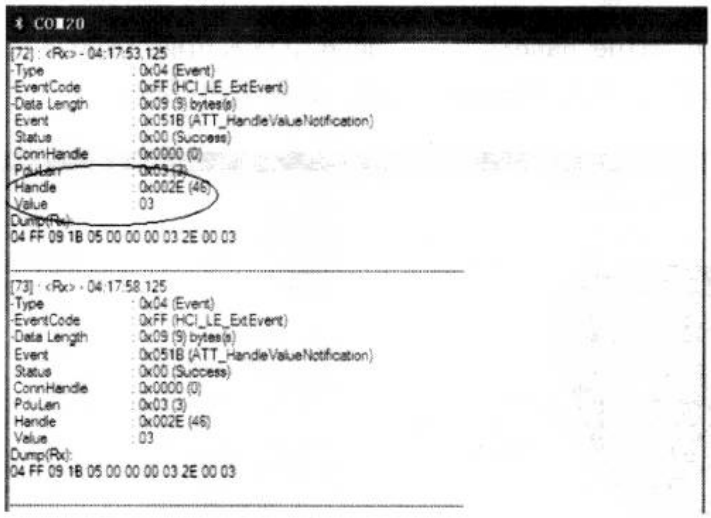


图 7-30　BToll 接收到节点设备的通知

从图 7-30 中可以发现，每隔 5s，BToll 收到第四个特性的值“03”，这是因为在节点设备示例工程提供的 GATT 服务中，定义了第三个特性的值为“03”，而在周期任务中又将第三个特性值复制到第四个特性值中。

7.4 自定义 GATT 数据服务

在节点设备示例工程中，提供了一个简单的 GATT 服务，GATT 服务包含一些称之为“特性”的数据，“特性”(Characteristics) 是服务用到的值，以及其内容和配置信息，存储于属性表中。读者可以在此基础上添加新的自定义的“特性”来满足特定的需求。本节是在节点设备示例工程的基础上，向 GATT 服务的属性表中添加一个具有读写属性的“特性”，预期实现的功能为：通过 BToll 向节点设备的这个“特性”中写入不同的数据，来控制节点设备 LED 灯的亮灭，并通过串口输出这个“特性”的当前值。

7.4.1 节点设备编程

节点设备示例工程的 PROFILE 文件夹中有两个源文件“simpleGATTprofile.c”和“simpleGATTprofile.h”，“特性”及属性表就是在这两个源文件中定义的，如图 7-31 所示。

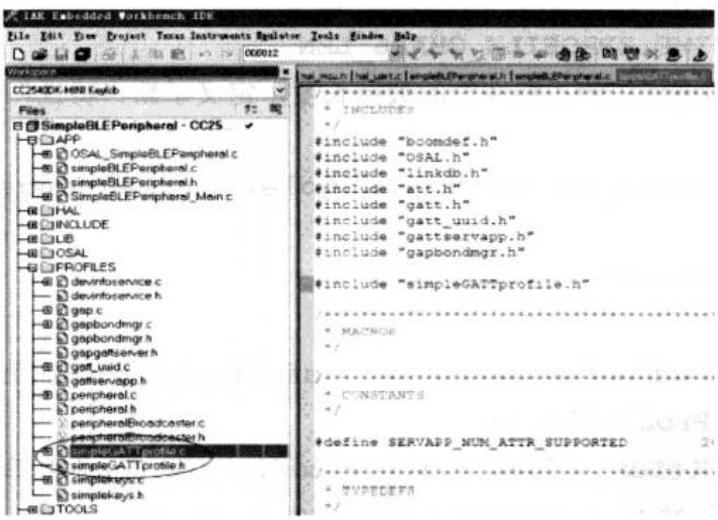


图 7-31 节点设备示例工程

修改“simpleGATTprofile.h”程序如下（新增加的内容以加粗字体显示）：

```
#ifndef SIMPLEGATTPROFILE_H
#define SIMPLEGATTPROFILE_H
```

上述代码主要为了解决重复包含问题，这是模块化编程的技巧。

```
#ifdef __cplusplus
extern "C"
{
```

```
#endif
#define SIMPLEPROFILE_CHAR1                          0
#define SIMPLEPROFILE_CHAR2                          1
#define SIMPLEPROFILE_CHAR3                          2
#define SIMPLEPROFILE_CHAR4                          3
#define SIMPLEPROFILE_CHAR5                          4
#define SIMPLEPROFILE_CHAR6                          5
#define SIMPLEPROFILE_SERV_UUID                   0xFFF0
#define SIMPLEPROFILE_CHAR1_UUID                  0xFFF1
#define SIMPLEPROFILE_CHAR2_UUID                  0xFFF2
#define SIMPLEPROFILE_CHAR3_UUID                  0xFFF3
#define SIMPLEPROFILE_CHAR4_UUID                  0xFFF4
#define SIMPLEPROFILE_CHAR5_UUID                  0xFFF5
#define SIMPLEPROFILE_CHAR6_UUID                  0xFFF6
```

上面代码为添加新的“特性”的UUID。

```
#define SIMPLEPROFILE_SERVICE                0x00000001
#define SIMPLEPROFILE_CHAR5_LEN                       5
#define SIMPLEPROFILE_CHAR6_LEN                       7
```

上面代码为添加新的“特性”的数据长度，定义了新“特性”的数据长度为7个字节。

```
typedef NULL_OK void (*simpleProfileChange_t)( uint8 paramID );
typedef struct
{
  simpleProfileChange_t       pfnSimpleProfileChange;
} simpleProfileCBs_t;
extern bStatus_t SimpleProfile_AddService( uint32 services );
extern bStatus_t SimpleProfile_RegisterAppCBs( simpleProfileCBs_t *appCallbacks );
extern bStatus_t SimpleProfile_SetParameter( uint8 param, uint8 len, void *value );
extern bStatus_t SimpleProfile_SetParameter( uint8 param, uint8 len, void *value );
#ifdef __cplusplus
}
#endif
#endif
```

修改“simpleGATTprofile.c”程序如下（修改的部分以加粗字体显示）：

```
#include "bcomdef.h"
#include "OSAL.h"
#include "linkdb.h"
#include "att.h"
#include "gatt.h"
#include "gatt_uuid.h"
#include "gattservapp.h"
#include "gapbondmgr.h"
#include "simpleGATTprofile.h"
#define SERVAPP_NUM_ATTR_SUPPORTED        20
CONST uint8 simpleProfileServUUID[ATT_BT_UUID_SIZE] =
{
  LO_UINT16(SIMPLEPROFILE_SERV_UUID), HI_
UINT16(SIMPLEPROFILE_SERV_UUID)
};
CONST uint8 simpleProfilechar1UUID[ATT_BT_UUID_SIZE] =
{
LO_UINT16(SIMPLEPROFILE_CHAR1_UUID), HI_
UINT16(SIMPLEPROFILE_CHAR1_UUID)
};
......
CONST uint8 simpleProfilechar6UUID[ATT_BT_UUID_SIZE] =
{
LO_UINT16(SIMPLEPROFILE_CHAR6_UUID), HI_
UINT16(SIMPLEPROFILE_CHAR6_UUID)
};
static simpleProfileCBs_t *simpleProfile_AppCBs = NULL;
static CONST gattAttrType_t simpleProfileService = { ATT_
BT_UUID_SIZE,
simpleProfileServUUID };
static uint8 simpleProfileChar1Props = GATT_PROP_READ |
GATT_PROP_WRITE;
static uint8 simpleProfileChar1 = 0;
static uint8 simpleProfileChar1UserDesp[17] =
"Characteristic 1\0";
......
```

```
static uint8 simpleProfileChar6Props = GATT_PROP_READ |
GATT_PROP_WRITE;
static uint8 simpleProfileChar6[SIMPLEPROFILE_CHAR6_LEN] =
{ 0, 0, 0, 0, 0 ,0 ,0};
static uint8 simpleProfileChar6UserDesp[17] =
"Characteristic 6\0";
```

上述代码为自定义的“特性”的权限、特性值以及用户描述符。

```
static gattAttribute_t simpleProfileAttrTbl[SERVAPP_NUM_
ATTR_SUPPORTED] =
{
  // 主服务
  {
    { ATT_BT_UUID_SIZE, primaryServiceUUID },      // 类型
    GATT_PERMIT_READ,      // 权限
    0,      // 句柄
    (uint8 *)&simpleProfileService      // 值
  },
// 特性 1 的声明
{
  { ATT_BT_UUID_SIZE, characterUUID },
  GATT_PERMIT_READ,
  0,
  &simpleProfileChar1Props
},
// 特性 1 的值
{
{ ATT_BT_UUID_SIZE, simpleProfilechar1UUID },
GATT_PERMIT_READ | GATT_PERMIT_WRITE,
0,
&simpleProfileChar1
},
// 特性 1 的用户描述符
{
  { ATT_BT_UUID_SIZE, charUserDescUUID },
  GATT_PERMIT_READ,
  0,
  simpleProfileChar1UserDesp
```

```
},
  ......
//特性 6 的声明
{
 { ATT_BT_UUID_SIZE, characterUUID },
 GATT_PERMIT_READ,
 0,
 &simpleProfileChar6Props
},
// 特性 6 的值
{
{ ATT_BT_UUID_SIZE, simpleProfilechar6UUID },
GATT_PERMIT_READ | GATT_PERMIT_WRITE,
0,
simpleProfileChar6      //注意不能写为 &simpleProfileChar6
},
//特性 6 的用户描述符
{
  { ATT_BT_UUID_SIZE, charUserDescUUID },
  GATT_PERMIT_READ,
  0,
  simpleProfileChar6UserDesp
},
};
```

上述代码即为 GATT 服务的属性表，在其中定义了主服务和 6 个“特性”。

```
......
CONST gattServiceCBs_t simpleProfileCBs =
{
  simpleProfile_ReadAttrCB,
  simpleProfile_WriteAttrCB,
  NULL
};
```

上述代码为 GATT 服务数据读写的回调函数。

```
......
bStatus_t SimpleProfile_SetParameter( uint8 param, uint8
len, void *value )
{
```

```
    bStatus_t ret = SUCCESS;
    switch ( param )
    {
      case SIMPLEPROFILE_CHAR1:
        if ( len == sizeof ( uint8 ) )
        {
          simpleProfileChar1 = *((uint8*)value);
        }
        else
        {
          ret = bleInvalidRange;
        }
        break;
        ……
        case SIMPLEPROFILE_CHAR6:
        if ( len == SIMPLEPROFILE_CHAR6_LEN )
        {
          VOID osal_memcpy( simpleProfileChar6, value,
SIMPLEPROFILE_CHAR6_LEN );
        }
        else
        {
          ret = bleInvalidRange;
        }
        break;
      default:
        ret = INVALIDPARAMETER;
        break;
    }
    return ( ret );
  }
```

上述代码为向“特性”中写入新的数据，通过 switch-case 语句来判断向哪个“特性”中写入数据。只有符合“特性”的数据长度的值才能被正确写入，否则无法写入数据。

```
  bStatus_t SimpleProfile_GetParameter( uint8 param, void
*value )
  {
```

```
    bStatus_t ret = SUCCESS;
    switch ( param )
    {
       case SIMPLEPROFILE_CHAR1:
       *((uint8*)value) = simpleProfileChar1;
       break;
      ......
    case SIMPLEPROFILE_CHAR6:
      VOID osal_memcpy( value, simpleProfileChar6,
SIMPLEPROFILE_CHAR6_LEN );
      break;
    default:
      ret = INVALIDPARAMETER;
      break;
    }
    return ( ret );
  }
```

上述代码为将“特性”的值复制给一个临时变量或缓冲区，通过 switch-case 语句来判断获取哪个“特性”中的数据。

```
  static uint8 simpleProfile_ReadAttrCB( uint16 connHandle,
gattAttribute_t *pAttr,
  uint8 *pValue, uint8 *pLen, uint16 offset, uint8 maxLen )
  {
    bStatus_t status = SUCCESS;
      if ( gattPermitAuthorRead( pAttr->permissions ) )
      {
        return ( ATT_ERR_INSUFFICIENT_AUTHOR );
      }
      if ( offset > 0 )
      {
  return ( ATT_ERR_ATTR_NOT_LONG );
  }
  if ( pAttr->type.len == ATT_BT_UUID_SIZE )
  {
    uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr-
>type.uuid[1]);
    switch ( uuid )
```

```
    {
      case SIMPLEPROFILE_CHAR1_UUID:
      case SIMPLEPROFILE_CHAR2_UUID:
      case SIMPLEPROFILE_CHAR4_UUID:
      *pLen = 1;
      pValue[0] = *pAttr->pValue;
      break;
      case SIMPLEPROFILE_CHAR5_UUID:
      *pLen = SIMPLEPROFILE_CHAR5_LEN;
       VOID osal_memcpy( pValue, pAttr->pValue,
SIMPLEPROFILE_CHAR5_LEN );
      break;
      case SIMPLEPROFILE_CHAR6_UUID:
        *pLen = SIMPLEPROFILE_CHAR6_LEN;
        VOID osal_memcpy( pValue, pAttr->pValue,
        SIMPLEPROFILE_CHAR6_LEN );
        break;
      default:
        *pLen = 0;
        status = ATT_ERR_ATTR_NOT_FOUND;
        break;
      }
    }
    else
    {
      *pLen = 0;
      status = ATT_ERR_INVALID_HANDLE;
    }
    return ( status );
  }
```

上述代码为读特性的回调函数，其中特性1和特性2具有读属性，新添加一个具有读写属性的特性6。回调函数最后返回是否读取成功。

```
  static bStatus_t simpleProfile_WriteAttrCB( uint16
connHandle, gattAttribute_t *pAttr,
  uint8 *pValue, uint8 len, uint16 offset )
  {
    bStatus_t status = SUCCESS;
```

```
    uint8 notifyApp = 0xFF;
    if ( gattPermitAuthorWrite( pAttr->permissions ) )
    {
    return ( ATT_ERR_INSUFFICIENT_AUTHOR );
    }
    if ( pAttr->type.len == ATT_BT_UUID_SIZE )
    {
      uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0],
pAttr->type.uuid[1]);
      switch ( uuid )
      {
        ......
        case SIMPLEPROFILE_CHAR6_UUID:
        if ( offset == 0 )
        {
          if ( len != SIMPLEPROFILE_CHAR6_LEN )
          {
            status = ATT_ERR_INVALID_VALUE_SIZE;
          }
        }
        else
        {
          status = ATT_ERR_ATTR_NOT_LONG;
        }
        if ( status == SUCCESS )
        {
      VOID osal_memcpy( pAttr->pValue, pValue,
      SIMPLEPROFILE_CHAR6_LEN );
          notifyApp = SIMPLEPROFILE_CHAR6;
      }
      break;
   case GATT_CLIENT_CHAR_CFG_UUID:
    status = GATTServApp_ProcessCCCWriteReq( connHandle,
pAttr, pValue, len,
    offset, GATT_CLIENT_CFG_NOTIFY );
      break;
      default:
```

```
        status = ATT_ERR_ATTR_NOT_FOUND;
        break;
      }
    }
    else
    {
      status = ATT_ERR_INVALID_HANDLE;
    }
    if ( (notifyApp != 0xFF ) && simpleProfile_AppCBs &&
simpleProfile_AppCBs->pfnSimpleProfileChange )
    {
      simpleProfile_AppCBs->pfnSimpleProfileChange(notifyApp );
    }
    return ( status );
  }
```

上述代码为写特性的回调函数，其中特性1和特性3具有写属性，新添加一个具有读写属性的特性6。回调函数最后返回是否写入成功。

修改“SimpleBLEPeripheral.c”文件的内容如下（新增加的部分以加粗字体显示）：

```
#include "hal_uart.h"
```

向SimpleBLEPeripheral_Init( )函数中添加：

```
1  halUARTCfg_t uartConfig;
2  uartConfig.configured      = TRUE;
3  uartConfig.baudRate      = HAL_UART_BR_115200;
4  uartConfig.flowControl   = FALSE;
5  HalUARTOpen (0, &uartConfig);
```

第1～4行，配置串口参数，包括串口波特率、是否使用流控等。

第5行，初始化串口。

蓝牙4.0 BLE协议栈中对串口的配置是使用一个结构体来实现的，该结构体为halUARTCfg_t，在此不必关心该结构体的具体定义形式，只需要对其功能有个了解，该结构体将串口初始化有关的参数集合在了一起，例如：波特率、是否打开串口、是否使用流控等，读者只需要将各个参数初始化就可以。

最后使用HalUARTOpen()函数对串口进行初始化，注意，该函数将halUARTCfg_t类型的结构体变量作为参数，因为halUARTCfg_t类型的结构体变量已经包含了串口初始化相关的参数，所以，将这些参数传递给HalUARTOpen()函数，HalUARTOpen()函数使用这些参数对串口进行了初始化。

```
{
```

```
    uint8 charValue1 = 1;
    uint8 charValue2 = 2;
    uint8 charValue3 = 3;
    uint8 charValue4 = 4;
    uint8 charValue5[SIMPLEPROFILE_CHAR5_LEN] = { 1, 2, 3,
4, 5 };
    uint8 charValue6[SIMPLEPROFILE_CHAR6_LEN] = { 1, 2, 3,
4, 5, 6, 7 };
  SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR1, sizeof
( uint8 ),&charValue1 );
  SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR2, sizeof
( uint8 ),&charValue2 );
  SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR3, sizeof
( uint8 ),&charValue3 );
  SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR4, sizeof
( uint8 ),&charValue4 );
  SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR5,
  SIMPLEPROFILE_CHAR5_LEN, charValue5 );
  SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR6,
  SIMPLEPROFILE_CHAR6_LEN, charValue6 );
  }
```

上述代码为在初始化函数中给 6 个特性值赋初值。

```
  static void simpleProfileChangeCB( uint8 paramID )
  {
    uint8 newValue;
    uint8 buf[7];
    switch( paramID )
    {
      case SIMPLEPROFILE_CHAR1:
       SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR1,
&newValue );
      break;
    case SIMPLEPROFILE_CHAR3:
       SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR3,
&newValue );
      break;
  case SIMPLEPROFILE_CHAR6:
```

```
1  SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR6, &buf);
2   if( osal_memcmp(buf, "LED_ONN", SIMPLEPROFILE_CHAR6_
LEN) )
      {
3       HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
4       HalUARTWrite(0, buf,7) ;
5       osal_memset( buf,0,7);
      }
6   if( osal_memcmp(buf, "LED_OFF", SIMPLEPROFILE_CHAR6_
LEN) )
      {
7    HalLedSet(HAL_LED_1, HAL_LED_MODE_OFF);
8       HalUARTWrite(0, buf,7) ;
9       osal_memset( buf,0,7);
      }
      break;
    default:
      break;
    }
  }
```

第1行，将第六个特性值复制到前面定义的buf中。

第2行，比较此时buf中的内容是否为“LED_ONN”，若是，则调用相应的函数处理。

第3行，调用HAL层驱动，点亮LED灯。

第4行，将buf中的内容写入串口，在PC端串口界面显示，显示的内容即为“LED_ONN”。

第5行，清空buf，防止PC端串口界面一直接收数据。

第6行，比较此时buf中的内容是否为“LED_OFF”，若是，则调用相应的函数处理。

第7行，调用HAL层驱动，熄灭LED灯。

第8行，将buf中的内容写入串口，在PC端串口界面显示，显示的内容即为“LED_OFF”。

第9行，清空buf，防止PC端串口界面一直接收数据。

7.4.2 实例测试

修改好工程文件后，编译、下载到开发板，并建立开发板与PC端的串口连接，打开PC端的串口调试助手，设置串口波特率为115200。将USB Dongle与PC端连

接，打开 BToll 调试工具，同样将波特率设置为 115200，建立 BToll 与节点设备的连接。想要向第六个“特性”中写入数据，首先要使用 UUID 找到这个“特性”值的句柄，在 BToll 的“Read/Write”选项卡下的特性读取区域选择“Read Using Characteristic UUID”，向“Characteristic UUID”栏中写入“F6:FF”，并单击“Read”按钮。因为在属性表中定义了第六个特性值为可读写属性，读取成功后，会在“Characteristic Value Handle”栏中显示第六个特性值的句柄“0x0035”，在“Value”栏中显示特性值，在“Status”状态栏中显示“Success”，如图 7-32 所示。

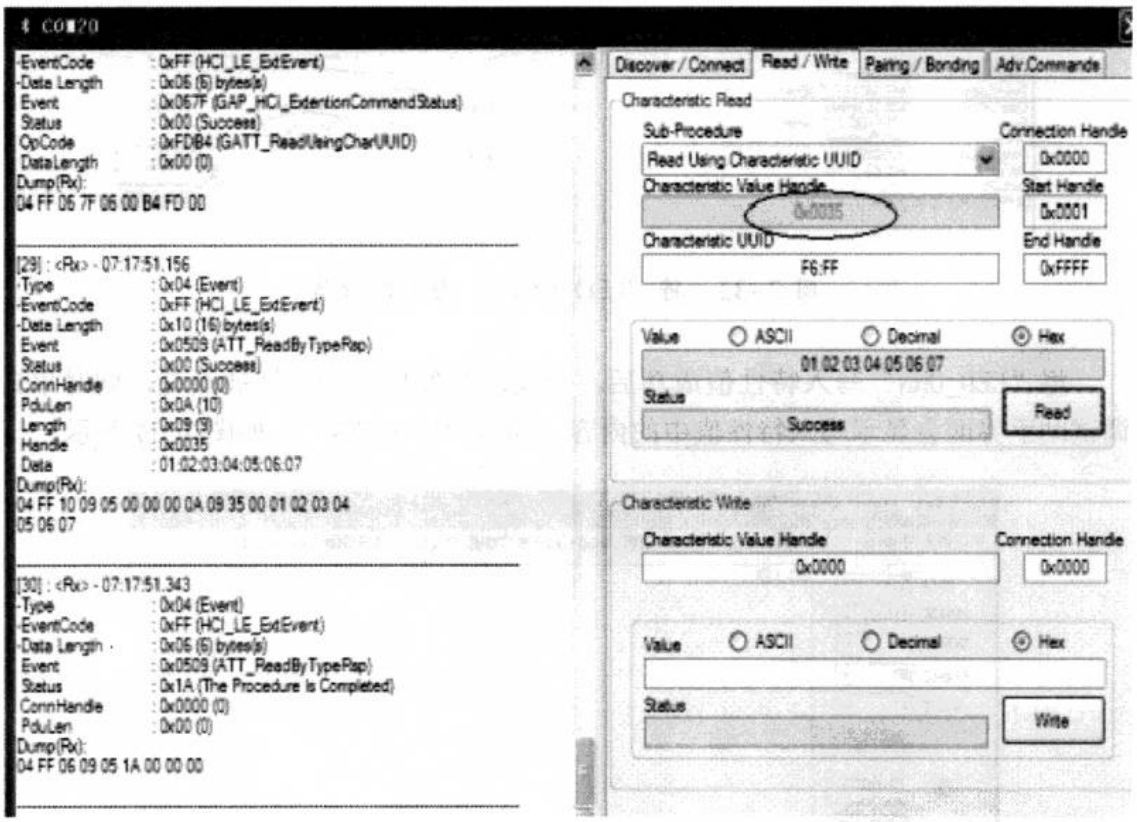


图 7-32 获得特性值的句柄

向“Characteristic Write”区域中的“Characteristic Value Handle”一栏中输入第六个特性值的句柄“0x0035”，向“Value”栏中输入 ASCII 格式的字符串“LED_ONN”，然后单击“Write”按钮，如图 7-33 所示。

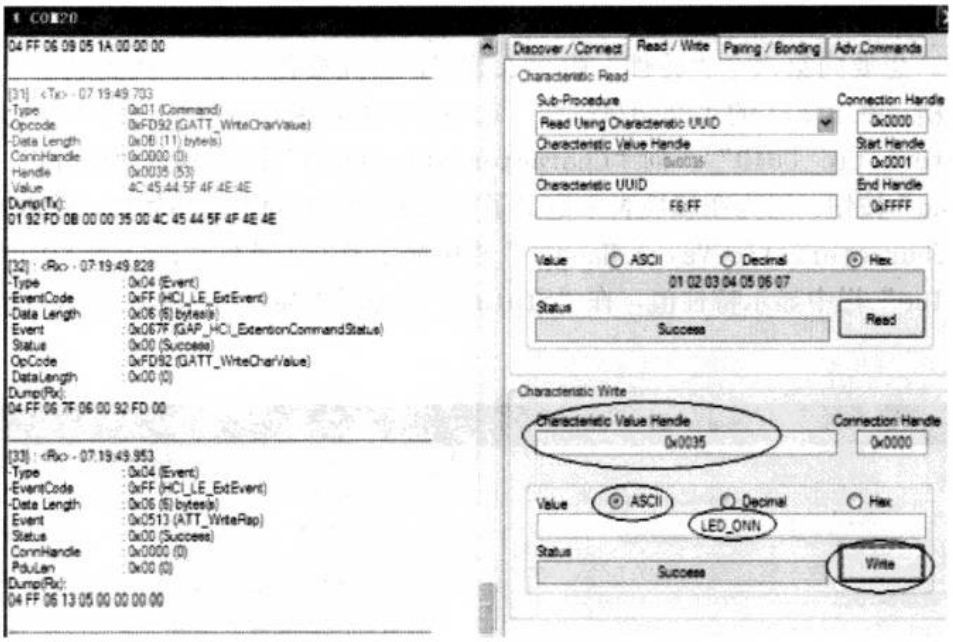


图 7-33　将“LED_ONN”写入特性值

将“LED_ONN”写入特性值成功后，会发现开发板上LED灯被点亮，同时串口调试助手界面会显示写入特性值中的内容，即为“LED_ONN”，如图7-34所示。

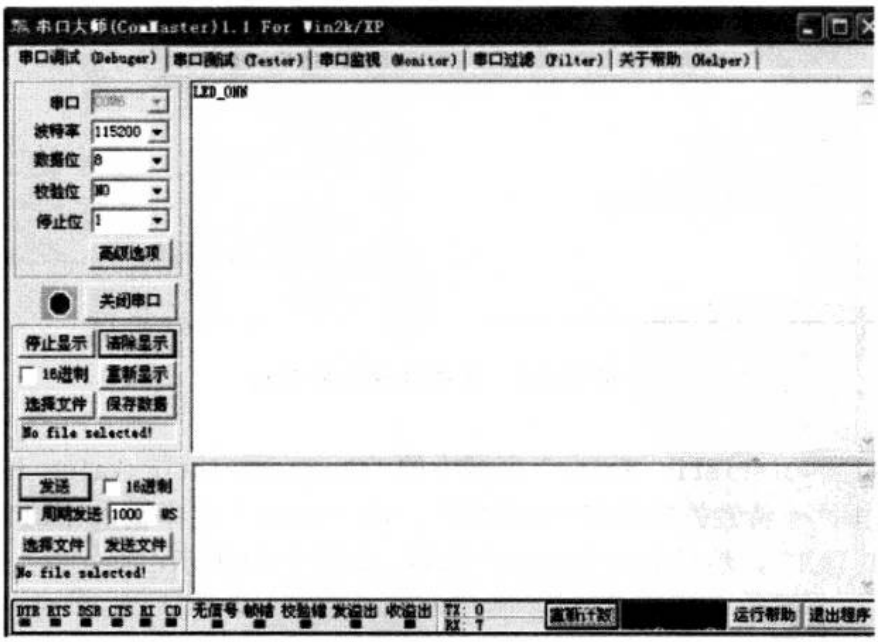


图 7-34　串口调试助手界面

此时，若单击BTo11特性读取区域的“Read”按钮，会得到第六个特性值中新写入的数据“LED_ONN”，如图7-35所示。

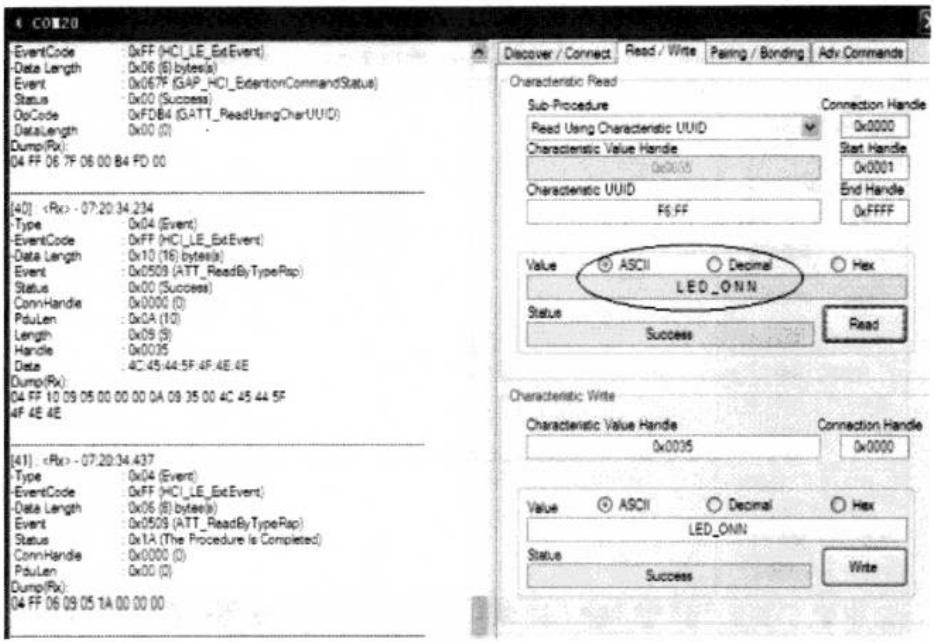


图 7-35　读取特性值

向“Characteristic Write”区域中的“Characteristic Value Handle”一栏中输入第六个特性值的句柄“0x0035”，向“Value”栏中输入 ASCII 格式的字符串“LED_OFF”，然后单击“Write”按钮，如图 7-36 所示。

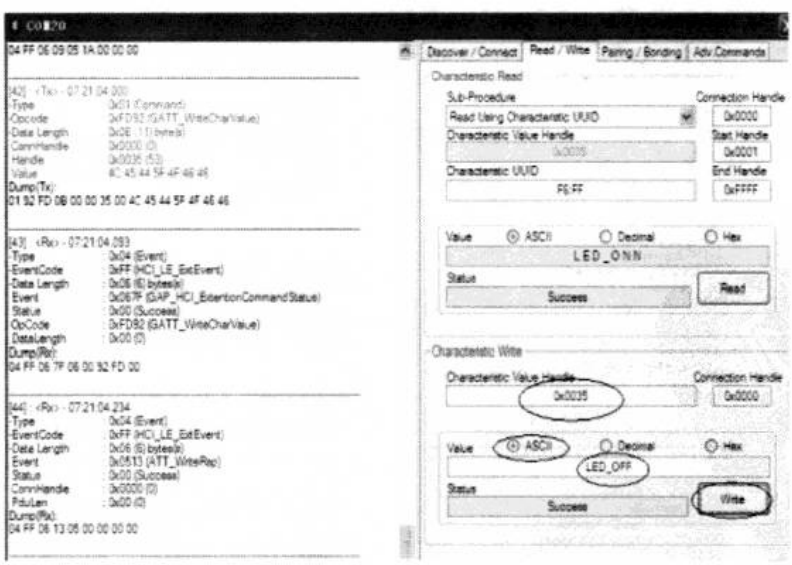


图 7-36　将“LED_OFF”写入特性值

将“LED_OFF”写入特性值成功后，会发现开发板上 LED 灯熄灭，同时串口调试助手界面会显示写入特性值中的内容，即为“LED_OFF”。如图 7-37 所示。

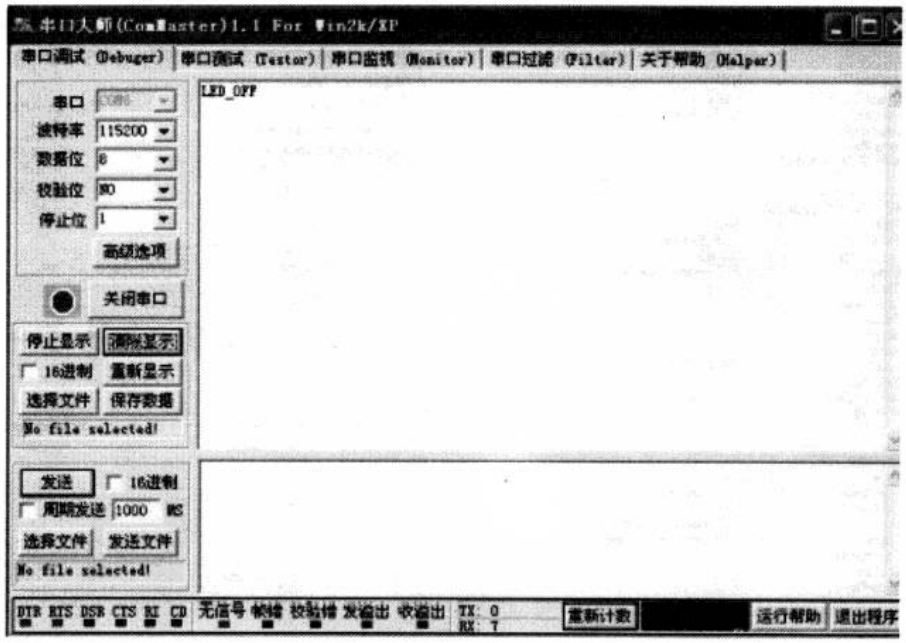


图 7-37　串口调试助手界面

此时，若单击 BTo11 特性读取区域的“Read”按钮，会得到第六个特性值中新写入的数据“LED_OFF”，如图 7-38 所示。

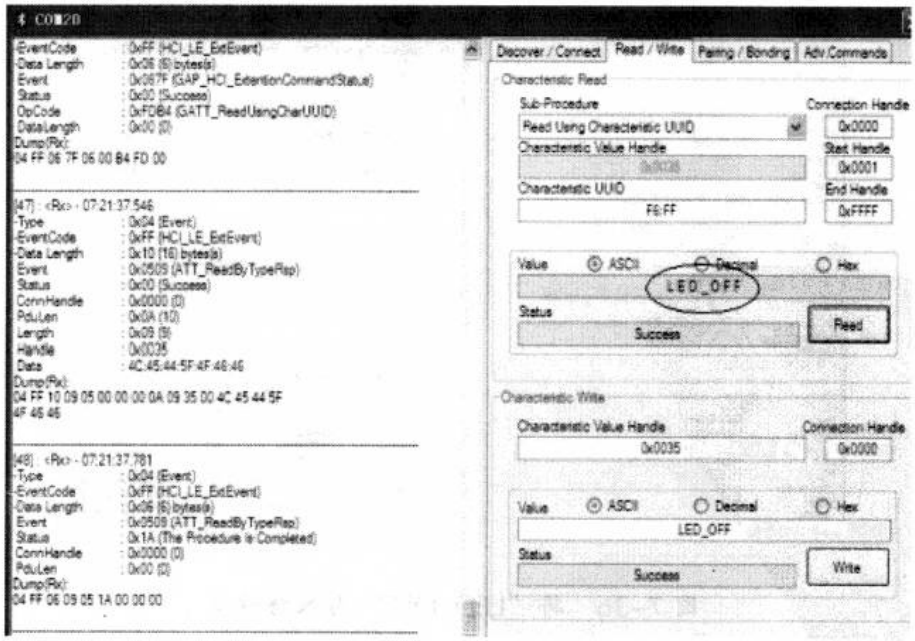


图 7-38　读取特性值

7.5 本章小结

本章主要讲述了使用 PC 端调试开发工具 BTo11 来进行蓝牙外设应用程序的开发调试，详细阐述了如何通过“特性”的 UUID 进行“特性”的发现，以及如何读写蓝牙节点设备中的应用数据。在本章的第 7.4 节中，通过一个实例向读者展示了如何创建一个自定义的“特性”来进行应用的开发。书中给出的实例为通过向“特性”中写入特定的应用数据来控制节点设备 LED 灯的亮灭以及串口输出的数据，读者只需简单修改程序就可以实现自己需要的的功能。

[扩展阅读] GATT 数据服务的通知与指示

首先先来回顾一下，从 GATT 层的角度来看，两个进行应用数据通信的设备所充当的角色。当两个设备建立连接后，它们处于以下两种角色之一：

▲ GATT 服务器——它是为 GATT 客户端提供数据服务的设备；

▲ GATT 客户端——它是从 GATT 服务器读写应用数据的设备。

GATT 服务包含一些称之为“特性”的数据，“特性”(Characteristics) 是服务用到的值，以及其内容和配置信息。GATT 服务器中的特性值，及其内容和配置信息（称为特性描述符）存储于属性表中。

蓝牙 4.0 BLE 协议栈通过两种方式传递应用数据，第一种方式是客户端设备（client）发送读写请求，服务器设备（server）响应请求并向客户端发送特性值中包含的应用数据，相信读者通过前面的学习，对这种方式已经很熟悉了；第二种方式是服务器设备使用通知或指示功能，将应用数据传递给客户端设备。

不经过客户端的读请求，GATT 服务器可以直接向客户端发送特性值，这个过程可使用通知或指示功能来完成。

▲ 通知——服务器被配置为将特性值通知给客户端，不需要属性协议层确认特性值是否被成功接收；

▲ 指示——服务器被配置为将特性值指示给客户端，属性协议层必须确认特性值被成功接收，才能进行下一步的数据发送。

通知与指示功能的流程如图 7-39 所示。

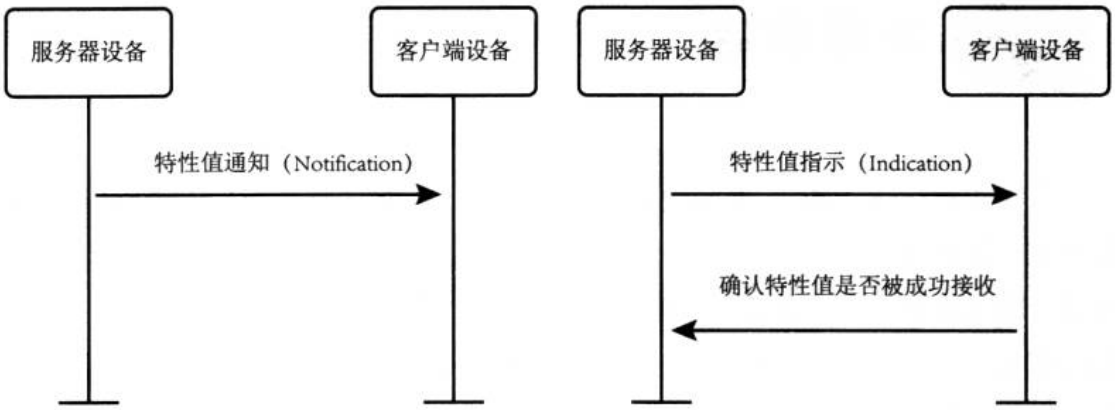
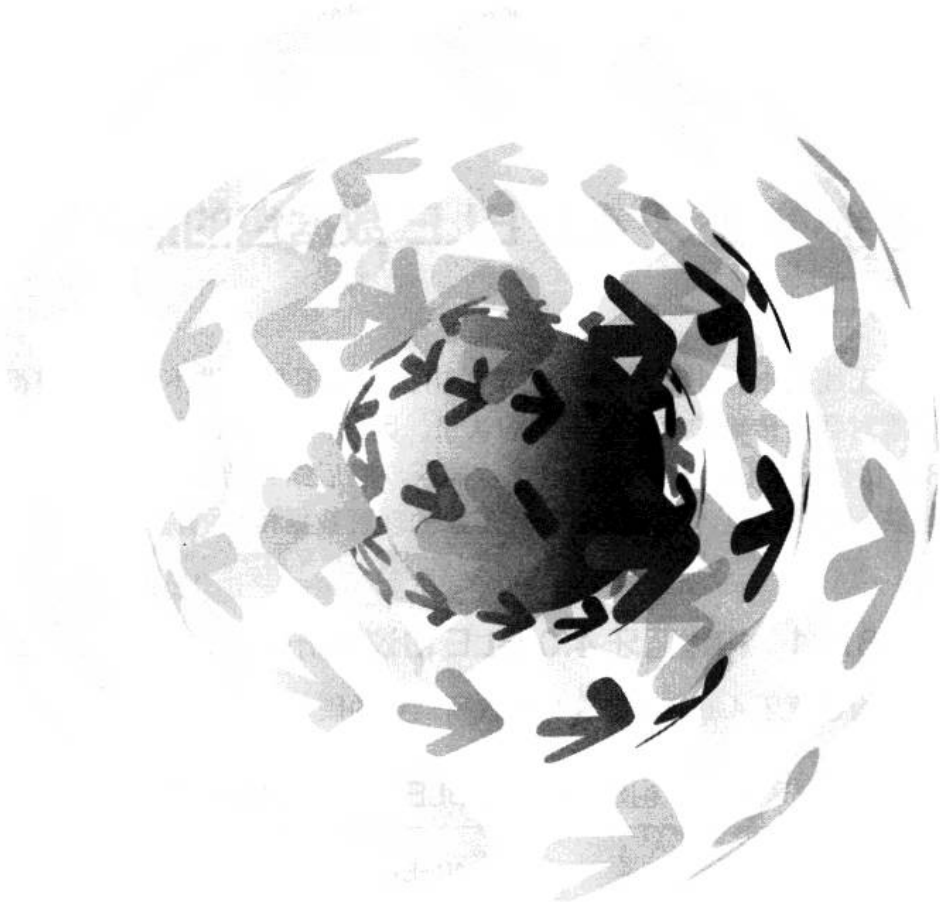


图 7-39　通知与指示

何时发送通知或指示的条件可以在配置文件中设置，也可以通过应用来设置。要想使能通知和指示功能，需要分别在相应的句柄处写入“0x0001”和“0x0002”，如表 7-1 所示。

表 7-1　通知和指示的使能值

配置	值	描述
通知（Notification）	0x0001	使能特性值通知
指示（Indication）	0x0002	使能特性值指示



第 8 章

蓝牙 4.0 BLE 无线数据传输分析

在第 6 章中讲述了节点设备示例工程和集中器设备示例工程，这两个示例工程实现了两个设备之间点对点的数据通信，相信读者经过前面实例的学习，已经对蓝牙无线数据传输的流程有了初步认识，本章将对无线数据传输的具体流程展开深层的剖析。

8.1 蓝牙 4.0 BLE 数据包的捕获

无线数据以电磁波的形式发射出去，看不见，摸不着，这使得读者在初学时很难对数据传输的流程有直观的认识。因此，笔者推荐读者可以自己构建蓝牙 4.0 BLE 协议分析仪，对无线数据进行捕获，然后结合前面章节讲述的示例工程来分析捕获到的数据包，这样就可以更形象的理解数据传输的流程。

8.1.1 构建蓝牙 4.0 BLE 协议分析仪

构建蓝牙 4.0 BLE 协议分析仪所需要的硬件和软件如表 8-1 所示。

表 8-1 构建蓝牙 4.0 BLE 协议分析仪所需要的硬件和软件

硬件	CC2540-EB 开发板（或 USB Dongle）
	仿真下载器 Usb Debug Adapter
软件	Texas Instruments Packet Sniffer 软件

构建蓝牙 4.0 BLE 协议分析仪的具体步骤如下。

首先，将仿真下载器 Usb Debug Adapter 与 CC2540-EB 开发板（也可用 USB Dongle 替代）连接起来，向 CC2540-EB 开发板中写入固件“sniffer_fw_cc2540_usb.hex”，固件的路径为 C:\Program Files\Texas Instruments\SmartRF Tools\Packet Sniffer\bin\general\firmware。

然后，打开 Texas Instruments Packet Sniffer 软件如图 8-1 所示，在下拉列表框中选择 Bluetooth Low Energy，单击 Start 按钮。

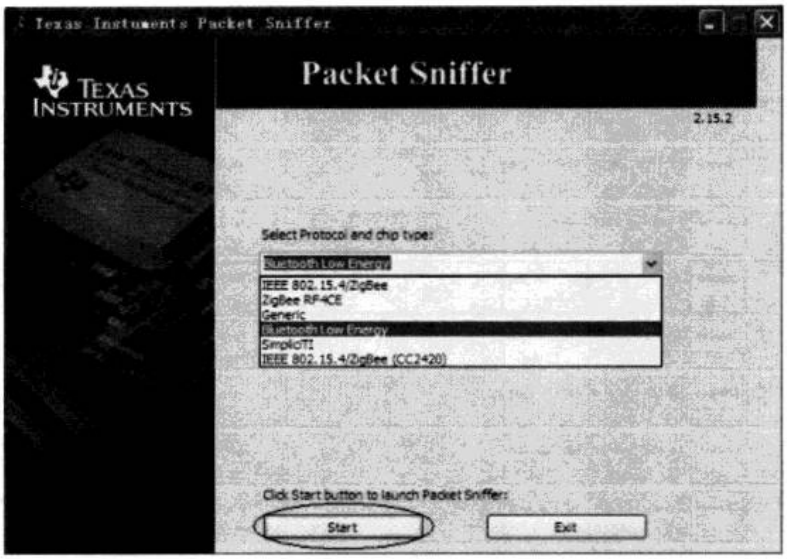


图 8-1　Texas Instruments Packet Sniffer 软件界面

此时，会弹出 Texas Instruments Packet Sniffer 主窗口如图 8-2 所示，在窗口底部 Select capturing device 中已经发现了开发板，选择 CC2540 USB Dongle[USB Device]，然后单击蓝色小三角按钮（开始抓包按钮）即可进行数据包的捕获。

图 8-2　Texas Instruments Packet Sniffer 主窗口

现在就可以捕获节点设备和集中器设备之间传输的数据包了，依次连接节点设备电源和集中器设备电源，若此时节点设备处于广告状态，Texas Instruments Packet Sniffer 软件就可以显示所捕获的广告中包含的数据，如图 8-3 所示。

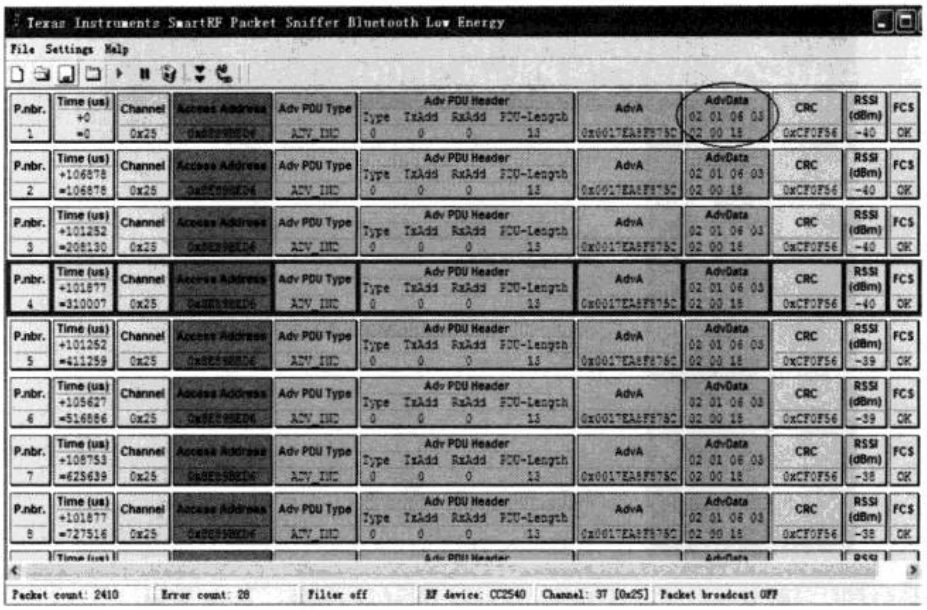


图 8-3　Texas Instruments Packet Sniffer 软件显示所捕获的数据包

8.1.2　蓝牙 4.0 BLE 数据包的结构

从 Texas Instruments Packet Sniffer 软件捕获到的数据包中可以看到，每个数据包（图 8-3 中每一行表示一个数据包）有很多段组成，这与蓝牙 4.0 BLE 协议是对应的，由于蓝牙 4.0 BLE 协议栈采用分层结构实现的，所以数据包在显示时也是不同的层使用不同的颜色来表示，这样方便读者查看不同层的数据。

读者刚刚接触到数据包时，会有很多疑问，例如，数据包是怎么构成的，如何分析这些数据包呢？请读者注意，这是蓝牙 4.0 BLE 网络内的数据包，因此，这些数据包肯定是符合蓝牙 4.0 BLE 协议的，所以需要从蓝牙 4.0 BLE 协议各层的数据构成入手，来分析数据包的构成。

下面来分析一个数据包，如图 8-4 所示。

Access Address	Adv PDU Type	Adv PDU Header				AdvA	AdvData	CRC	RSSI (dBm)	FCS
		Type	TxAdd	RxAdd	PDU-Length		02 01 05 03			
0x8E89BED6	ADV_IND	0	0	0	13	0x0017EA8F875C	02 09 18	0x551E0F	0	OK

图 8-4　数据包分析实例

从图 8-4 中可以看到“Access Address”、“Adv PDU Header”、“AdvA”、“AdvData”等不同的字段，这是从哪里来的呢？在蓝牙 4.0 协议规范[●]中对此进行

● 关于蓝牙 4.0 BLE 协议中各层数据包的格式请读者参考蓝牙 4.0 核心协议规范《Blutooth Specification Version 4.0》，该手册中详细阐述了蓝牙 4.0 BLE 协议中各层数据包的格式。

了定义。上述数据包各个字段的含义如表 8-2 所示。

表 8-2　数据包各字段含义

Access Address	ADV_IND	AdvA	AdvData	CRC	RSSI	FCS
访问地址	可连接的非定向广告	广告设备的地址	广告数据	循环冗余校验	接收的信号强度指示	帧校验序列
4 字节长度	1 字节长度	6 字节长度	0~31 字节长度	3 字节长度	1 字节长度	0 ~ 2 字节长度

8.2 蓝牙 4.0 BLE 数据传输实验剖析

上一小节讲述了如何构建一个蓝牙 4.0 BLE 协议分析仪，以及如何对无线数据包进行捕获。本小节将结合蓝牙 4.0 BLE 协议分析仪捕获到的数据包，对蓝牙节点设备与集中器设备之间连接建立过程以及数据传输过程进行深层的剖析。

8.2.1 实验原理及流程图

下面先来回顾一下第 6 章示例工程中讲述的节点设备与集中器设备之间数据传输的实验原理及操作流程。

▲ 首先集中器设备发起扫描请求，扫描正在发送广告的节点设备，若 GAP 服务的 UUID 相匹配，则两设备可以建立连接；

▲ 集中器发起建立连接的请求，外部设备响应后，两设备进入连接状态；

▲ 集中器通过特定的 UUID 进行 GATT 数据服务的发现；

▲ 发现 GATT 数据服务后，集中器发送要进行数据操作的“特性”的 UUID，节点设备将这个“特性”的句柄返回给集中器，句柄即为特性值在属性表中的地址；

▲ 集中器通过句柄进行应用数据的读取和写入操作。通过按键来触发数据的读写操作，当按键按下时，交替地进行读取特性值和写入特性值的操作，并且将每次写入的值增加 1。

集中器设备扫描发现节点设备并与之建立连接的流程如图 8-5 所示。

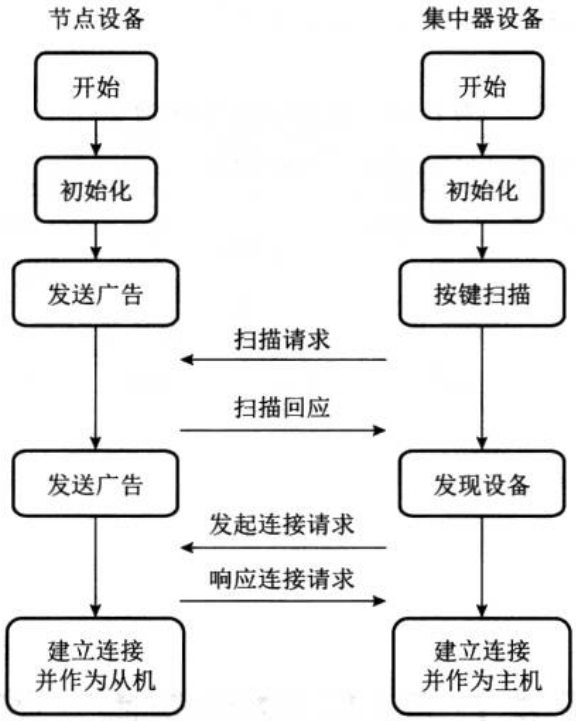


图 8-5 建立连接的流程

8.2.2 节点设备发送的广告数据

初始化完成后，节点设备处于可发现状态，以广告的形式向外界发送数据，未指定广告接收者。节点设备发送广告并处于可发现状态时，有两种模式：一种是受限的发现模式；另一种是不受限制的发现模式。其中，受限制的发现模式指的是外设在发送广告时，如果没收到集中器设备发来的建立连接请求，则只保持 30s 的可发现状态，然后转为不可被发现的待机状态。而不受限制的发现模式则是外设在没有收到集中器设备的连接请求时，一直发送广告，处于可被发现的状态。

节点设备发送的广告中包含的数据内容如下：

```
static uint8 advertData[] =
{
1  0x02,
2  GAP_ADTYPE_FLAGS,    //0x01
3  DEFAULT_DISCOVERABLE_MODE |
   GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,  //0x05
4  0x03,
5  GAP_ADTYPE_16BIT_MORE,     //0x02
6  LO_UINT16( SIMPLEPROFILE_SERV_UUID ),  //0xF0
```

```
7   HI_UINT16( SIMPLEPROFILE_SERV_UUID ),   //0xFF
};
```

第 1 行，广告中设置发现模式的数据段，占用 2 字节数据长度。

第 2 行，广告类型标志。

第 3 行，定义节点设备的可发现模式。根据预编译选项中是否包含“CC2540_MINIDK”宏定义，来决定节点设备的广告状态为受限的可发现模式或为不受限的可发现模式。

第 4 行，广告中设置 GAP 基本服务 UUID 的数据段，占用 3 字节数据长度。

第 5 行，定义 UUID 为 16bit，即 2 字节数据长度。

第 6 ～ 7 行，分别为 UUID 的低 8 位和高 8 位。只有 GAP 服务的 UUID 相匹配，两设备才能建立连接。

从协议分析仪捕获到的数据包中，可以看到“AdvData”数据段的内容为“02 01 05 03 02 F0 FF”，如图 8-6 所示，即为上述代码中第 1 ～ 7 行的广告数据。

Time (us)	Channel	Access Address	Adv PDU Type	Adv PDU Header				AdvA	AdvData	CRC	RSSI (dBm)	FCS
				Type	TxAdd	RxAdd	PDU-Length		02 01 05 03			
+105624 =2186232	0x25	0x8E89BED6	ADV_IND	0	0	0	13	0x0017EA5F8921	02 F0 FF	0x982C6D	0	OK

图 8-6　广告数据包

8.2.3　节点设备扫描回应的数据

集中器设备可以通过扫描来发现正在发送广告数据的节点设备，当节点设备接收到集中器设备的扫描请求时，回应的数据内容如下：

```
static uint8 scanRspData[] =
{
1    0x14,
2    GAP_ADTYPE_LOCAL_NAME_COMPLETE,   //0x09
3    0x53,      // 'S'
4    0x69,      // 'i'
5    0x6d,      // 'm'
6    0x70,      // 'p'
7    0x6c,      // 'l'
8    0x65,      // 'e'
9    0x42,      // 'B'
10   0x4c,      // 'L'
11   0x45,      // 'E'
12   0x50,      // 'P'
```

```
13  0x65,     // 'e'
14  0x72,     // 'r'
15  0x69,     // 'i'
16  0x70,     // 'p'
17  0x68,     // 'h'
18  0x65,     // 'e'
19  0x72,     // 'r'
20  0x61,     // 'a'
21  0x6c,     // 'l'
22  0x05,
23   GAP_ADTYPE_SLAVE_CONN_INTERVAL_RANGE,
//0x12
24   LO_UINT16( DEFAULT_DESIRED_MIN_CONN_INTERVAL ),
//0x50
25   HI_UINT16( DEFAULT_DESIRED_MIN_CONN_INTERVAL ),
//0x00
26   LO_UINT16( DEFAULT_DESIRED_MAX_CONN_INTERVAL ),
//0x20
27   HI_UINT16( DEFAULT_DESIRED_MAX_CONN_INTERVAL ),
//0x03
28  0x02,
29  GAP_ADTYPE_POWER_LEVEL,  //0x0A
30  0       // 0dBm
};
```

第1行，本地广告设备名数据段占用20个字节数据长度。

第2行，指明接下来的数据为本地广告设备的名称。

第3～21行，本地广告设备的完整名称为“SimpleBLEPeripherial”。

第22行，连接间隔数据段占用5字节数据长度。

第23行，指明接下来的数据为连接间隔的最小值和最大值。

第24、25行，分别为连接间隔最小值的低8位和高8位，共16bit，即2字节数据长度。

第26、27行，分别为连接间隔最大值的低8位和高8位，共16bit，即2字节数据长度。

第28行，发射功率数据段为2字节数据长度。

第29行，指明接下来的数据为发射功率，发射功率的可调范围为-127～127dBm。

第30行，将发射功率设置为0dBm。

从协议分析仪捕获到的数据包中，可以看到“ScanRspData”数据段的内容为“14 09 53 69 6D 70 6C 65 42 4C 45 50 65 72 69 70 68 65 72 61 6C 05 12 50 00 20 03 02 0A 00”，如图8-7所示，即为上述代码中第1～30行扫描回应的数据。

Adv PDU Type	Adv PDU Header				AdvA	ScanRspData
	Type	TxAdd	RxAdd	PDU-Length		14 09 53 69 6D 70 6C 65 42 4C 45 50 65 72 69
ADV_SCAN_RSP	4	0	0	36	0x0017EA8F8921	70 68 65 72 61 6C 05 12 50 00 20 03 02 0A 00

图 8-7　扫描回应的数据包

集中器设备接收到节点设备扫描回应的数据后，向节点设备发送连接请求，节点设备接受请求并作为从机进入连接状态。连接请求的数据包如图 8-8 所示。

Adv PDU Type	Adv PDU Header				InitA	AdvA	LLData (Part 1)				
	Type	TxAdd	RxAdd	PDU-Length			AccessAddr	CRCInit	WinSize	WinOffset	Interval
ADV_CONNECT_REQ	5	0	0	34	0x0017EA8F875C	0x78C5E59B82A7	0x8BE777B4	0E 46 B0	02	0x004E	0x0050

图 8-8　连接请求的数据包

图 8-8 中，“ADV_CONNECT_REQ”数据段表示此数据包内容为连接请求；“InitA”数据段为发起连接请求的设备，此时为集中器设备；“AdvA”数据段为接收连接请求的设备，此时为节点设备；后面的数据段表示连接间隔、从机延时、管理超时等信息。

8.2.4　蓝牙 4.0 BLE 应用数据传输流程分析

集中器设备与节点设备建立连接后，进行服务发现、特性发现以及特性中应用数据读写的流程如图 8-9 所示。

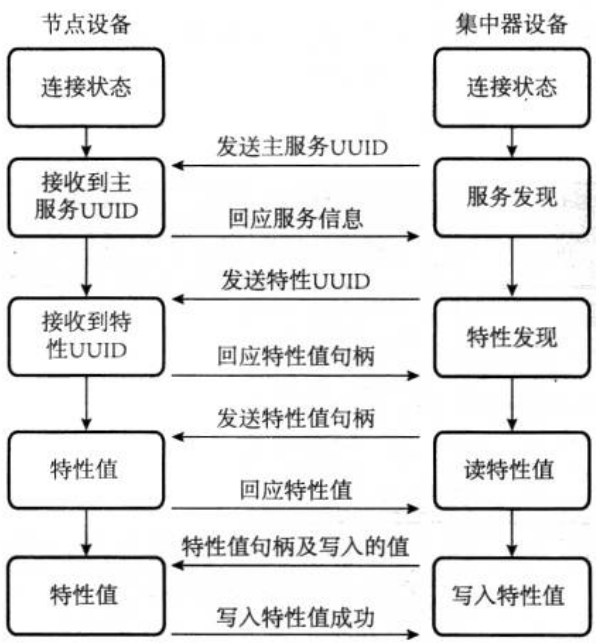


图 8-9　应用数据传输流程

当集中器设备需要读取节点设备中提供的应用数据时，需要首先进行GATT数据服务发现。集中器进行GATT数据服务发现时，需要给出想要发现的主服务的UUID，只有主服务UUID匹配，才能获得GATT数据服务。

在第6章讲述的集中器示例工程中，服务发现处理函数内容如下：

```
static void simpleBLECentralStartDiscovery( void )
{
1  uint8  uuid[ATT_BT_UUID_SIZE]  =  {  LO_UINT16(SIMPLEPROFILE_SERV_UUID),HI_UINT16(SIMPLEPROFILE_SERV_UUID) };
2  simpleBLESvcStartHdl = simpleBLESvcEndHdl = simpleBLECharHdl = 0;
3  simpleBLEDiscState = BLE_DISC_STATE_SVC;
4  GATT_DiscPrimaryServiceByUUID( simpleBLEConnHandle, uuid,ATT_BT_UUID_SIZE, simpleBLETaskId );
}
```

第1行，指定想要发现的主服务的UUID，共16bit，分为高8位和低8位。在“simpleGATTpeofile.h”文件中可以找到主服务UUID的定义。

第2行，将服务的起始句柄、结束句柄以及“特性”句柄清零，句柄为特性在属性表中的地址。

第3行，将当前发现状态标志设为服务发现。

第4行，通过指定的主服务UUID来发现GATT数据服务，只发现UUID匹配的服务。这一函数处理上一次GATT过程的结果，并初始化下一步的发现操作。

对比协议分析仪捕获到的数据包，可以找到“ATT_Find_By_Type_Value_Req”数据段，即为集中器设备发送的GATT数据服务发现请求，要发现的主服务的UUID为“FFF0”。如果想要读属性表中所有的属性，需要将起始句柄设为“0x0001”，结束句柄设为“0xFFFF”，如图8-10所示。

Data Type	Data Header					L2CAP Header		ATT_Find_By_Type_Value_Req				
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	AttValue
L2CAP-S	2	0	0	0	13	0x0009	0x0004	0x06	0x0001	0xFFFF	0x2800	F0 FF

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
L2CAP-C	1	1	0	0	0	0xFAD7C9	0	OK

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
L2CAP-C	1	1	1	0	0	0xFADA6F	0	OK

Data Type	Data Header					L2CAP Header		ATT_Find_By_Type_Value_Rsp	CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode HandleInfo			
L2CAP-S	2	0	1	0	9	0x0005	0x0004	0x07 23 00 39 00	0x92B5FE	0	OK

图8-10 GATT数据服务发现请求及回应

图8-10中，“ATT_Find_By_Type_Value_Rsp”是对数据服务发现请求的回应，包含了服务器的信息。回应的数据中应该包含一个或多个完整的句柄信息，不能被数据包分割，句柄信息列表按照发现的结果顺序排列。

集中器设备发现GATT主服务后，需要进一步寻找要操作的“特性”的句柄，需要通过这个的“特性”的UUID来进行句柄的获取。这一操作在集中器示例工程中，是通过调用GATT特性发现函数进行处理的，simpleBLEGATTDiscoveryEvent函数内容如下：

```
static void simpleBLEGATTDiscoveryEvent( gattMsgEvent_t
*pMsg )
{
  attReadByTypeReq_t req;
  if ( simpleBLEDiscState == BLE_DISC_STATE_SVC )
  {
    if(pMsg->method==ATT_FIND_BY_TYPE_VALUE_RSP&&
    pMsg->msg.findByTypeValueRsp.numInfo > 0 )
  {
1   simpleBLESvcStartHdl = pMsg->msg.findByTypeValueRsp.
handlesInfo[0].handle;
2 simpleBLESvcEndHdl = pMsg->msg.findByTypeValueRsp.
handlesInfo[0].grpEndHandle;
}
3 if((pMsg->method==ATT_FIND_BY_TYPE_VALUE_RSP&&pMsg-
>hdr.status= =
bleProcedureComplete ) || ( pMsg->method == ATT_ERROR_
RSP ) )
{
4 if ( simpleBLESvcStartHdl != 0 )
{
5   simpleBLEDiscState = BLE_DISC_STATE_CHAR;
6   req.startHandle = simpleBLESvcStartHdl;
7   req.endHandle = simpleBLESvcEndHdl;
8   req.type.len = ATT_BT_UUID_SIZE;
9    req.type.uuid[0] = LO_UINT16(SIMPLEPROFILE_CHAR1_
UUID);
10   req.type.uuid[1] = HI_UINT16(SIMPLEPROFILE_CHAR1_
UUID);
11   GATT_ReadUsingCharUUID( simpleBLEConnHandle, &req,
```

```
simpleBLETaskId );
        }
      }
    }
  ......
  }
```

第1、2行，获取返回的消息中的GATT服务的起始句柄和结束句柄，一个服务当中可能包含多个“特性”。

第3～8行，若从上一步骤获得服务的起始句柄不为0，将这些参数赋值给req结构体，用来进行下一步的GATT特性发现操作。

第9、10行，要操作的“特性”的UUID。

第11行，通过发送某个“特性”的UUID，来获得这个特性值的句柄。

当节点设备收到集中器设备发送的“特性”发现请求时，将特性值的句柄返回，集中器设备收到返回的特性值的句柄信息后，再次调用simpleBLEGATTDiscoveryEvent函数来处理返回的GATT消息：

```
static void simpleBLEGATTDiscoveryEvent( gattMsgEvent_t
*pMsg )
{
1  attReadByTypeReq_t req;
  ......
2  else if ( simpleBLEDiscState == BLE_DISC_STATE_CHAR )
   {
3    if(pMsg->method==ATT_READ_BY_TYPE_RSP&&
      pMsg->msg.readByTypeRsp.numPairs > 0 )
      {
4     simpleBLECharHdl= BUILD_UINT16( pMsg->msg.
readByTypeRsp.dataList[0],pMsg->msg.readByTypeRsp.
dataList[1] );
5      LCD_WRITE_STRING( "Simple Svc Found", HAL_LCD_
LINE_1 );
6     simpleBLEProcedureInProgress = FALSE;
      }
7     simpleBLEDiscState = BLE_DISC_STATE_IDLE;
   }
}
```

第1行，定义一个结构体，用于处理消息中包含的数据。

第2～4行，从返回的数据中得到第一个特性值的句柄，因为前面已经发送了

这个特性值的 UUID。

第 5 行，液晶显示字符串“Simple Svc Found”，表明已经找到要操作的“特性”的句柄，后面可以进行“特性”中数据的读写操作了。

第 6 ～ 7 行，将处理进程标志设为完成，发现状态标志设为空闲，这两个标志只是在程序的编写中起判断作用。

对比协议分析仪捕获到的数据包，可以找到“ATT_Read_By_Type_Req”数据段，即为集中器设备发送的“特性”发现请求，发送的“特性”的 UUID 为“0xFFF1”，从属性表中开始查找的起始句柄为“0x0023”，结束句柄为“0xFFFF”，如图 8-11 所示。

Data Type	Data Header					L2CAP Header		ATT_Read_By_Type_Req				CRC
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	
L2CAP-S	2	0	0	0	11	0x0007	0x0004	0x08	0x0023	0xFFFF	F1 FF	0x301BE3

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
L2CAP-C	1	1	0	0	0	0xB757C7	-31	OK

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
L2CAP-C	1	1	1	0	0	0xB75A61	-30	OK

Data Type	Data Header					L2CAP Header		ATT_Read_By_Type_Rsp			CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Length	AttData			
L2CAP-S	2	0	1	0	9	0x0005	0x0004	0x09	0x03	25 00 01	0x797568	-30	OK

Data Type	Data Header					L2CAP Header		ATT_Read_By_Type_Req				CRC
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	
L2CAP-S	2	0	0	0	11	0x0007	0x0004	0x08	0x0026	0xFFFF	F1 FF	0x96B1F4

图 8-11　通过 UUID 进行“特性”发现

节点设备接收到“特性”的 UUID 后，在属性表中找到特性值的句柄，并将句柄值返回给集中器设备。从图 8-11 中可以看到，“ATT_Read_By_Type_Rsp”数据段中包含了返回的数据长度、句柄值以及句柄长度。其中，UUID 为“0xFFF1”的特性值的句柄为“0x0025”。

获得特性值的句柄以后，集中器设备就可以通过这个句柄进行应用数据的读写操作了。在集中器示例工程中，通过按键处理函数来发送要读取的特性的句柄，代码如下所示：

```
static void simpleBLECentral_HandleKeys( uint8 shift,
uint8 keys )
{
(void)shift;
if ( keys & HAL_KEY_SW_1 )
{
```

```
      ......
1   else if ( simpleBLEState == BLE_STATE_CONNECTED
&&simpleBLECharHdl != 0&&simpleBLEProcedureInProgress ==
FALSE )
      {
      ......
2   attReadReq_t req;
3   req.handle = simpleBLECharHdl;
4   status=GATT_ReadCharValue(simpleBLEConnHandle,&req,
simpleBLETaskId );
      }
    }
  }
```

第1行，已建立连接，要操作的“特性”的句柄不为0，且上一阶段处理进程已完成。

第2行，定义一个结构体变量req，将要读取的特性值的句柄赋值给结构体变量。

第3行，调用协议栈提供的读取函数，从指定的句柄处读取特性值数据，并返回状态标志来判断是否正在进行读取数据的操作。

节点设备收到读特性值请求以及句柄后，将特性值数据返回给集中器设备，当集中器设备的天线接收到返回的数据时，调用事件处理函数流程为SimpleBLECentral_ProcessEvent--> simpleBLECentral_ProcessOSALMsg-->simpleBLECentralProcessGATTMsg。

GATT消息处理函数simpleBLECentralProcessGATTMsg内容如下：

```
static void simpleBLECentralProcessGATTMsg( gattMsgEvent_
t *pMsg )
{
1  if ( simpleBLEState != BLE_STATE_CONNECTED )
   {
2    return;
   }
3  if ( ( pMsg->method == ATT_READ_RSP ) ||( ( pMsg->method
== ATT_ERROR_RSP )&&( pMsg->msg.errorRsp.reqOpcode == ATT_
READ_REQ ) ) )
   {
4   if ( pMsg->method == ATT_ERROR_RSP )
     {
5     uint8 status = pMsg->msg.errorRsp.errCode;
```

```
6        LCD_WRITE_STRING_VALUE( "Read Error", status, 10,
HAL_LCD_LINE_1 );
      }
      else
      {
7     uint8 valueRead = pMsg->msg.readRsp.value[0];
8     LCD_WRITE_STRING_VALUE("Read rsp:", valueRead, 10,
HAL_LCD_LINE_1 );
      }
9    simpleBLEProcedureInProgress = FALSE;
    }
    ......
}
```

第 1、2 行，如果 GATT 消息到来时，设备已断开连接，则忽略这个消息。

第 3 ～ 6 行，若返回读操作失败，获取操作的错误码，并在 LCD 上显示。错误码代表了出错的原因。

第 7、8 行，若返回读操作成功，在 LCD 上显示读取的特性值，即为应用数据。

第 9 行，将处理过程标志赋值为 FALSE，表示读特性操作已经完成。

下面来分析读特性值操作的数据包，“ATT_Read_Req”数据段为读特性值请求，包含了要读取的特性值的句柄“0x0025”，如图 8-12 所示。

Access Address	Data Type	Data Header					L2CAP Header		ATT_Read_Req		CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle			
0xD5CC10C0	L2CAP-S	2	1	1	0	7	0x0003	0x0004	0x0A	0x0025	0xA2BE58	-30	OK

Access Address	Data Type	Data Header					CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length			
0xD5CC10C0	L2CAP-C	1	0	1	0	0	0xB75CB2	-30	OK

Access Address	Data Type	Data Header					CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length			
0xD5CC10C0	L2CAP-C	1	0	0	0	0	0xB75114	-30	OK

Access Address	Data Type	Data Header					L2CAP Header		ATT_Read_Rsp		CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttValue			
0xD5CC10C0	L2CAP-S	2	1	0	0	6	0x0002	0x0004	0x0B	01	0xD601FC	-30	OK

图 8-12　读特性值操作

图 8-12 中，“ATT_Read_Rsp”数据段即为节点设备接收到读特性值请求后，返回的特性值，“0x0025”句柄处的特性值为“01”，即为我们要寻找的应用数据。需要注意，特性值的数据长度及读写权限是在属性表中定义的，只有具有读权限的

特性值才能被正确读取，否则数据包中会返回读错误的操作码。

下面来分析向特性中写入应用数据的流程。集中器示例工程中，将simpleBLECharVal的值（初始值为0）写入到第一个“特性”中。处理流程为：按键按下时，HAL层发送一个OSAL消息给应用，产生SYS_EVENT_MSG事件，通过调用SimpleBLECentral_ProcessEvent --> simpleBLECentral_ProcessOSALMsg函数来处理这个消息。若检测到为按键事件，则进一步调用按键处理函数simpleBLECentral_HandleKeys来处理按键任务，进行写特性值操作。执行写特性值操作的代码如下所示（省略部分无关代码）：

```
static void simpleBLECentral_HandleKeys( uint8 shift,
uint8 keys )
{
  (void)shift;
  if ( keys & HAL_KEY_SW_1 )
  {
  ......
1   else if ( simpleBLEState == BLE_STATE_CONNECTED
&&simpleBLECharHdl != 0&&simpleBLEProcedureInProgress ==
FALSE )
    {
      uint8 status;
2     if ( simpleBLEDoWrite )
      {
3       attWriteReq_t req;
4       req.handle = simpleBLECharHdl;
5       req.len = 1;
6       req.value[0] = simpleBLECharVal;
7       req.sig = 0;
8       req.cmd = 0;
9 status=GATT_WriteCharValue(simpleBLEConnHandle, &req,
simpleBLETaskId );
      }
  ......
  }
}
```

第1行，已建立连接，要操作的“特性”的句柄不为0，且上一阶段处理进程已完成。

第2～8行，定义一个结构体变量req，将句柄、要写入的应用数据等参数赋

值给结构体变量。

第 9 行，调用协议栈提供的写特性值函数，向指定的句柄中写入数据，并返回状态标志来判断是否正在进行写入数据的操作。

下面来分析写特性值操作的数据包，“ATT_write_Req”数据段为写特性值请求，包含了要写入的特性值的句柄“0x0025”，写入的应用数据为“0”，在“ATT_Write_Rsp”数据段中返回的操作码为“0x13”，表示写入数据成功，如图 8-13 所示。

Access Address	Data Type	Data Header					L2CAP Header		ATT_Write_Req			CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue			
0xD5CC10C0	L2CAP-S	2	0	0	0	8	0x0004	0x0004	0x12	0x0025	00	0xC21FF8	-30	OK

Access Address	Data Type	Data Header					CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length			
0xD5CC10C0	L2CAP-C	1	1	0	0	0	0xB757C7	-30	OK

Access Address	Data Type	Data Header					CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length			
0xD5CC10C0	L2CAP-C	1	1	1	0	0	0xB75A61	-31	OK

Access Address	Data Type	Data Header					L2CAP Header		ATT_Write_Rsp	CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode			
0xD5CC10C0	L2CAP-S	2	0	1	0	5	0x0001	0x0004	0x13	0x007356	-30	OK

图 8-13　写特性值操作

8.2.5　数据收发实验回顾

结合协议分析仪捕获的数据包可以方便的对程序中的数据流进行跟踪，读者在使用蓝牙 4.0 BLE 协议栈进行应用程序的开发时，只需要调用协议栈中提供的函数即可实现数据的无线发送和接收，集中器设备与节点设备建立连接阶段发送和接收的数据如图 8-14 所示。

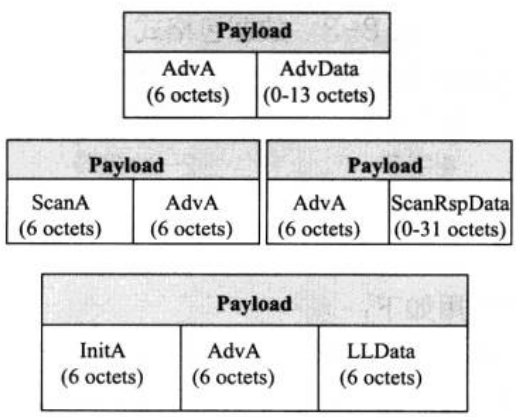


图 8-14　连接阶段的数据

Payload 可以理解为这一层数据包中所包含的有效数据，称为有效载荷。为什么称为有效载荷呢？这好比读者需要邮寄一封信，写完信后，要用信封将其包装起来，同时写好地址并贴好邮票，邮局需要盖上邮戳，然后才能寄出去。从读者的角度看，信封、邮票和邮戳对于信的内容来说是“附加的一些数据”，但这却是为了确保这封信能顺利到达接收方，而信的实际内容便是有效载荷。

同样的道理，在蓝牙 4.0 BLE 协议栈中，不同的层都会有相应的数据头、有效载荷和一些校验信息，这主要是为了确保通信过程中数据的完整性，关于蓝牙 4.0 BLE 数据包的格式请参阅本章扩展阅读部分。

8.3 本章小结

本章首先讲述了如何构建一个蓝牙 4.0 BLE 协议分析仪，以及如何进行无线数据包的捕获。然后，结合捕获到的数据包来对设备之间建立连接的过程以及数据传输的基本流程进行了深层的剖析。结合实验现象来进行分析，通过实践来进行验证，一直是笔者推荐的学习思路。

[扩展阅读] 蓝牙 4.0 BLE 协议栈数据包格式

在蓝牙 4.0 BLE 无线网络中，需要将命令或者数据按照特定的格式组成数据包（Packet），以便在不同的设备之间进行无线传输。蓝牙 4.0 BLE 数据包的格式如表 8-3 所示。

表 8-3 数据包格式

Preamble	Access Address	PDU	CRC
前同步码	存取地址	协议数据单元	循环冗余校验
1 字节 长度	4 字节 长度	2~39 字节 长度	3 字节 长度

数据包各部分组成及作用如下：

▲ 前同步码（Preamble）——所有链路层数据包都有 8 位的前同步码，前同步码用于接收器执行频率同步、符号定时估计和自动增益控制；

▲ 存取地址（Access Address）——广告通道数据包的存取地址固定为“0x8E89BED6”，而数据通道数据包的存取地址对于任意两个设备的链路层连接都

是不同的；

▲ 协议数据单元（PDU）——数据包在广告物理层通道和数据物理层通道传输时，使用不同结构的协议数据单元；

▲ 循环冗余校验（CRC）——每一个链路层数据包的结尾都有 24 位的循环冗余校验，在协议数据单元之后计算。

广告通道的协议数据单元由 16 位的数据头和可变大小的有效载荷构成，如表 8-4 所示。

表 8-4 广告通道的协议数据单元

Header（数据头）	Payload（有效载荷）
16 位	由数据头长度字段决定

广告通道的协议数据单元中，数据头字段如表 8-5 所示。

表 8-5 数据头字段

PDU 类型	射频单元	TxAdd	RxAdd	长度	射频单元
4 位	2 位	1 位	1 位	6 位	2 位

TxAdd 和 RxAdd 字段中包含 PDU 类型的信息。数据头中的长度字段，指明了协议数据单元中有效载荷字段的字节长度。

下面来分析一个广告通道的数据包，“Adv PDU Header”即为协议数据单元的数据头，而“AdvA”和“AdvData”即为有效载荷，如图 8-15 所示。

Access Address	Adv PDU Type	Adv PDU Header				AdvA	AdvData	CRC	RSSI (dBm)	FCS
		Type	TxAdd	RxAdd	PDU-Length		02 01 05 03			
0x8E89BED6	ADV_IND	0	0	0	13	0x0017EA8F875C	02 09 18	0x551E0F	0	OK

图 8-15 广告通道的数据包

数据通道的协议数据单元由 16 位的数据头和可变大小的有效载荷构成，有可能包含消息完整性校验（MIC）字段，如表 8-6 所示。

表 8-6 数据通道的协议数据单元

Header（数据头）	Payload（有效载荷）	消息完整性校验（MIC）字段
16 位	格式由数据头中的 LLID 字段决定	32 位

数据通道的协议数据单元中，数据头字段如表 8-7 所示。

表 8-7　数据头字段

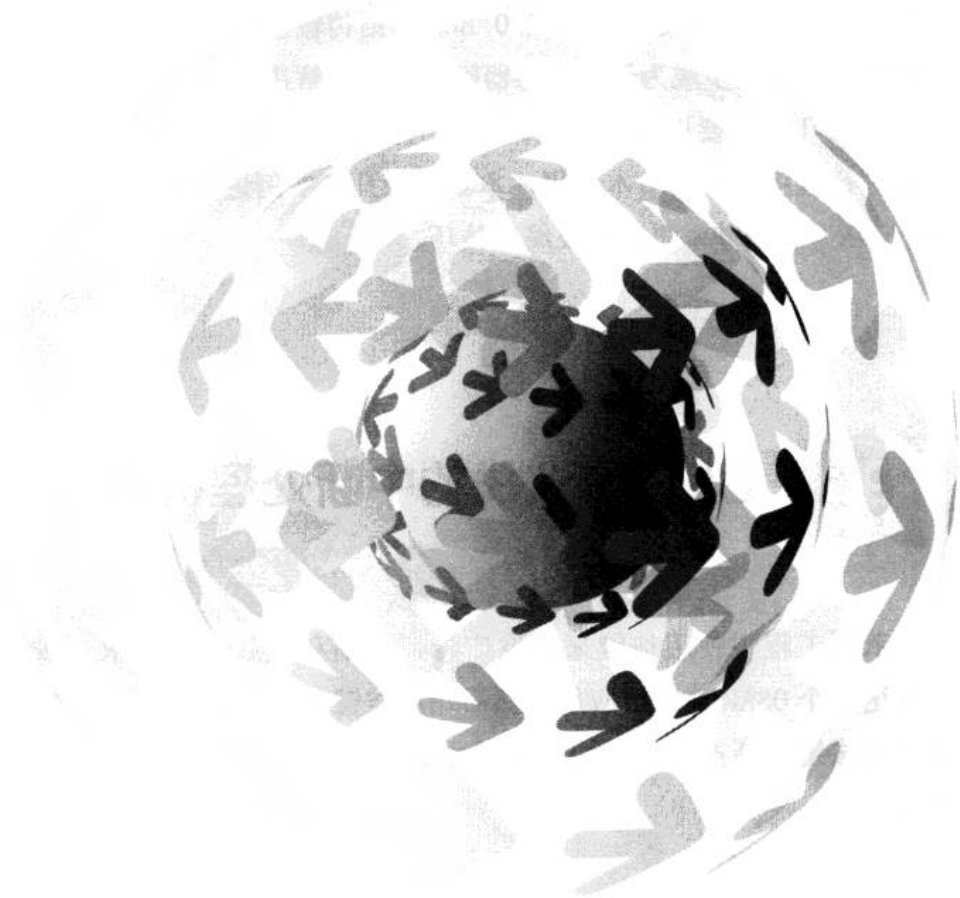
LLID	NESN	SN	MD	射频单元	PDU 长度	射频单元
2 位	1 位	1 位	1 位	2 位	6 位	2 位

有效载荷的格式是由数据头中的 LLID 字段决定的。数据头中的长度字段指明了协议数据单元中有效载荷字段的字节长度和消息完整性校验字段的字节长度。

下面来分析一个数据通道的数据包，“Data Header”即为协议数据单元的数据头字段，可以看到其中包含 LLID、NESN、SN、MD、PDU-Length 五部分，如图 8-16 所示。

P.nbr.	Time (us)	Channel	Access Address	Data Type	Data Header					CRC	RSSI (dBm)	FCS
	+99999				LLID	NESN	SN	MD	PDU-Length			
110	=10512527	0x08	0xD5CC10C0	L2CAP-C	1	1	1	0	0	0xB75A61	0	OK

图 8-16　数据通道的数据包



第 9 章

蓝牙 4.0 BLE 无线网络技术拓展

随着蓝牙4.0 BLE技术的逐步推广和应用，目前市面上已经出现大量的蓝牙4.0 BLE产品，包括开发板、蓝牙4.0 BLE数据传输模块等，蓝牙4.0 BLE技术依靠低功耗的优势，逐步成为无线传感器网络的一种解决方案。

前面几章主要以实验为主，理论与实践相结合，讲解了利用蓝牙4.0 BLE协议栈进行应用开发的基本流程，本章结合几个实例进行讲解，向读者展示蓝牙4.0 BLE应用开发的基本思路以及编程技巧。

9.1 BLE协议栈的中断处理机制

示例工程中使用按键来触发广告、扫描和发起连接请求，按键按下后，HAL层发送一个OSAL消息给应用，然后调用按键处理函数进行按键事件的处理。本节将讲述按键中断产生后，协议栈的中断处理机制以及中断处理函数中需要处理的内容。

9.1.1 中断处理编程

首先，读者需要了解在IAR EW集成开发环境下，声明中断处理函数的方法：

```
#pragma vector=P1INT_VECTOR
__interrupt void P1_ISR (void)
{
……   // 在此添加自己的中断处理内容
}
```

其中，加粗部分是固定的格式，I/O口中断处理函数的名字可以任意取，如P1_ISR或者p1_isr。下面以“SimpleBLEPeripheral示例工程”为例，进行协议栈中断处理流程的讲解。工程文件布局如图9-1所示。

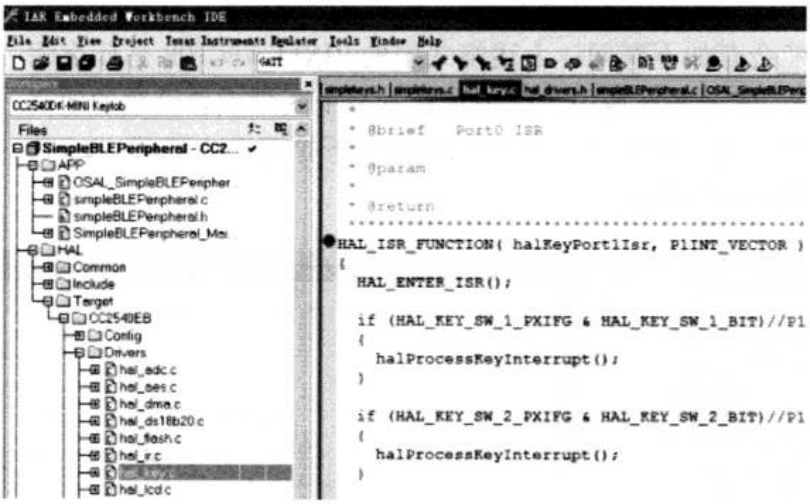


图 9-1 工程文件布局

在“hal_key.c”文件中找到 HAL_ISR_FUNCTION 中断处理函数，如下所示：

```
HAL_ISR_FUNCTION( halKeyPort1Isr, P1INT_VECTOR )
{
1  HAL_ENTER_ISR();
2  if (HAL_KEY_SW_1_PXIFG & HAL_KEY_SW_1_BIT)
  {
3  halProcessKeyInterrupt();
  }
4  if (HAL_KEY_SW_2_PXIFG & HAL_KEY_SW_2_BIT)
  {
5  halProcessKeyInterrupt();
  }
6  HAL_KEY_SW_1_PXIFG = 0;
7  HAL_KEY_SW_2_PXIFG = 0;
8  HAL_KEY_CPU_PORT_1_IF = 0;
9  CLEAR_SLEEP_MODE();
10  HAL_EXIT_ISR();
}
```

第 1 行，进入中断服务程序。

第 2、3 行，判断 Key1 按键按下时，执行 Key1 按键对应的 HAL 层的按键处理。

第 4、5 行，判断 Key2 按键按下时，执行 Key2 按键对应的 HAL 层的按键处理。

第 6 ～ 8 行，清除 CPU 中断标志位，首先将 PxIFG 标志位清零，然后再将 PxIF 标志位清零。

第9、10行，退出中断服务程序。

读者可能会有如下疑问：上述格式中的PINT1_VECTOR是什么呢，从哪里定义的呢？在IAR EW开发环境中，如果想查看某个变量或者函数的定义，只需要在该变量或者函数上面右键单击，然后在弹出的下拉菜单中选择“go to definition of”即可，如图9-2所示。

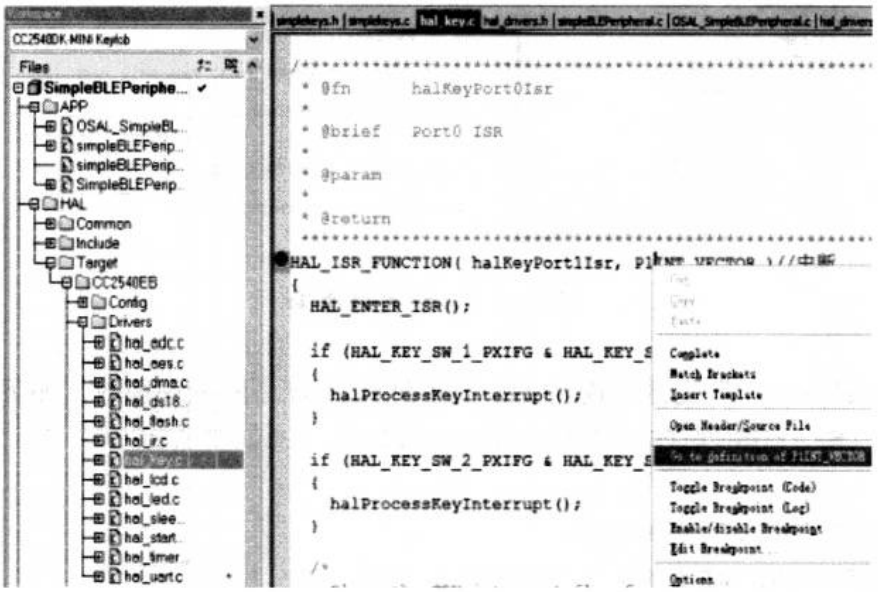


图9-2 查看变量或者函数的定义

此时，会跳转到中断向量表中P1INT_VECTOR的定义处，如图9-3所示。

```
#define  RFERR_VECTOR    VECT(  0, 0x03 )   /*
#define  ADC_VECTOR      VECT(  1, 0x0B )   /*
#define  URX0_VECTOR     VECT(  2, 0x13 )   /*
#define  URX1_VECTOR     VECT(  3, 0x1B )   /*
#define  ENC_VECTOR      VECT(  4, 0x23 )   /*
#define  ST_VECTOR       VECT(  5, 0x2B )   /*
#define  P2INT_VECTOR    VECT(  6, 0x33 )   /*
#define  UTX0_VECTOR     VECT(  7, 0x3B )   /*
#define  DMA_VECTOR      VECT(  8, 0x43 )   /*
#define  T1_VECTOR       VECT(  9, 0x4B )   /*
#define  T2_VECTOR       VECT( 10, 0x53 )   /*
#define  T3_VECTOR       VECT( 11, 0x5B )   /*
#define  T4_VECTOR       VECT( 12, 0x63 )   /*
#define  P0INT_VECTOR    VECT( 13, 0x6B )   /*
#define  UTX1_VECTOR     VECT( 14, 0x73 )   /*
#define  P1INT_VECTOR    VECT( 15, 0x7B )   /*
#define  RF_VECTOR       VECT( 16, 0x83 )   /*
#define  WDT_VECTOR      VECT( 17, 0x8B )   /*
```

图9-3 中断向量表

下面再来看一下 HAL_ISR_FUNCTION 函数的定义，在 HAL_ISR_FUNCTION 函数上面右键单击，然后在弹出的下拉菜单中选择“go to definition of”，找到该函数的宏定义处，如图 9-4 所示。

```
/* ---------------------- IAR Compiler ---------------------- */
#ifdef __IAR_SYSTEMS_ICC__

#ifdef CC2541
#include <ioCC2541.h>
#else // CC2540
#include <ioCC2540.h>
#endif // CC2541

#define HAL_COMPILER_IAR
#define HAL_MCU_LITTLE_ENDIAN()   __LITTLE_ENDIAN__
#define _PRAGMA(x) _Pragma(#x)
#define HAL_ISR_FUNC_DECLARATION(f,v)   _PRAGMA(vector=v) __near_func __interrupt void f(void)
#define HAL_ISR_FUNC_PROTOTYPE(f,v)     _PRAGMA(vector=v) __near_func __interrupt void f(void)
#define HAL_ISR_FUNCTION(f,v)           HAL_ISR_FUNC_PROTOTYPE(f,v); HAL_ISR_FUNC_DECLARATION(f,v)
```

图 9-4 中断处理函数的宏定义

图 9-4 中的宏定义可以简化为如下内容：

```
_pragma (#vector=P1INT_VECTOR)
__near_func __interrupt void f (void)
{
……   // 在此添加自己的中断处理内容
}
```

当有按键按下时，首先调用 halProcessKeyInterrupt() 函数进行处理，代码如下：

```
void halProcessKeyInterrupt (void)
{
  bool valid=FALSE;
  if (HAL_KEY_SW_1_PXIFG & HAL_KEY_SW_1_BIT)
  {
    HAL_KEY_SW_1_PXIFG = ~(HAL_KEY_SW_1_BIT);// 清除中断标志
    valid = TRUE;
  }
  if (HAL_KEY_SW_2_PXIFG & HAL_KEY_SW_2_BIT)
```

```
    {
      HAL_KEY_SW_2_PXIFG = ~(HAL_KEY_SW_2_BIT);
      valid = TRUE;
    }
    if (valid)
    {
      HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);    //点亮LED灯
      osal_start_timerEx(Hal_TaskID,HAL_KEY_EVENT,
      HAL_KEY_DEBOUNCE_VALUE);  //有按键按下时，产生定时任务事件
    }
  }
```

当osal_start_timerEx()函数设置的定时时间到达后，触发Hal_ProcessEvent()处理函数，函数代码如下：

```
  uint16 Hal_ProcessEvent( uint8 task_id, uint16 events )
  {
    uint8 *msgPtr;
    (void)task_id;
    if ( events & SYS_EVENT_MSG )
    {
      msgPtr = osal_msg_receive(Hal_TaskID);
      while (msgPtr)
      {
        osal_msg_deallocate( msgPtr );
        msgPtr = osal_msg_receive( Hal_TaskID );
      }
    return events ^ SYS_EVENT_MSG;
    }
  if (events & HAL_KEY_EVENT)
  {
  #if (defined HAL_KEY) && (HAL_KEY == TRUE)
  HalKeyPoll();  //检测哪个按键按下
  if (!Hal_KeyIntEnable)
    {
      osal_start_timerEx( Hal_TaskID, HAL_KEY_EVENT, 100);
    }
    #endif
      return events ^ HAL_KEY_EVENT;
```

```
    }
    return 0;
  }
```

接下来HAL层发送一个OSAL消息给应用，通过应用层的事件处理函数来处理OSAL消息，SimpleBLEPeripheral_ProcessEvent()函数内容如下所示：

```
  uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
uint16 events )
  {
    VOID task_id;
    if ( events & SYS_EVENT_MSG )
    {
      uint8 *pMsg;
      if ( (pMsg = osal_msg_receive( simpleBLEPeripheral_
TaskID )) != NULL )
      {
        simpleBLEPeripheral_ProcessOSALMsg( (osal_event_hdr_
t *)pMsg );
        VOID osal_msg_deallocate( pMsg );
      }
    return (events ^ SYS_EVENT_MSG);
    }
    ......
    return 0;
  }
```

调用simpleBLEPeripheral_ProcessOSALMsg()函数来处理OSAL消息中的按键事件，函数代码如下所示：

```
  static void simpleBLEPeripheral_ProcessOSALMsg( osal_
event_hdr_t *pMsg )
  {
    switch ( pMsg->event )
    {
    #if defined( CC2540_MINIDK )
    case KEY_CHANGE:
      simpleBLEPeripheral_HandleKeys(((keyChange_t*)pMsg)-
>state,((keyChange_t*)pMsg)->keys );
      break;
    #endif
```

```
      default:
        break;
      }
    }
```

最后，在按键处理函数中执行按键事件的处理，按键处理函数内容如下：

```
    static void simpleBLEPeripheral_HandleKeys( uint8 shift,
uint8 keys )
    {
      uint8 SK_Keys = 0;
      uint8 buf[12];  //开辟一个缓冲区
      VOID shift;
      if ( keys & HAL_KEY_SW_1 )
      {
        SK_Keys |= SK_KEY_LEFT;
        //Key1 按键按下时，通过串口输出字符串 "key1_pressed"
        osal_memcpy(buf,"key1_pressed",12);
        HalUARTWrite(0, buf,12) ;
        osal_memset(buf,0,12);
      }
      if ( keys & HAL_KEY_SW_2 )
      {
        SK_Keys |= SK_KEY_RIGHT;
        //Key2 按键按下时，通过串口输出字符串 "key2_pressed"
        osal_memcpy(buf,"key2_pressed",12);
        HalUARTWrite(0, buf,12) ;
        osal_memset(buf,0,12);
        if( gapProfileState != GAPROLE_CONNECTED )
        {
        uint8 current_adv_enabled_status;
        uint8 new_adv_enabled_status;
        GAPRole_GetParameter(GAPROLE_ADVERT_ENABLED,
        &current_adv_enabled_status );
        if( current_adv_enabled_status == FALSE )
          {
            new_adv_enabled_status = TRUE;
          }
          else
```

```
        {
      new_adv_enabled_status = FALSE;
        }
   GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof
( uint8 ), &new_adv_enabled_status );
     }
     }
     SK_SetParameter( SK_KEY_ATTR, sizeof ( uint8 ), &SK_
Keys );
   }
```

修改“SimpleBLEPeripheral.c”文件内容如下（新增加的部分以加粗字体显示）：

```
#include "hal_uart.h"
```

向 SimpleBLEPeripheral_Init( ) 函数中添加：

```
1  halUARTCfg_t uartConfig;
2  uartConfig.configured      = TRUE;
3  uartConfig.baudRate      = HAL_UART_BR_115200;
4  uartConfig.flowControl   = FALSE;
5  HalUARTOpen (0, &uartConfig);
```

第 1 ～ 4 行，配置串口参数，包括串口波特率、是否使用流控等。

第 5 行，初始化串口。

蓝牙 4.0 BLE 协议栈中对串口的配置是使用一个结构体来实现的，该结构体为 halUARTCfg_t，在此读者不必关心该结构体的具体定义形式，只需要对其功能有个了解，该结构体将串口初始化有关的参数集合在了一起，例如：波特率、是否打开串口、是否使用流控等，读者只需要将各个参数初始化就可以。

最后使用 HalUARTOpen() 函数对串口进行初始化，注意，该函数将 halUARTCfg_t 类型的结构体变量作为参数，因为 halUARTCfg_t 类型的结构体变量已经包含了串口初始化相关的参数，所以，将这些参数传递给 HalUARTOpen() 函数，HalUARTOpen() 函数使用这些参数对串口进行了初始化。

9.1.2 实例测试

将代码编译并下载到开发板中，将开发板与 PC 通过串口线连接，打开 PC 端的串口调试助手，将波特率设置为 115200，此时，开发板上的 Key1 键按下后，点亮 LED 灯，并通过串口向 PC 端输出了字符串“key1_pressed”，如图 9-5 所示。

图 9-5　串口输出字符串 1

此时，按下开发板上的 Key2 按键，通过串口向 PC 端输出了字符串“key2_pressed”，如图 9-6 所示。

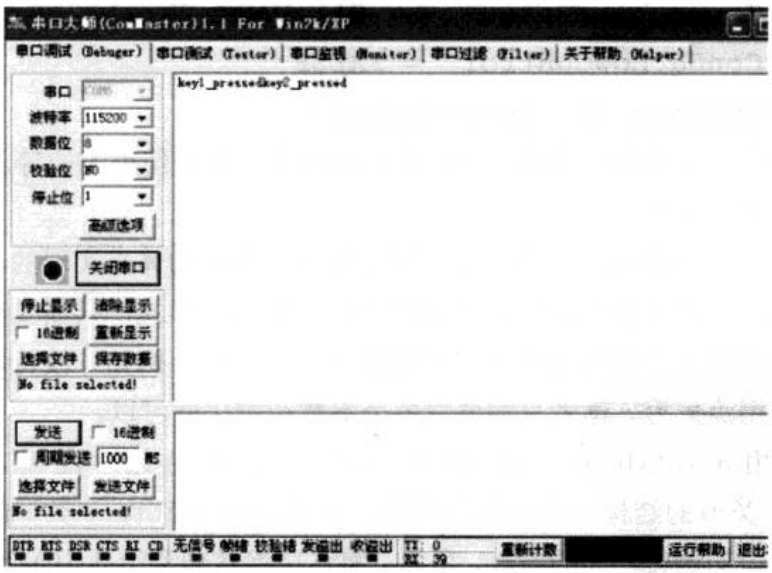


图 9-6　串口输出字符串 2

9.2 获取广告设备地址

每一个蓝牙 BLE 设备都有唯一的 48 位 IEEE 设备地址（BD_ADDR，Bluetooth

Device Address），它由三个部分构成：

▲ NAP(Non-significant Address Part)：无效地址部分，共 16 位；

▲ UAP(Upper Address Part)：高位地址部分，共 8 位，NAP 和 UAP 为制造商标识，是由 IEEE 分配的；

▲ LAP(Lower Address Part)：低位地址部分，共 24 位，是由制造商指定的序列号。

每个蓝牙设备的地址都是唯一的，因此，集中器设备就可以通过接收到的不同设备地址来区分不同的广告设备，并有选择的与广告设备建立网络连接。本实验实现的功能为：集中器设备在未与广告设备建立连接时，接收广告设备的地址并通过 LCD 和串口显示。

9.2.1 广告设备编程

本实验中，广告设备工程文件布局[●]如图 9-7 所示。

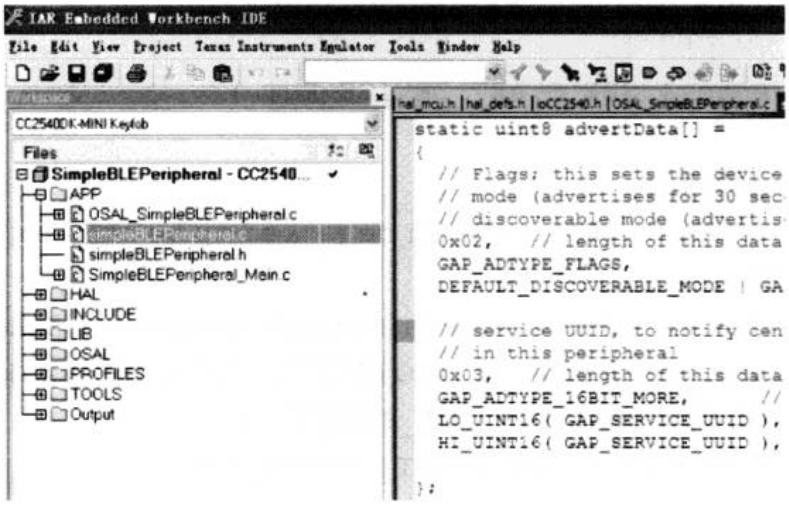


图 9-7 广告设备工程文件布局

广告设备通过按下 Key2 按键来触发广告或停止广告，当 Key2 按键按下时，产生按键中断（在第 9.1 小节中阐述了协议栈的中断处理机制），在按键中断处理函数中设置了一个 HAL_KEY_EVENT 事件，HAL 层发送一个 OSAL 消息给应用，产生一个 SYS_EVENT_MSG 事件，首先通过 SimpleBLEPeripheral_ProcessEvent 函数来对事件进行处理：

```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
```

● 使用 BLE 协议栈进行应用程序开发时，一般不需要建立新的工程，只需要在 TI BLE 协议栈软件包中提供的几个示例工程的基础上进行修改即可，TI 提供的 BLE 协议栈开发的示例工程存储在以下路径： C:\Texas Instruments\BLE-CC254x-1.2.1\Projects\ble，读者选择一个工程进行修改就可以使用，在本实验中，所使用的工程在以下路径：C:\Texas Instruments\BLE-CC254x-1.2.1\Projects\ble\SimpleBLEPeripheral\CC2540DB，打开 SimpleBLEPeripheral.eww 即可。

```
uint16 events )
   {
     VOID task_id;
     if ( events & SYS_EVENT_MSG )
     {
       uint8 *pMsg;
       if ( (pMsg = osal_msg_receive( simpleBLEPeripheral_
TaskID )) != NULL )
       {
         simpleBLEPeripheral_ProcessOSALMsg( (osal_event_hdr_
t *)pMsg );
         VOID osal_msg_deallocate( pMsg );
       }
       return (events ^ SYS_EVENT_MSG);
     }
   if ( events & SBP_START_DEVICE_EVT )
   {
   //注册了回调函数
   VOID GAPRole_StartDevice( &simpleBLEPeripheral_
PeripheralCBs );
   VOID GAPBondMgr_Register( &simpleBLEPeripheral_BondMgrCBs );
   osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_
PERIODIC_EVT,SBP_PERIODIC_EVT_PERIOD );
   return ( events ^ SBP_START_DEVICE_EVT );
   }
   ……
   }
```

判断为OSAL消息后，通过调用simpleBLEperipheral_ProcessOSALMsg函数来处理这个消息事件：

```
static void simpleBLEPeripheral_ProcessOSALMsg( osal_
event_hdr_t *pMsg )
   {
     switch ( pMsg->event )
     {
     #if defined( CC2540_MINIDK )
       case KEY_CHANGE:
         simpleBLEPeripheral_HandleKeys( ((keyChange_t *)
```

```
pMsg)->state, ((keyChange_t*)pMsg)->keys );
        break;
    #endif
    default:
      break;
    }
  }
```

进一步判断接收到的 OSAL 消息为按键事件，然后调用按键处理函数如下：

```
  static void simpleBLEPeripheral_HandleKeys( uint8 shift,
uint8 keys )
  {
    uint8 SK_Keys = 0;
    VOID shift;
    if ( keys & HAL_KEY_SW_1 )
    {
      SK_Keys |= SK_KEY_LEFT;
    }
    if ( keys & HAL_KEY_SW_2 )
    {
    SK_Keys |= SK_KEY_RIGHT;
  1  if( gapProfileState != GAPROLE_CONNECTED )
      {
      uint8 current_adv_enabled_status;
      uint8 new_adv_enabled_status;
  2   GAPRole_GetParameter(GAPROLE_ADVERT_ENABLED,&current_
adv_enabled_status );
  3  if( current_adv_enabled_status == FALSE )
        {
        new_adv_enabled_status = TRUE;
        }
    else
        {
  4     new_adv_enabled_status = FALSE;
        }
  5  GAPRole_SetParameter(GAPROLE_ADVERT_
ENABLED,sizeof(uint8),&new_adv_enabled_status );
      }
```

```
    }
  6  SK_SetParameter( SK_KEY_ATTR, sizeof ( uint8 ), &SK_
Keys );
  }
```

第1行，判断设备GAP状态是否为连接状态，如果未连接，则执行相应的处理程序。

第2行，读取设备当前的广告状态。

第3行，如果设备当前的广告状态为FALSE，即设备未广告时，将参数更改为设备开启广告。

第4行，如果设备当前的广告状态为TRUE，即设备广告时，参数更改为设备停止广告，即每一次按键按下，都会改变设备的广告状态。

第5行，将新的广告状态写入设备。

第6行，将按键状态值写入按键Profile的“特性”中，如果此时设备处于连接状态并且使能了按键通知，改变后的按键值将会通知给集中器设备。

设备的广告状态发生变化，即GAP状态发生了改变，此时调用GAP回调函数，在回调函数中处理GAP状态改变时，液晶屏上显示的内容。

```
static void peripheralStateNotificationCB( gaprole_States_
t newState )
{
  switch ( newState )
  {
  ……
  case GAPROLE_ADVERTISING:  //广告状态
  {
    #if (defined HAL_LCD) && (HAL_LCD == TRUE)
    HalLcdWriteString( "Advertising",  HAL_LCD_LINE_3 );
    #endif
  }
    break;
  case GAPROLE_CONNECTED:   //连接状态
    {
      #if (defined HAL_LCD) && (HAL_LCD == TRUE)
        HalLcdWriteString( "Connected",  HAL_LCD_LINE_3 );
      #endif
    }
    break;
case GAPROLE_WAITING:     //未连接状态
```

```
    {
      #if (defined HAL_LCD) && (HAL_LCD == TRUE)
        HalLcdWriteString( "Disconnected",HAL_LCD_LINE_3 );
        #endif
      }
      break;
      ……
    default:
      {
        #if (defined HAL_LCD) && (HAL_LCD == TRUE)
          HalLcdWriteString( "",  HAL_LCD_LINE_3 );
        #endif
      }
      break;
    }
    gapProfileState = newState;
    #if !defined( CC2540_MINIDK )
    VOID gapProfileState;
    #endif
  }
```

设备发送的广告中包含的数据内容如下：

```
static uint8 advertData[] =
{
1  0x02,
2  GAP_ADTYPE_FLAGS,   //0x01
3  DEFAULT_DISCOVERABLE_MODE |
GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,  //0x05
4  0x03,
5  GAP_ADTYPE_16BIT_MORE,    //0x02
6  LO_UINT16( SIMPLEPROFILE_SERV_UUID ),  //0xF0
7  HI_UINT16( SIMPLEPROFILE_SERV_UUID ),  //0xFF
};
```

第 1 行，广告中设置发现模式的数据段，占用 2 字节数据长度。

第 2 行，广告类型标志。

第 3 行，定义节点设备的可发现模式。根据预编译选项中是否包含“CC2540_MINIDK”宏定义，来决定节点设备的广告状态为受限的可发现模式或为不受限的可发现模式。

第4行，广告中设置GAP基本服务UUID的数据段，占用3字节数据长度。

第5行，定义UUID为16bit，即2字节数据长度。

第6、7行，分别为UUID的低8位和高8位。只有GAP服务的UUID相匹配，两设备才能建立连接。

9.2.2 集中器设备编程

集中器设备的工程文件布局如图9-8所示。

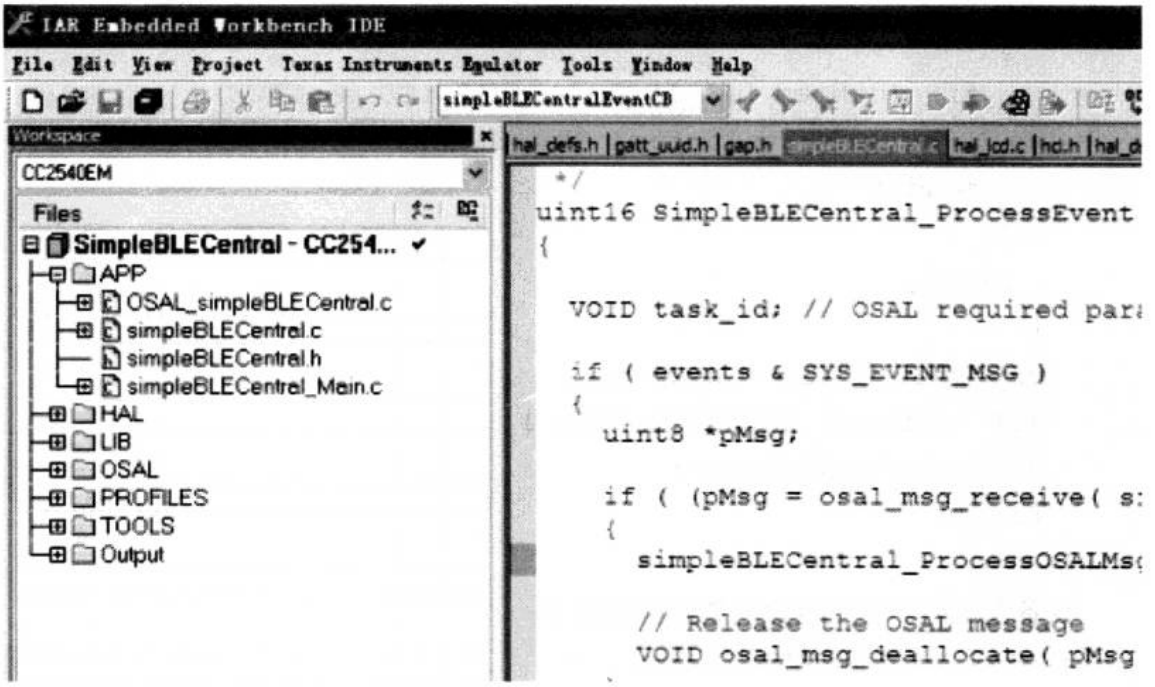


图9-8 集中器设备的工程文件布局

修改“SimpleBLECentral.c”文件内容如下（新增加的部分以加粗字体显示）：

```
#include "hal_uart.h"
```

向SimpleBLEPeripheral_Init( )函数中添加：

```
1  halUARTCfg_t uartConfig;
2  uartConfig.configured      = TRUE;
3  uartConfig.baudRate      = HAL_UART_BR_115200;
4  uartConfig.flowControl   = FALSE;
5  HalUARTOpen (0, &uartConfig);
```

第1～4行，配置串口参数，包括串口波特率、是否使用流控等。

第5行，初始化串口。

任务事件处理函数内容如下：

```
uint16 SimpleBLECentral_ProcessEvent( uint8 task_id,
```

```
uint16 events )
    {
      VOID task_id;
      if ( events & SYS_EVENT_MSG )
      {
        uint8 *pMsg;
        if ( (pMsg = osal_msg_receive( simpleBLETaskId )) !=
NULL )
        {
          simpleBLECentral_ProcessOSALMsg( (osal_event_hdr_t *)
pMsg );
          VOID osal_msg_deallocate( pMsg );
        }
        return (events ^ SYS_EVENT_MSG);
      }
      if ( events & START_DEVICE_EVT )
      {
    //注册了回调函数，每当GAP状态改变时，调用GAP回调函数来处理
      VOID GAPCentralRole_StartDevice( (gapCentralRoleCB_t *)
&simpleBLERoleCB );
      GAPBondMgr_Register((gapBondCBs_t *)&simpleBLEBondCB );
      return ( events ^ START_DEVICE_EVT );
      }
      if ( events & START_DISCOVERY_EVT )
      {
        simpleBLECentralStartDiscovery( );
        return ( events ^ START_DISCOVERY_EVT );
      }
      return 0;
    }
```

当集中器设备扫描处于广告状态的设备时，GAP状态发生改变，调用GAP回调函数进行处理：

```
    static void simpleBLECentralEventCB( gapCentralRoleEvent_
t *pEvent )
    {
      switch ( pEvent->gap.opcode )
      {
```

```
        case GAP_DEVICE_INIT_DONE_EVENT:  // 设备初始化
          {
          LCD_WRITE_STRING( "BLE Central", HAL_LCD_LINE_1 );
          LCD_WRITE_STRING( bdAddr2Str( pEvent->initDone.
devAddr ),HAL_LCD_LINE_2 );
    }
      break;
    case GAP_DEVICE_INFO_EVENT:  // 处理扫描发现的设备信息
      {
        if ( DEFAULT_DEV_DISC_BY_SVC_UUID == TRUE )
          {
            if ( simpleBLEFindSvcUuid( GAP_SERVICE_UUID,
            pEvent->deviceInfo.pEvtData, pEvent->deviceInfo.
dataLen ) )
            {
              uint8 buf[6];
              simpleBLEAddDeviceInfo( pEvent->deviceInfo.addr,
              pEvent->deviceInfo.addrType );// 向设备发现列表中添
加一个设备
          LCD_WRITE_STRING( "AdvDevice_Addr:", HAL_LCD_LINE_3 );
          LCD_WRITE_STRING( bdAddr2Str( pEvent->deviceInfo.
addr ),HAL_LCD_LINE_4 );      // 通过 LCD 显示广告设备的地址
          osal_memcpy(buf, pEvent->deviceInfo.addr, B_ADDR_
LEN);
          HalUARTWrite(0, buf,6) ;    // 通过串口输出广告设备的地址
          osal_memset( buf,0,6);
          HalLedSet(HAL_LED_1,HAL_LED_MODE_ON);    // 点亮 LED 灯
        }
      }
    }
    break;
    case GAP_DEVICE_DISCOVERY_EVENT:      // 设备发现完成
      {
        simpleBLEScanning = FALSE;
        if ( DEFAULT_DEV_DISC_BY_SVC_UUID == FALSE )
        {
          simpleBLEScanRes = pEvent->discCmpl.numDevs;
```

```
        osal_memcpy( simpleBLEDevList, pEvent->discCmpl.
pDevList,(sizeof( gapDevRec_t ) * pEvent->discCmpl.numDevs)
);
      }
       LCD_WRITE_STRING_VALUE( "Devices Found",
simpleBLEScanRes, 10,HAL_LCD_LINE_1 );
      if ( simpleBLEScanRes > 0 )
      {
        LCD_WRITE_STRING( "<- To Select", HAL_LCD_LINE_2 );
      }
      simpleBLEScanIdx = simpleBLEScanRes;
    }
    break;
  case GAP_LINK_ESTABLISHED_EVENT:  //建立连接
    {
      if ( pEvent->gap.hdr.status == SUCCESS )
      {
      simpleBLEState = BLE_STATE_CONNECTED;
       simpleBLEConnHandle = pEvent->linkCmpl.
connectionHandle;
      simpleBLEProcedureInProgress = TRUE;
      if ( simpleBLECharHdl == 0 )
      {
        osal_start_timerEx( simpleBLETaskId, START_
DISCOVERY_EVT,DEFAULT_SVC_DISCOVERY_DELAY );
      }
      LCD_WRITE_STRING( "Connected", HAL_LCD_LINE_1 );
      LCD_WRITE_STRING( bdAddr2Str( pEvent->linkCmpl.
devAddr ),HAL_LCD_LINE_2 );
      }
      else
      {
        simpleBLEState = BLE_STATE_IDLE;
        simpleBLEConnHandle = GAP_CONNHANDLE_INIT;
        simpleBLERssi = FALSE;
        simpleBLEDiscState = BLE_DISC_STATE_IDLE;
        LCD_WRITE_STRING("Connect Failed", HAL_LCD_LINE_1 );
```

```
        LCD_WRITE_STRING_VALUE( "Reason:", pEvent->gap.hdr.
status, 10,HAL_LCD_LINE_2 );
      }
    }
    break;
  case GAP_LINK_TERMINATED_EVENT:  //连接断开
    {
      simpleBLEState = BLE_STATE_IDLE;
      simpleBLEConnHandle = GAP_CONNHANDLE_INIT;
      simpleBLERssi = FALSE;
      simpleBLEDiscState = BLE_DISC_STATE_IDLE;
      simpleBLECharHdl = 0;
      simpleBLEProcedureInProgress = FALSE;
      LCD_WRITE_STRING( "Disconnected", HAL_LCD_LINE_1 );
      LCD_WRITE_STRING_VALUE("Reason:",pEvent->linkTerminate.
reason,10,HAL_LCD_LINE_2);
      }
    break;
  case GAP_LINK_PARAM_UPDATE_EVENT:  //连接参数更新
      {
        LCD_WRITE_STRING( "Param Update", HAL_LCD_LINE_1 );
      }
      break;
    default:
      break;
    }
  }
```

使用协议栈的串口功能，需要在预编译选项中写入宏定义“HAL_UART=TRUE”，如图9-9所示。

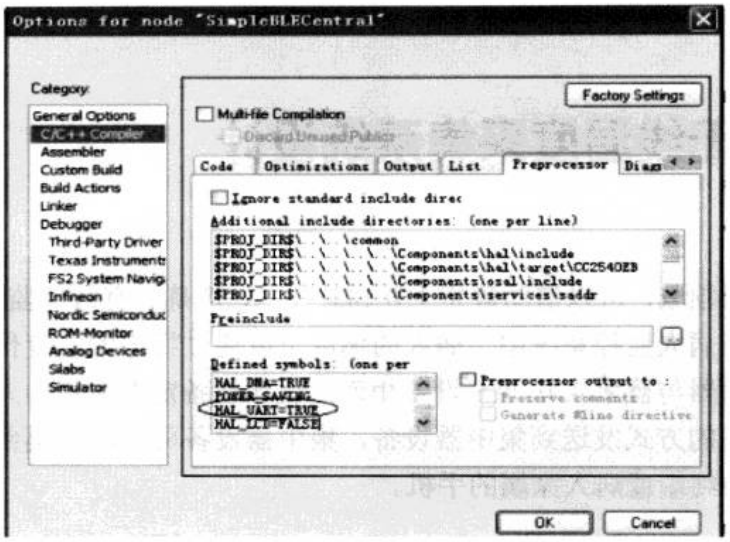


图 9-9　预编译中写入串口宏定义

9.2.3　实例测试

将广告设备代码下载到开发板 A 中，将集中器设备代码下载到开发板 B 中，将开发板 B 通过串口与 PC 连接，打开串口调试助手，将波特率设为 115200。当集中器设备扫描到处于广告状态的设备时，在 LCD 上显示“AdvDevice_Addr:”，“0x0017EA8F875C”，即为广告设备的地址，同时，通过串口向 PC 端输出数据“5C 87 8F EA 17 00”，如图 9-10 所示。

图 9-10　串口输出广告设备地址

9.3 无线温度采集系统设计

在医学应用领域，可以借助蓝牙 4.0 BLE 技术，准确、有效地监测病人的体温、血压等信息。在病人监控系统中，病人的体温可以通过特定的温度传感器采集，因此，可以将传感器与蓝牙 4.0 BLE 设备相连，蓝牙设备定期采集病人的体温信息，并将数据以无线的方式发送到集中器设备，集中器设备可以将采集到的温度数据传输给医生的监控终端或病人家属的手机。

9.3.1 实验原理及流程图

温度传感器采集到的温度数据存放到节点设备属性表的特性中，然后节点设备周期性地将温度数据通知给集中器设备。温度传感器可选择 DS18B20、SHT10 等器件，关于温度传感器的选型在此不作赘述，读者可以根据测量需要自行选择。系统设计流程如图 9-11 所示。

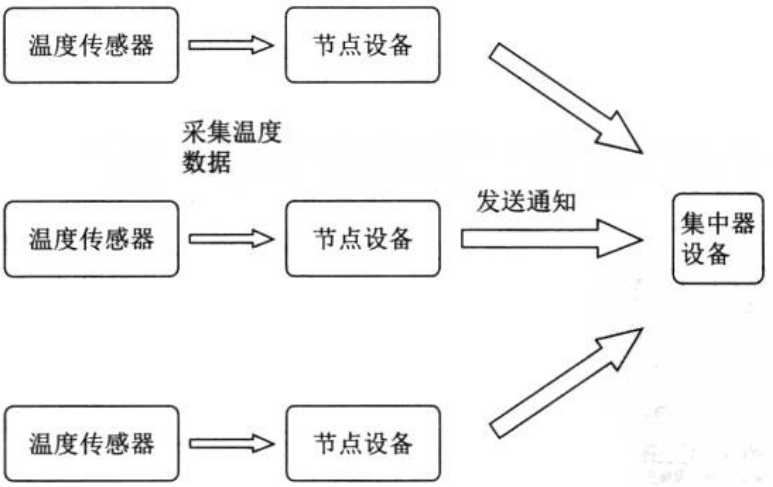


图 9-11 无线温度采集系统

9.3.2 节点设备编程

节点设备采用“thermometer”示例工程，工程文件布局如图 9-12 所示。

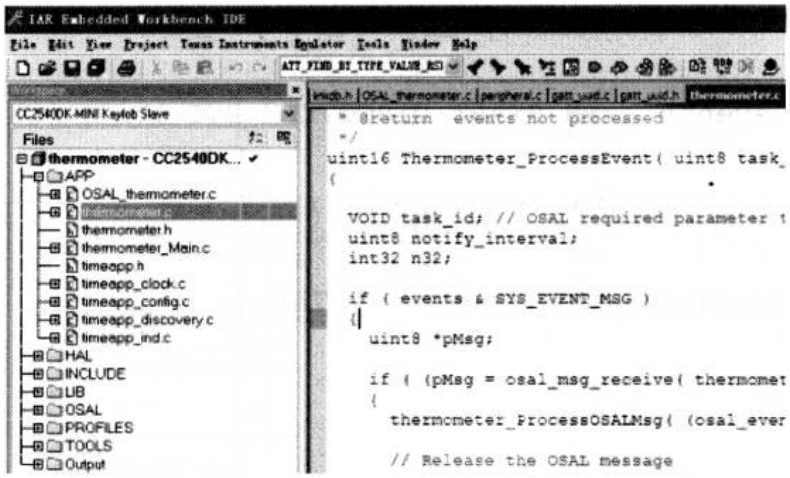


图 9-12 “thermometer”示例工程文件布局

应用层的事件处理函数代码如下所示：

```
uint16 Thermometer_ProcessEvent( uint8 task_id, uint16 events )
{
  VOID task_id;
  uint8 notify_interval;
  int32 n32;
  if ( events & SYS_EVENT_MSG )
  {
    uint8 *pMsg;
    if ( (pMsg = osal_msg_receive( thermometerTaskId )) != NULL )
    {
      thermometer_ProcessOSALMsg( (osal_event_hdr_t *)pMsg );
      VOID osal_msg_deallocate( pMsg );
    }
      return (events ^ SYS_EVENT_MSG);
  }
  if ( events & TH_START_DEVICE_EVT )
  {
    VOID GAPRole_StartDevice( &thermometer_PeripheralCBs ); // 回调函数
    VOID GAPBondMgr_Register( &thermometer_BondMgrCBs );
    updateUI();
```

```
        return ( events ^ TH_START_DEVICE_EVT );
      }
      if ( events & TH_START_DISCOVERY_EVT )
      {
        if ( timeAppPairingStarted )
        {
          timeAppDiscPostponed = TRUE;
        }
        else
        {
          timeAppDiscState = timeAppDiscStart();
        }
        timeAppDiscState = timeAppDiscStart();
        return ( events ^ TH_START_DISCOVERY_EVT );
      }
      if ( events & TH_PERIODIC_IMEAS_EVT )
      {
        performPeriodicImeasTask();     // 执行周期任务
        return (events ^ TH_PERIODIC_IMEAS_EVT);
      }
      return 0;
    }
```

周期任务中，每隔一段时间发送一次采集到的数据，performPeriodicImeasTask()函数内容如下所示：

```
    static void performPeriodicImeasTask( void )
    {
      if (gapProfileState == GAPROLE_CONNECTED)
      {
        thermometerImeasNotify(); // 使用通知功能发送采集到的温度数据
        osal_start_timerEx( thermometerTaskId, TH_PERIODIC_
IMEAS_EVT, 1000 );
      }
    }
```

thermometerImeasNotify()函数即为处理温度数据通知的函数，代码内容如下：

```
    static void thermometerImeasNotify(void)
    {
```

```
    uint8 *p = thermometerIMeas.value;
    uint32 temperature;
    uint8 flags = thermometerFlags[thermometerFlagsIdx];
    *p++ = flags;
    if(flags & THERMOMETER_FLAGS_FARENHEIT)
    {
      temperature =  (thermometerCelcius *9/5) +320;
    }
    else
    {
       temperature = thermometerCelcius;
    }
    temperature = 0xFF000000 | temperature;
    osal_buffer_uint32( p, temperature );
    p+=4;    //4 字节长度的温度值
    if(flags & THERMOMETER_FLAGS_TYPE)
    {
      uint8 site;
      Thermometer_GetParameter( THERMOMETER_TYPE, &site );
      *p++ =  site;
    }
    thermometerIMeas.len = (uint8) (p - thermometerIMeas.
value);
    if(temperatureIMeasCharConfig == true)
    {
    Thermometer_IMeasNotify( gapConnHandle,
&thermometerIMeas);
    }
  }
```

进一步调用 Thermometer_IMeasNotify() 函数如下所示：

```
  bStatus_t Thermometer_IMeasNotify( uint16 connHandle,
attHandleValueNoti_t *pNoti)
  {
    uint16 value = GATTServApp_ReadCharCfg( connHandle,
thermometerIMeasConfig );
    if ( value & GATT_CLIENT_CFG_NOTIFY )// 判断通知功能是否开启
    {
```

```
    pNoti->handle = thermometerAttrTbl[THERMOMETER_IMEAS_
VALUE_POS].handle ; //句柄
    return GATT_Notification( connHandle, pNoti, FALSE );
//发送通知
    }
    return bleIncorrectMode;
  }
```

9.3.3 集中器设备编程

集中器设备使用“SimpleBLECentral”示例工程，应用层初始化函数代码如下所示：

```
void SimpleBLECentral_Init( uint8 task_id )
{
  simpleBLETaskId = task_id;
  {
    uint8 scanRes = DEFAULT_MAX_SCAN_RES;
    GAPCentralRole_SetParameter ( GAPCENTRALROLE_MAX_
SCAN_RES,sizeof( uint8 ), &scanRes );
  }
  GAP_SetParamValue( TGAP_GEN_DISC_SCAN, DEFAULT_SCAN_
DURATION );
  GAP_SetParamValue( TGAP_LIM_DISC_SCAN, DEFAULT_SCAN_
DURATION );
  GGS_SetParameter( GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_
LEN, (uint8*) simpleBLEDeviceName );
  {
    uint32 passkey = DEFAULT_PASSCODE;
    uint8 pairMode = DEFAULT_PAIRING_MODE;
    uint8 mitm = DEFAULT_MITM_MODE;
    uint8 ioCap = DEFAULT_IO_CAPABILITIES;
    uint8 bonding = DEFAULT_BONDING_MODE;
    GAPBondMgr_SetParameter( GAPBOND_DEFAULT_PASSCODE,
sizeof( uint32 ),&passkey );
    GAPBondMgr_SetParameter( GAPBOND_PAIRING_MODE, sizeof
( uint8 ),&pairMode );
    GAPBondMgr_SetParameter( GAPBOND_MITM_PROTECTION,
sizeof( uint8 ),&mitm );
```

```
      GAPBondMgr_SetParameter( GAPBOND_IO_CAPABILITIES,
sizeof( uint8 ),&ioCap );
      GAPBondMgr_SetParameter( GAPBOND_BONDING_ENABLED,
sizeof( uint8 ),&bonding );
    }
    VOID GATT_InitClient();  //初始化GATT客户端
    GATT_RegisterForInd( simpleBLETaskId );//注册任务来接收通知
    GGS_AddService( GATT_ALL_SERVICES );
    GATTServApp_AddService( GATT_ALL_SERVICES );
    RegisterForKeys( simpleBLETaskId );
    HalLedSet( (HAL_LED_1 | HAL_LED_2), HAL_LED_MODE_OFF );
    osal_set_event( simpleBLETaskId, START_DEVICE_EVT );
    }
```

应用层任务事件处理函数代码如下所示：

```
    uint16 SimpleBLECentral_ProcessEvent( uint8 task_id,
uint16 events )
    {
    VOID task_id;
    if ( events & SYS_EVENT_MSG )
    {
    uint8 *pMsg;
    if ( (pMsg = osal_msg_receive( simpleBLETaskId )) != NULL )
    {
      simpleBLECentral_ProcessOSALMsg( (osal_event_hdr_t *)
pMsg );
      VOID osal_msg_deallocate( pMsg );
    }
    return (events ^ SYS_EVENT_MSG);
    }
    if ( events & START_DEVICE_EVT )
    {
      VOID GAPCentralRole_StartDevice( (gapCentralRoleCB_t *)
&simpleBLERoleCB );
      GAPBondMgr_Register((gapBondCBs_t *)&simpleBLEBondCB );
      return ( events ^ START_DEVICE_EVT );
    }
    if ( events & START_DISCOVERY_EVT )
```

```
    {
      simpleBLECentralStartDiscovery( );
      return ( events ^ START_DISCOVERY_EVT );
    }
    return 0;
  }
```

在GATT消息处理函数中处理接收到的通知，代码如下所示：

```
static void simpleBLECentralProcessGATTMsg( gattMsgEvent_
t *pMsg )
  {
    if ( simpleBLEState != BLE_STATE_CONNECTED )
    {
      return;
    }
    if( pMsg->method == ATT_HANDLE_VALUE_NOTI )// 用来接收通知
    {
      uint8 valueRead=pMsg->msg.handleValueNoti.value[0];
      LCD_WRITE_STRING_VALUE( "Notification:", valueRead,
10,HAL_LCD_LINE_5 );
    }
    ......
    else if ( simpleBLEDiscState != BLE_DISC_STATE_IDLE )
    {
      simpleBLEGATTDiscoveryEvent( pMsg );
    }
  }
```

9.4 蓝牙4.0 BLE无线传感器网络组建

无线传感器网络（Wireless Sensor Network，WSN）由部署在被检测区域的大量微型传感器节点组成，这些节点通过无线通信的方式形成一个网络系统，用于感知、采集和处理网络覆盖区域中被感知对象的信息。蓝牙4.0 BLE设备在没有数据发送时，可处于休眠状态，电流仅为0.4μA，直到需要进行数据发送时才被唤醒，因此非常适合用作对功耗有严格要求的无线传感器网络节点。

9.4.1 集中器与多个节点设备建立连接

一个蓝牙 4.0 BLE 集中器设备可以同时与三个节点设备保持网络连接，当网络中的一个节点设备发送完数据，断开连接之后，又可以有新的节点设备加入网络。而且每一个节点设备与集中器断开连接后，又可以加入另一个网络，这样就间接增加了网络中的设备数量。网络拓扑如图 9-13 所示。

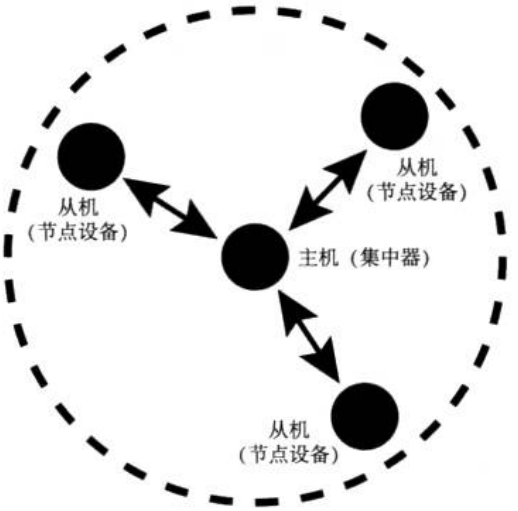


图 9-13　网络拓扑

集中器设备示例工程文件布局如图 9-14 所示。

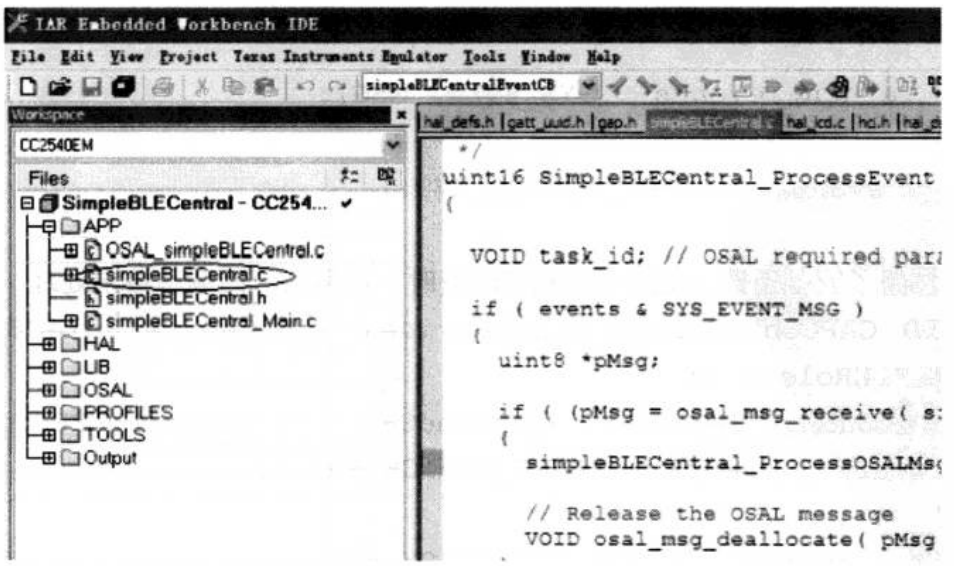


图 9-14　集中器设备示例工程文件布局

修改“SimpleBLECentral.c”文件内容如下（新增加的部分以加粗字体显示）：

```
#include "hal_uart.h"
```

向SimpleBLEPeripheral_Init( )函数中添加：

```
1  halUARTCfg_t uartConfig;
2  uartConfig.configured      = TRUE;
3  uartConfig.baudRate      = HAL_UART_BR_115200;
4  uartConfig.flowControl  = FALSE;
5  HalUARTOpen (0, &uartConfig);
```

第1～4行，配置串口参数，包括串口波特率、是否使用流控等。

第5行，初始化串口。

任务事件处理函数内容如下：

```
uint16 SimpleBLECentral_ProcessEvent( uint8 task_id,
uint16 events )
{
  VOID task_id;
  if ( events & SYS_EVENT_MSG )
  {
    uint8 *pMsg;
    if ( (pMsg = osal_msg_receive( simpleBLETaskId )) != NULL )
    {
      simpleBLECentral_ProcessOSALMsg( (osal_event_hdr_t *)
pMsg );
      VOID osal_msg_deallocate( pMsg );
    }
    return (events ^ SYS_EVENT_MSG);
  }
if ( events & START_DEVICE_EVT )
{
//注册了回调函数，每当GAP状态改变时，调用GAP回调函数来处理
VOID GAPCentralRole_StartDevice( (gapCentralRoleCB_t *)
&simpleBLERoleCB );
  GAPBondMgr_Register( (gapBondCBs_t *) &simpleBLEBondCB );
  return ( events ^ START_DEVICE_EVT );
  }
  if ( events & START_DISCOVERY_EVT )
  {
    simpleBLECentralStartDiscovery( );
    return ( events ^ START_DISCOVERY_EVT );
```

```
    }
    return 0;
  }
```

任务事件处理函数中调用 simpleBLECentral_ProcessOSALMsg() 函数来处理 OSAL 消息，simpleBLECentral_ProcessOSALMsg() 函数内容如下：

```
  static void simpleBLECentral_ProcessOSALMsg( osal_event_
hdr_t *pMsg )
  {
    switch ( pMsg->event )
    {
      case KEY_CHANGE:
        simpleBLECentral_HandleKeys( ((keyChange_t *)pMsg)-
>state, ((keyChange_t*)pMsg)->keys );
        break;
    case GATT_MSG_EVENT:
      simpleBLECentralProcessGATTMsg( (gattMsgEvent_t *)
pMsg );
      break;
    }
  }
```

从上述代码中可以看到，若判断 OSAL 消息为按键事件，则调用按键处理函数来进行按键事件的处理，simpleBLECentral_HandleKeys 函数内容如下：

```
  static void simpleBLECentral_HandleKeys( uint8 shift,
uint8 keys )
  {
    (void)shift;
    if ( keys & HAL_KEY_SW_1 )  //Key1 按键按下，扫描广告设备
    {
      if ( !simpleBLEScanning )
      {
        simpleBLEScanning = TRUE;
        simpleBLEScanRes = 0;
        LCD_WRITE_STRING( "Discovering...", HAL_LCD_LINE_1 );
        LCD_WRITE_STRING( "", HAL_LCD_LINE_2 );
    GAPCentralRole_StartDiscovery(DEFAULT_DISCOVERY_
MODE,DEFAULT_DISCOVERY_ACTIVE_SCAN, DEFAULT_DISCOVERY_WHITE_
LIST );
```

```
        }
        else
        {
          GAPCentralRole_CancelDiscovery();
        }
    }
    if ( keys & HAL_KEY_SW_2 )  //Key2 键按下，选择要与之建立连接
的设备
    {
      if ( !simpleBLEScanning && simpleBLEScanRes > 0 )
      {
        simpleBLEScanIdx++;
        if ( simpleBLEScanIdx >= simpleBLEScanRes )
        {
          simpleBLEScanIdx = 0;
        }
        LCD_WRITE_STRING_VALUE( "Device", simpleBLEScanIdx +
1,10,HAL_LCD_LINE_1 );
        LCD_WRITE_STRING( bdAddr2Str( simpleBLEDevList[simple
BLEScanIdx].addr ),HAL_LCD_LINE_2 );
      }
    }
    if ( keys & HAL_KEY_SW_2 )  //Key2 键按下，与广播设备建立连接
    {
      uint8 addrType;
      uint8 *peerAddr;
      if ( simpleBLEScanRes > 0 )
      {
        peerAddr = simpleBLEDevList[simpleBLEScanIdx].addr;
        addrType = simpleBLEDevList[simpleBLEScanIdx].
addrType;
        GAPCentralRole_EstablishLink( DEFAULT_LINK_HIGH_DUTY_
CYCLE,DEFAULT_LINK_WHITE_LIST, addrType, peerAddr );  //发起
建立连接请求
        LCD_WRITE_STRING( "Connecting", HAL_LCD_LINE_1 );
        LCD_WRITE_STRING( bdAddr2Str( peerAddr ), HAL_LCD_
LINE_2 );
```

```
    }
    }
  }
```

当集中器设备扫描处于广告状态的设备或发起建立连接的请求时，GAP 状态发生了改变，调用 GAP 回调函数进行处理：

```
  static void simpleBLECentralEventCB( gapCentralRoleEvent_
t *pEvent )
  {
    switch ( pEvent->gap.opcode )
    {
      case GAP_DEVICE_INIT_DONE_EVENT:  // 设备初始化
      {
        LCD_WRITE_STRING( "BLE Central", HAL_LCD_LINE_1 );
        LCD_WRITE_STRING( bdAddr2Str( pEvent->initDone.
devAddr ),HAL_LCD_LINE_2 );
      }
      break;
  case GAP_DEVICE_INFO_EVENT:  // 处理扫描发现的设备信息
    {
      if ( DEFAULT_DEV_DISC_BY_SVC_UUID == TRUE )
      {
         if ( simpleBLEFindSvcUuid( GAP_SERVICE_UUID,pEvent-
>deviceInfo.pEvtData, pEvent->deviceInfo.dataLen ) )
      {
        simpleBLEAddDeviceInfo( pEvent->deviceInfo.
addr,pEvent->deviceInfo.addrType );    // 向设备发现列表中添加一
个设备
        LCD_WRITE_STRING( "AdvDevice_Addr:", HAL_LCD_LINE_3 );
        LCD_WRITE_STRING( bdAddr2Str( pEvent->deviceInfo.
addr ),HAL_LCD_LINE_4 );          // 通过 LCD 显示广告设备的地址
        }
      }
    }
    break;
  case GAP_DEVICE_DISCOVERY_EVENT:  // 设备发现完成
  {
    simpleBLEScanning = FALSE;
```

```
      if ( DEFAULT_DEV_DISC_BY_SVC_UUID == FALSE )
      {
        simpleBLEScanRes = pEvent->discCmpl.numDevs;
        osal_memcpy( simpleBLEDevList, pEvent->discCmpl.
pDevList,(sizeof( gapDevRec_t ) * pEvent->discCmpl.numDevs) );
      }
        LCD_WRITE_STRING_VALUE( "Devices
Found",simpleBLEScanRes, 10,HAL_LCD_LINE_1 );
    if ( simpleBLEScanRes > 0 )
      {
        LCD_WRITE_STRING( "<- To Select", HAL_LCD_LINE_2 );
      }
      simpleBLEScanIdx = simpleBLEScanRes;
      }
      break;
    case GAP_LINK_ESTABLISHED_EVENT:  //建立连接
      {
      if ( pEvent->gap.hdr.status == SUCCESS )
        {
        uint8 buf[9];   //开辟一个缓冲区
        simpleBLEState = BLE_STATE_CONNECTED;
        simpleBLEConnHandle = pEvent->linkCmpl.
connectionHandle;
      simpleBLEProcedureInProgress = TRUE;
      if ( simpleBLECharHdl == 0 )
      {
        osal_start_timerEx( simpleBLETaskId, START_DISCOVERY_
EVT,DEFAULT_SVC_DISCOVERY_DELAY );
      }
        osal_memcpy(buf,"connected",9);
        HalUARTWrite(0, buf,9) ;  //通过串口输出广告设备的地址
        osal_memset( buf,0,9);
        HalLedSet(HAL_LED_1,HAL_LED_MODE_ON);  //点亮LED灯
        LCD_WRITE_STRING( "Connected", HAL_LCD_LINE_1 );
        LCD_WRITE_STRING( bdAddr2Str( pEvent->linkCmpl.
devAddr ),HAL_LCD_LINE_2 );
        }
```

```
        else
        {
          simpleBLEState = BLE_STATE_IDLE;
          simpleBLEConnHandle = GAP_CONNHANDLE_INIT;
          simpleBLERssi = FALSE;
          simpleBLEDiscState = BLE_DISC_STATE_IDLE;
          LCD_WRITE_STRING( "Connect Failed", HAL_LCD_LINE_1 );
          LCD_WRITE_STRING_VALUE( "Reason:", pEvent->gap.hdr.
status, 10,HAL_LCD_LINE_2 );
        }
      }
      break;
    case GAP_LINK_TERMINATED_EVENT:  //断开连接
      {
        simpleBLEState = BLE_STATE_IDLE;
        simpleBLEConnHandle = GAP_CONNHANDLE_INIT;
        simpleBLERssi = FALSE;
        simpleBLEDiscState = BLE_DISC_STATE_IDLE;
        simpleBLECharHdl = 0;
        simpleBLEProcedureInProgress = FALSE;
        LCD_WRITE_STRING( "Disconnected", HAL_LCD_LINE_1 );
        LCD_WRITE_STRING_VALUE( "Reason:", pEvent-
>linkTerminate.reason,10, HAL_LCD_LINE_2 );
        }
        break;
      case GAP_LINK_PARAM_UPDATE_EVENT:  //更新连接参数
          {
            LCD_WRITE_STRING( "Param Update", HAL_LCD_LINE_1 );
          }
          break;
        default:
          break;
      }
    }
```

9.4.2 实例测试

将代码编译并下载到开发板中，将开发板与PC通过串口线连接，打开PC端的

串口调试助手，将波特率设置为115200，并使三个节点设备处于广告状态。此时，按下开发板上的Key1按键后，集中器设备扫描处于广告状态的节点设备，可以发现在LCD上显示“Device Found 3”，说明发现了三个节点设备，然后按下Key2按键，与其中一个节点设备建立了连接并通过串口显示“connected”；再次按下Key1按键，进行节点设备扫描时，在LCD上显示“Device Found 2”，说明剩余两个节点设备还没有建立连接，此时通过按下Key2按键即可将另一个节点设备加入网络，重复这个过程，可以使集中器设备同时与三个节点设备保持连接状态。集中器设备通过串口向PC端输出的字符串如图9-15所示。

图9-15 串口输出字符串

9.5 本章小结

本章对蓝牙4.0 BLE无线网络开发进行了浅显的讲解，尽量使读者熟悉开发的流程，需要说明的是，本章代码设计方面主要是考虑了初学者的易读性，在实际项目开发过程中，读者需要结合自身项目情况进行适当的代码优化即可。

[扩展阅读] 倒F天线设计

天线是无线通讯系统中的重要元件，广泛应用于无线电通信、广播、电视、雷达、导航、电子对抗、遥感、射电天文等工程。天线性能的好坏直接关系到整个系统的性能。天线是发射和接收无线电波的元件。发射天线的任务是将高频电流或导

行电磁波转变为空间中传播的无线电波，而接收天线则是将空间中传播的无线电波转变为高频电流或者导行电磁波。天线主要起着在高频电流或导行电磁波和空间传播的无线电波之间的换能作用。天线设计的准则就是使上述的换能效果得到最优。对天线性能的描述主要有：天线的增益、天线的方向图、天线的带宽、天线的波瓣宽度等等。

天线也是无线传感器网中的一个关键的元件，其作用是辐射和接收无线电波。基于无线传感器的特殊应用环境，无线传感器天线也有其自己独特的特点。由于无线传感器网络节点的很小，要求无线传感器的天线尽可能的要小型化；基于无线传感器低能耗的特点，要求天线具有较高的效率；而一般的无线传感器网络节点众多，天线的低成本也是要考虑的一个重要问题；针对无线传感器网络的应用也对天线的方向图提出了更高的要求。应用到无线传感器网络中的天线主要有平面倒 F 天线(PIFA)、微带天线以及单极子天线等等。下面以最常见的一种平面倒 F 天线为例说明天线的性能。

TI 官方天线设计参考中给出的倒 F 天线如图 9-16 所示，倒 F 天线尺寸参数如表 9-1 所示。

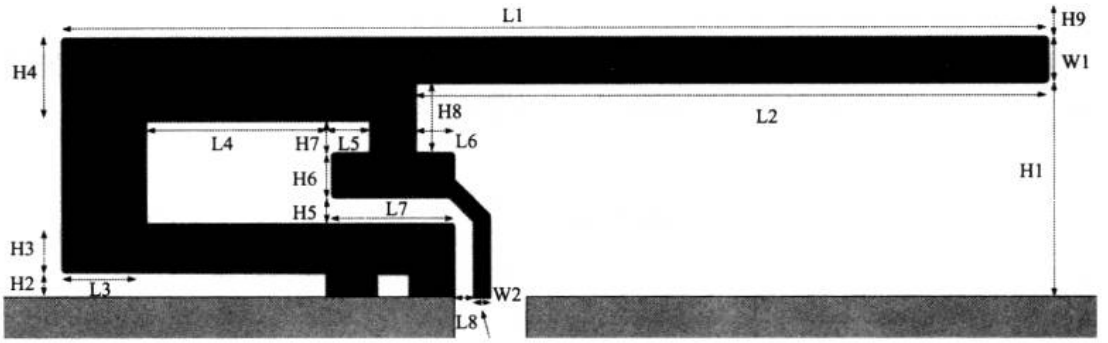


图 9-16　倒 F 天线

表 9-1　倒 F 天线尺寸参数

名称	参数	名称	参数
H1	5.70 mm	W2	0.46 mm
H2	0.74 mm	L1	25.58 mm
H3	1.29 mm	L2	16.40 mm
H4	2.21 mm	L3	2.18 mm
H5	0.66 mm	L4	4.80 mm
H6	1.21 mm	L5	1.00 mm
H7	0.80 mm	L6	1.00 mm
H8	1.80 mm	L7	3.20 mm
H9	0.61 mm	L8	0.45 mm
W1	1.21 mm		

在三维电磁场仿真软件CST中建立倒F天线仿真模型如图9-17所示。

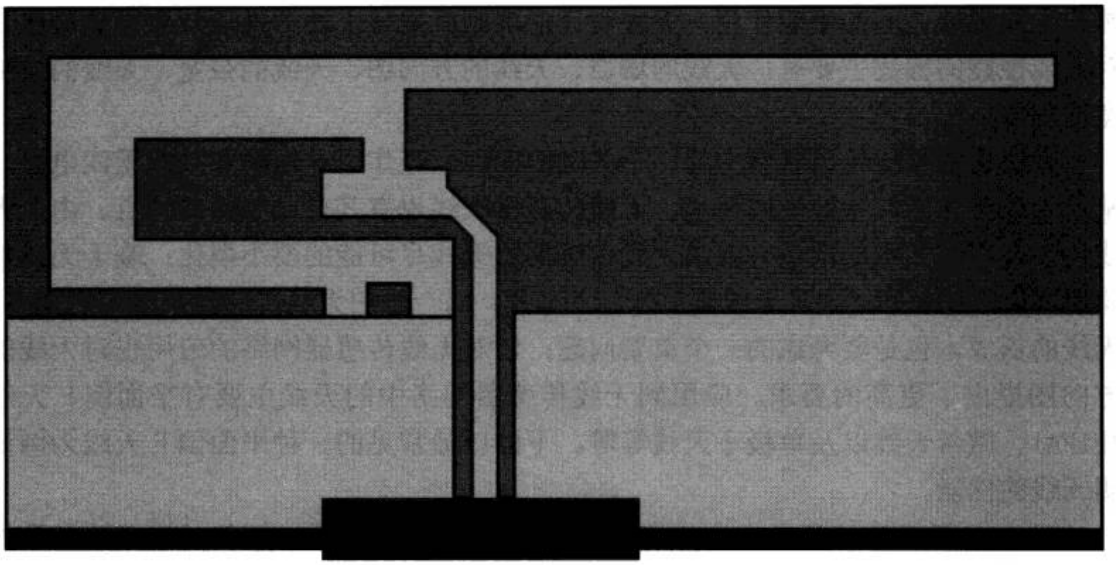


图9-17 倒F天线仿真模型

倒F天线S11参数如图9-18所示，可以看出该天线的-10dB带宽为2.10～2.70GHz，覆盖了无线传感器网工作频段2.45GHz。

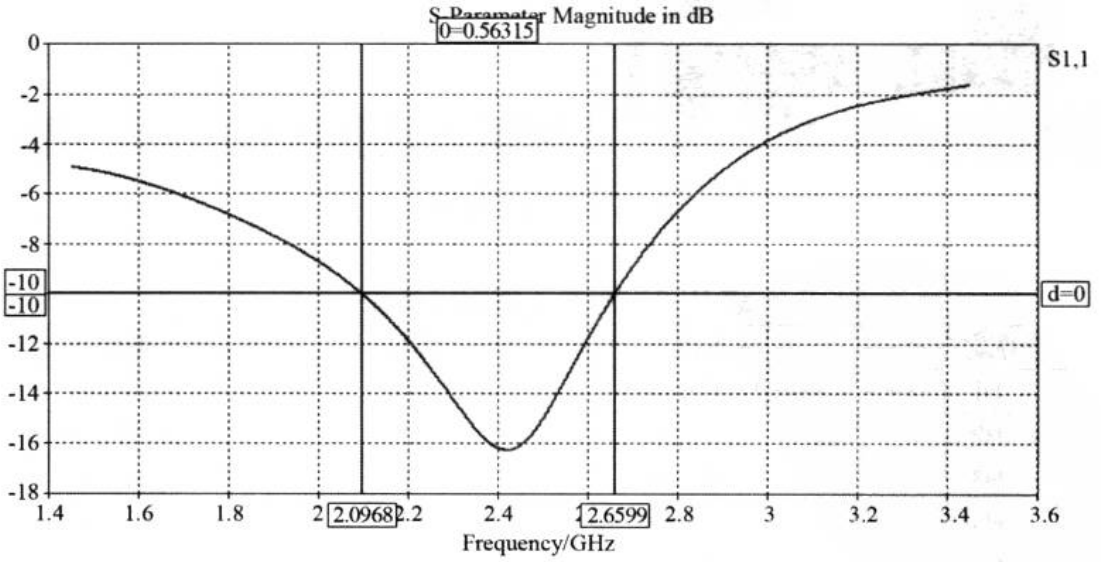


图9-18 倒F天线S11参数

倒F天线三维方向图如图9-19所示，可以看出平面倒F天线有较宽的辐射方向图。

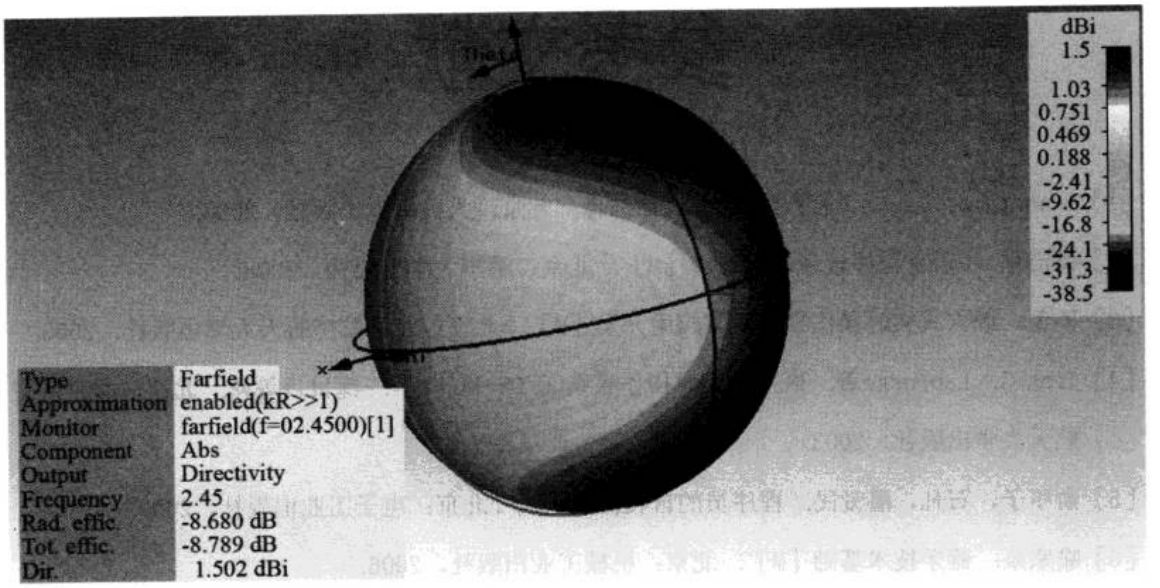


图 9-19　倒 F 天线三维方向图

参考文献

[1] Kenneth A. Reek. C和指针 [M]. 徐波译. 北京：人民邮电出版社，2008.

[2] 谭浩强. C语言程序设计(第三版) [M]. 北京：清华大学出版社，2005.

[3] 罗蕾. 嵌入式实时操作系统及其应用开发 [M]. 北京：北京航空航天大学出版社，2005.

[4] Jean J. Labrosse著. 嵌入式实时操作系统uC/OS-II [M]. 邵贝贝等译. 北京：北京航空航天大学出版社，2003.

[5] 俞甲子，石凡，潘爱民. 程序员的自我修养 [M].北京：电子工业出版社，2009.

[6] 喻宗泉. 蓝牙技术基础 [M]. 北京：机械工业出版社，2006.

[7] 钱志鸿. 蓝牙技术原理、开发与应用 [M]. 北京:北京航空航天大学出版社，2006.

[8] Using the ADC to Measure Supply Voltage[EB/OL].http://www.ti.com/lit/an/swra100a/swra100a.pdf

[9] CC2540 Bluetooth Low Energy Software Developer's Guide[EB/OL].http://www.ti.com/tool/ble-stack

[10] CC2540 Bluetooth Low Energy Sample Application Guide[EB/OL].http://www.ti.com /tool/ble-stack

[11] Core_V4.0[EB/OL].http://www.bluetooth.org/Technical/Specifications/adopted.htm

北航 C1640724

有关此电子图书的说明

由于本人工作的便利条件，可以帮助您提供各种中文电子图书，均为质量清晰的PDF格式文件，且清晰度要高于网上大量传播的超星PDG格式的图书。方便阅读和携带，且价格比纸张图书便宜。只要图书出版不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您提供PDF格式的电子版本。所以，当您想要看什么图书时，可以联系我。我的联系QQ为：2642294141，大家可以在QQ上联系我或给我留言。价格为每200页/5.00元，不足200页按照200页计算。

此PDF文件为本人制作，请各位爱书之人尊重个人劳动，敬请您不要修改此PDF文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，以免这些资源以后更难以得到。谢谢！